aws

Developer Guide

Amazon DocumentDB



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon DocumentDB: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is Amazon DocumentDB 1
Overview 1
Clusters 3
Instances 4
Regions and AZs 7
Regions 7
Availability zones 8
Pricing 10
Free trial 11
Monitoring 11
Interfaces 11
AWS Management Console 12
AWS CLI 12
MongoDB drivers 12
What's next? 12
How it works 13
Amazon DocumentDB versions 13
Amazon DocumentDB endpoints 15
TLS support 19
Amazon DocumentDB storage 19
Amazon DocumentDB replication 20
Amazon DocumentDB reliability 20
Read preference options 21
TTL deletes 26
Billable resources 26
What is a document database? 29
Use cases 30
Understanding documents 31
Working with documents 37
Get started guide 49
Prerequisites 49
Step 1: Create a cluster 51
Step 2: Connect to your cluster 54
Step 3: Insert and query data 55

Step 4: Explore ... 57
Quick start using CloudFormation ... 57
Prerequisites ... 58
Launching an Amazon DocumentDB CloudFormation stack ... 59
Accessing the Amazon DocumentDB cluster ... 64
Termination protection and deletion protection ... 64
Migrating and upgrading ... 66
Quick start guide ... 66
Prepare the DMS source ... 67
Setup DMS ... 67
Enable DocumentDB compression ... 67
Create a replication task ... 67
Monitor progress ... 68
Additional information ... 68
Migration runbook ... 69
Compatibility ... 69
Workload discovery ... 72
Create new target cluster ... 75
Index migration ... 76
User migration ... 77
Data migration ... 78
Monitoring ... 89
Validation ... 91
Migration from Couchbase Server ... 97
Introduction ... 97
Comparison to Amazon DocumentDB ... 97
Discovery ... 100
Planning ... 105
Migration ... 108
Validation ... 117
Engine version support dates ... 120
Release calendar for Amazon DocumentDB major versions ... 120
Release calendar for Amazon DocumentDB minor versions ... 121
Upgrading a major engine version ... 122
Supported upgrade paths ... 122
Prerequisites ... 123

Best practices ... 125
What changes after upgrading to Amazon DocumentDB 8.0 ... 128
Post-upgrade considerations for clusters upgraded from 3.6 or 4.0 ... 129
Performing the upgrade ... 129
Troubleshooting ... 132
Upgrading a minor engine version ... 132
Before performing a minor version upgrade ... 132
Performing a minor version upgrade ... 133
Verifying the minor version upgrade ... 134
Upgrading using AWS DMS ... 134
Step 1: Enable change streams ... 135
Step 2: Modify the change streams retention duration ... 135
Step 3: Migrate your indexes ... 136
Step 4: Create an AWS DMS replication instance ... 137
Step 5: Create an AWS DMS source endpoint ... 138
Step 6: Create an AWS DMS target endpoint ... 141
Step 7: Create and run a migration task ... 143
Step 8: Changing the application endpoint to the target Amazon DocumentDB cluster ... 145
Using a long-term support (LTS) release ... 145
Current LTS Release ... 145
Who should use LTS ... 146
How to create a new LTS cluster ... 146
How to upgrade from engine version 3.6.0 or 4.0.0 to a 5.0.0 LTS cluster ... 147
How to check if my existing engine version 5.0.0 cluster is on LTS ... 147
Extended Support ... 147
Extended Support overview ... 148
Extended Support charges ... 149
Responsibilities ... 149
Features and configurations ... 151
Engine status ... 151
Cluster types ... 151
Features ... 152
Instance types ... 153
AWS integrations ... 154
Choosing Regions and Availability Zones ... 155
Region availability ... 156

OpenSearch integration ... 158
Amazon OpenSearch Service as a destination ... 158
Limitations ... 162
Developing ... 163
Connect to DocumentDB ... 163
Connecting programmatically ... 164
Connecting from outside an Amazon VPC ... 194
Connecting as a replica set ... 195
Connect using Studio 3T ... 200
Connect using DataGrip ... 210
Connect using Amazon EC2 ... 215
Connect using the JDBC driver ... 250
Connect using the ODBC driver ... 267
Program with Amazon DocumentDB ... 281
DocumentDB Java programming guide ... 281
Using JSON schema validation ... 313
Querying ... 323
Querying documents ... 323
Query plan ... 326
Explain results ... 329
Query planner v2 ... 334
Query planner v3 ... 348
Geospatial data ... 353
Partial index ... 359
Text search ... 366
Indexes ... 374
Index management ... 375
Index types ... 385
Index properties ... 398
Using change streams ... 418
Supported operations ... 419
Expanded events ... 420
Billing ... 420
Limitations ... 421
Enabling change streams ... 422
Using change streams with Python ... 423

Full document lookup ... 426
Using $changeStreamSplitLargeEvent to split large events into fragments ... 427
Resuming a change stream ... 428
Resuming with `startAtOperationTime` ... 429
Resuming with `postBatchResumeToken` ... 431
Transactions ... 432
Modifying log retention duration ... 433
Change streams on secondary instances ... 436
Transactions ... 437
Requirements ... 438
Best practices ... 438
Limitations ... 438
Monitoring and diagnostics ... 439
Transaction isolation level ... 440
Use cases ... 441
Supported commands ... 479
Unsupported capabilities ... 479
Sessions ... 480
Transaction errors ... 481
Retryable writes ... 482
Requirements ... 482
Supported operations ... 483
Limitations ... 483
Enabling ... 484
How it works ... 484
Error handling ... 485
Migration ... 485
Using collation ... 485
Limitations ... 486
Using views ... 486
Best practices ... 487
Aggregator operator compatibility ... 487
Leveraging indexes and views ... 488
Generative AI ... 488
SageMaker Canvas ... 489
Vector search ... 491

Using AWS Lambda with change streams ... 503
Limitations ... 504
MongoDB compatibility ... 505
MongoDB 8.0 compatibility ... 505
What's new in Amazon DocumentDB 8.0 ... 505
Get started with Amazon DocumentDB ... 506
Upgrade or migrate to Amazon DocumentDB 5.0 or 8.0 ... 507
Functional differences ... 507
MongoDB 5.0 compatibility ... 507
What's new in Amazon DocumentDB 5.0 ... 508
What's new in Amazon DocumentDB 5.0.1 ... 508
Get started with Amazon DocumentDB 5.0 ... 506
Upgrade or migrate to Amazon DocumentDB 5.0 ... 507
Functional differences ... 507
MongoDB 4.0 compatibility ... 511
Amazon DocumentDB 4.0 features ... 511
Get started with Amazon DocumentDB 4.0 ... 513
Upgrade or migrate to Amazon DocumentDB 4.0 ... 513
Functional differences ... 514
Functional differences with MongoDB ... 514
Functional benefits of Amazon DocumentDB ... 515
Updated functional differences ... 516
Functional differences with MongoDB ... 520
Supported MongoDB APIs, operations, and data types ... 529
Database commands ... 530
Query and projection operators ... 539
Update operators ... 542
Geospatial ... 544
Cursor methods ... 546
Aggregation pipeline operators ... 548
Data types ... 565
Indexes and index properties ... 566
Aggregation pipeline operators ... 568
Geospatial ... 1203
Query and projection operators ... 1228
Update operators ... 1296

Managing clusters and instances 1351
Operational tasks overview 1351
Adding a replica to an Amazon DocumentDB cluster 1352
Describing clusters and instances 1352
Creating a cluster snapshot 1355
Restoring from a snapshot 1356
Removing an instance from a cluster 1357
Deleting a cluster 1357
Global clusters 1358
What is a global cluster? 1358
How are global clusters useful? 1358
What are the current limitations of global clusters? 1359
Quick start guide 1360
Managing global clusters 1371
Connecting global clusters 1378
Monitoring global clusters 1379
Disaster recovery 1380
Managing clusters 1404
Understanding clusters 1405
Cluster settings 1407
Cluster storage configurations 1410
Determining a cluster's status 1413
Cluster lifecycle 1415
Scaling clusters 1457
Cloning a volume for a cluster 1460
Understanding cluster fault tolerance 1473
Managing instances 1475
Determining an instance's status 1475
Instance lifecycle 1475
Managing instance classes 1499
NVMe-backed instances 1512
Managing subnet groups 1515
Creating a subnet group 1516
Describing a subnet group 1521
Modifying a subnet group 1524
Deleting a subnet group 1527

High availability and replication ... 1528
Read scaling ... 1529
High availability ... 1529
Adding replicas ... 1530
Failover ... 1531
Replication lag ... 1535
Managing events ... 1537
Viewing event categories ... 1537
Viewing Amazon DocumentDB events ... 1540
Managing cluster parameter groups ... 1542
Describing cluster parameter groups ... 1543
Creating cluster parameter groups ... 1551
Modifying cluster parameter groups ... 1554
Modifying clusters to use customized cluster parameter groups ... 1559
Copying cluster parameter groups ... 1560
Resetting cluster parameter groups ... 1562
Deleting cluster parameter groups ... 1565
Cluster parameters reference ... 1568
Understanding endpoints ... 1589
Finding a cluster's endpoints ... 1589
Finding an instance's endpoint ... 1592
Connecting to endpoints ... 1595
Understanding Amazon DocumentDB ARNs ... 1597
Constructing an ARN ... 1597
Finding an ARN ... 1601
Tagging resources ... 1604
Overview of resource tags ... 1604
Tag constraints ... 1605
Adding or updating tags ... 1605
Listing tags ... 1607
Removing tags ... 1608
Maintaining Amazon DocumentDB ... 1610
Maintenance actions for Amazon DocumentDB ... 1611
Version numbering ... 1612
Manage maintenance windows ... 1613
Engine patch notifications ... 1615

View pending maintenance ... 1617
Engine updates ... 1620
Minor versions ... 1623
Operating system updates ... 1624
User-initiated updates ... 1628
Global clusters ... 1629
Understanding service-linked roles ... 1630
Service-linked role permissions ... 1630
Creating a service-linked role ... 1632
Modifying a service-linked role ... 1632
Deleting a service-linked role ... 1632
Supported Regions for Amazon DocumentDB service-linked roles ... 1633
Performance optimization ... 1634
Common performance issues ... 1634
High CPU ... 1634
Connection issues ... 1636
Query running slow ... 1643
Anti-patterns ... 1647
Compound index anti-pattern more than 3 attributes ... 1648
Long running queries ... 1650
Low used and redundant indexes ... 1652
Multi-key indexes with large arrays ... 1655
Performance improvement tips ... 1656
1. Use $match as First Stage in Aggregation Pipelines ... 1656
2. Use $project in Aggregation Pipeline to Minimize Pipeline Data Size ... 1657
3. Enable Document Compression for Lower Storage Costs, I/O Costs, and Improved Query Performance ... 1657
4. Leverage Indexes for Optimal Query Performance ... 1658
5. Optimize Data Models Based on Query Patterns ... 1658
Query plan analysis ... 1661
Overall timing ... 1663
Execution stages ... 1663
Understanding docsExamined in Explain Plan Execution Stats ... 1665
Index-only scans ... 1672
When Amazon DocumentDB uses an index-only scan ... 1673
Requirements for a covered query ... 1675

Restrictions ... 1676
Confirming an index-only scan ... 1676
Best practices ... 1677
Plan cache filters ... 1678
Supported commands ... 1679
Supported query shapes ... 1680
Setting, listing, and clearing filters ... 1681
Verifying that a filter is applied ... 1683
Examples ... 1683
Behavior with hints and index changes ... 1686
Best practices ... 1687
Basic operational guidelines ... 1687
Instance sizing ... 1688
Working with indexes ... 1690
Security best practices ... 1691
Cost optimization ... 1691
Using metrics to identify performance issues ... 1692
TTL and time series workloads ... 1697
Migrations ... 1697
Working with cluster parameter groups ... 1698
Aggregation pipeline queries ... 1698
`batchInsert` and `batchUpdate` ... 1698
Managing indexes ... 1698
Amazon DocumentDB index creation ... 1698
Maintaining Amazon DocumentDB indexes ... 1704
Managing document compression ... 1706
Managing document compression ... 1707
Monitoring document compression ... 1709
Managing dictionary-based compression ... 1710
Performance considerations ... 1711
Enabling dictionary-based compression ... 1711
Getting Started ... 1712
Monitoring ... 1712
Security ... 1718
Password management with AWS Secrets Manager ... 1719
Limitations for Secrets Manager integration with Amazon DocumentDB ... 1719

Overview of managing primary user passwords with AWS Secrets Manager 1720
Enforcing Amazon DocumentDB management of the primary user password in AWS Secrets Manager 1720
Managing the primary user password for a cluster with Secrets Manager 1721
Data protection 1721
Client-side field level encryption 1723
Encrypting data at rest 1729
Encrypting data in transit 1737
Key management 1750
Identity and Access Management 1751
Audience 1751
Authenticating with identities 1752
Managing access using policies 1753
How Amazon DocumentDB works with IAM 1755
Identity-based policy examples 1761
Troubleshooting 1764
Managing access permissions to your Amazon DocumentDB resources 1766
Using identity-based policies (IAM policies) 1771
AWS managed policies for Amazon DocumentDB 1775
Amazon DocumentDB API permissions reference 1794
Cross-service confused deputy prevention 1803
Authentication using IAM identity 1804
Getting started with IAM users and roles 1805
Configuring compute types 1808
Monitoring authentication requests 1809
Using IAM authentication 1809
Drivers supporting IAM 1809
IAM identity authentication FAQ 1810
Authentication using SCRAM-SHA-256 1810
Creating users 1811
Updating users 1812
Authenticating 1813
Managing Amazon DocumentDB users 1813
Primary and `serviceadmin` user 1813
Creating additional users 1814
Automatically rotating passwords 1816

Role-Based Access Control ... 1817
RBAC concepts ... 1818
Getting started with RBAC built-in roles ... 1820
Getting started with RBAC user-defined roles ... 1823
Connecting to Amazon DocumentDB as a User ... 1827
Common commands ... 1829
Functional differences ... 1834
Limits ... 1834
Database access using Role-Based Access Control ... 1835
Updating certificates ... 1844
Updating your application and Amazon DocumentDB cluster ... 1844
Automatic server certificate rotation ... 1849
Troubleshooting ... 1850
Frequently Asked Questions ... 1850
Compliance validation ... 1857
Resilience ... 1858
Infrastructure security ... 1859
VPC endpoints (AWS PrivateLink) ... 1859
Considerations for VPC endpoints ... 1860
Region availability ... 1861
Creating an interface VPC endpoint for Amazon DocumentDB API ... 1862
Creating a VPC endpoint policy for the Amazon DocumentDB API ... 1862
Security best practices ... 1863
Auditing events ... 1864
Supported events ... 1865
Enable auditing ... 1870
Disable auditing ... 1877
Access your events ... 1880
Filter DML events ... 1880
Amazon VPC and Amazon DocumentDB ... 1886
DocumentDB clusters in a VPC ... 1886
Accessing a cluster in a VPC ... 1898
Create an IPv4-only VPC for a cluster ... 1901
Create a dual-stack VPC for a cluster ... 1907
Backing up and restoring ... 1918
Back up and restore: concepts ... 1919

Understanding backup storage usage ... 1921
Dumping, restoring, importing, and exporting data ... 1922
mongodump ... 1923
mongorestore ... 1924
mongoexport ... 1924
mongoimport ... 1925
Tutorial ... 1925
Cluster snapshot considerations ... 1928
Backup storage ... 1929
Backup window ... 1929
Backup retention period ... 1931
Copy cluster snapshot encryption ... 1931
Comparing automatic and manual snapshots ... 1932
Creating a manual cluster snapshot ... 1934
Copying a cluster snapshot ... 1937
Copying shared snapshots ... 1938
Copying snapshots across AWS Regions ... 1938
Limitations ... 1938
Handling encryption ... 1939
Parameter group considerations ... 1939
Copying a cluster snapshot ... 1939
Sharing a cluster snapshot ... 1949
Sharing an encrypted snapshot ... 1950
Sharing a snapshot ... 1953
Restoring from a cluster snapshot ... 1956
Restoring to a point in time ... 1964
Deleting a cluster snapshot ... 1970
Monitoring and logging ... 1972
Monitoring a cluster's status ... 1973
Cluster status values ... 1974
Monitoring a cluster's status ... 1975
Monitoring an instance's status ... 1977
Instance status values ... 1978
Monitoring instance status using the AWS Management Console or AWS CLI ... 1980
Instance health status values ... 1982
Monitoring instance health status using the AWS Management Console ... 1982

Viewing Amazon DocumentDB recommendations ... 1984
Event subscriptions ... 1987
Subscribing to events ... 1988
Manage subscriptions ... 1990
Categories and messages ... 1993
Monitoring Amazon DocumentDB with CloudWatch ... 1996
Amazon DocumentDB metrics ... 1997
Viewing CloudWatch data ... 2016
Amazon DocumentDB dimensions ... 2022
Monitoring opcounter metrics ... 2022
Monitoring database connections ... 2022
Logging Amazon DocumentDB API calls with CloudTrail ... 2023
Amazon DocumentDB Information in CloudTrail ... 2023
Profiling operations ... 2024
Supported operations ... 2025
Limitations ... 2025
Enabling the profiler ... 2026
Disabling the profiler ... 2030
Disabling profiler logs export ... 2031
Accessing your profiler logs ... 2034
Common queries ... 2034
Monitoring with Performance Insights ... 2035
Performance Insights concepts ... 2036
Enabling and disabling Performance Insights ... 2039
Configuring access policies for Performance Insights ... 2042
Analyzing metrics with the Performance Insights dashboard ... 2046
Retrieving metrics with the Performance Insights API ... 2064
Amazon CloudWatch metrics for Performance Insights ... 2078
Performance Insights for counter metrics ... 2081
Serverless ... 2083
Serverless use cases ... 2083
Using Amazon DocumentDB serverless for existing provisioned workloads ... 2085
Advantages of Amazon DocumentDB serverless ... 2085
How serverless works ... 2086
Overview ... 2087
Configurations for Amazon DocumentDB clusters ... 2088

Amazon DocumentDB serverless scaling capacity 2089
Amazon DocumentDB serverless scaling 2091
Idle state (0.5 DCUs) 2093
Serverless requirements and limitations 2094
Feature requirements 2094
Creating a cluster that uses serverless 2095
Creating an Amazon DocumentDB serverless cluster 2095
Adding an Amazon DocumentDB serverless instance 2098
Migrating to serverless 2099
Migrating existing DocumentDB clusters to serverless 2099
Migrating from MongoDB to DocumentDB serverless 2104
Managing serverless 2104
Viewing and modifying a cluster's scaling capacity range configuration 2104
Viewing and modifying the promotion tier of serverless readers 2108
Checking and upgrading the platform version of a serverless cluster 2110
Serverless instance limits 2113
Serverless scaling configuration 2116
Choosing the scaling capacity range for a DocumentDB serverless cluster 2116
Choosing the `MinCapacity` setting for a DocumentDB serverless cluster 2116
Choosing the `MaxCapacity` setting for a DocumentDB serverless cluster 2118
Avoiding out-of-memory errors 2120
Why is my serverless instance not scaling down? 2122
Monitoring serverless 2122
Out of Memory: incompatible-parameters status 2123
Amazon CloudWatch metrics for DocumentDB serverless 2123
Monitoring DocumentDB serverless performance with Performance Insights 2127
Elastic Clusters 2129
Elastic cluster use cases 2130
User profiles 2130
Content management and historical records 2130
Advantages of elastic clusters 2130
AWS service integration 2130
Region and version availability 2131
Region availability for elastic clusters 2131
Version availability 2132
Limitations 2132

Elastic cluster management ... 2132
Query and write operations ... 2133
Collection and index management ... 2133
Administration and diagnostics ... 2133
Opt-in features ... 2134
How it works ... 2134
Amazon DocumentDB elastic cluster sharding ... 2134
Elastic cluster migration ... 2138
Elastic cluster scaling ... 2138
Elastic cluster reliability ... 2138
Elastic cluster storage and availability ... 2138
Functional differences between Amazon DocumentDB 4.0 and elastic clusters ... 2139
Get started ... 2140
Prerequisites ... 2141
Step 1: Create an elastic cluster ... 2142
Step 2: Connect to your elastic cluster ... 2148
Step 3: Shard your collection, insert and query data ... 2149
Step 4: Explore ... 2153
Best practices ... 2153
Choosing shard keys ... 2153
Connection management ... 2154
Unsharded collections ... 2154
Scaling elastic clusters ... 2154
Monitoring elastic clusters ... 2155
Managing elastic clusters ... 2155
Modifying elastic cluster configurations ... 2156
Monitoring an elastic cluster ... 2159
Deleting an elastic cluster ... 2163
Managing elastic cluster snapshots ... 2165
Stopping and starting an elastic cluster ... 2179
Maintaining elastic clusters ... 2184
Data encryption at rest ... 2192
How Amazon DocumentDB elastic clusters use grants in AWS KMS ... 2193
Create a customer managed key ... 2194
Monitoring your encryption keys for Amazon DocumentDB elastic clusters ... 2195
Learn more ... 2201

Service-linked roles ... 2201
Service-linked role permissions for elastic clusters ... 2202
Quotas ... 2206
Supported instance types ... 2206
Supported Regions ... 2208
Regional quotas ... 2210
Aggregation quotas ... 2213
Cluster quotas ... 2213
Instance quotas ... 2215
Naming constraints ... 2222
TTL constraints ... 2224
Elastic cluster quotas ... 2224
Elastic cluster shard quotas ... 2226
Elastic cluster CPU, memory, connection, and cursor quotas per shard ... 2226
Troubleshooting ... 2228
Connectivity issues ... 2228
Can't connect to an Amazon DocumentDB endpoint ... 2228
Testing a connection to an Amazon DocumentDB instance ... 2233
Connecting to an invalid endpoint ... 2234
Driver configuration impacting number of connections ... 2235
Indexes ... 2235
Index build fails ... 2235
Background index build latency issues and fails ... 2236
Index bloat ... 2236
Performance and resource utilization ... 2237
View insert, update, and delete statistics ... 2238
Analyze cache performance ... 2240
Find and terminate long running or blocked queries ... 2241
See a query plan and optimize a query ... 2242
How can I see a query plan in elastic clusters? ... 2245
List all running operations on an instance ... 2247
Know when a query is making progress ... 2249
Determine why a system suddenly runs slowly ... 2252
Determine the Cause of High CPU Utilization ... 2254
Find the open cursors on an instance ... 2255
View the current Amazon DocumentDB engine version ... 2255

Analyze index usage and identify unused indexes 2256
Identify missed indexes 2258
How do I determine database collection bloat? 2259
Summary of useful queries 2261
Garbage collection 2262
Understanding Garbage collection in Amazon DocumentDB 2262
Garbage collection process 2263
Storage architecture and extended storage 2263
Monitoring garbage collection 2265
Example collStats output 2267
Frequently asked questions 2269
Error codes 2272
Release notes 2303
Amazon DocumentDB engine version release notes 2304
General Amazon DocumentDB release notes 2366
Amazon DocumentDB elastic clusters release notes 2373
Document history 2377

What is Amazon DocumentDB (with MongoDB compatibility)

Amazon DocumentDB (with MongoDB compatibility) is a fast, reliable, and fully managed database service. Amazon DocumentDB makes it easy to set up, operate, and scale MongoDB-compatible databases in the cloud. With Amazon DocumentDB, you can run the same application code and use the same drivers and tools that you use with MongoDB.

Before using Amazon DocumentDB, you should review the concepts and features described in How it works. After that, complete the steps in Get started guide.

Topics

- Overview of Amazon DocumentDB
- Clusters
- Instances
- Regions and Availability Zones
- Amazon DocumentDB pricing
- Monitoring
- Interfaces
- What's next?
- Amazon DocumentDB: how it works
- What is a document database?

Overview of Amazon DocumentDB

The following are some high-level features of Amazon DocumentDB:

- Amazon DocumentDB supports two types of clusters: instance-based clusters and elastic clusters. Elastic clusters support workloads with millions of reads/writes per second and petabytes of storage capacity. For more information about elastic clusters, see Using Amazon DocumentDB Elastic Clusters. The content below refers to Amazon DocumentDB instance-based clusters .
- Amazon DocumentDB automatically grows the size of your storage volume as your database storage needs grow. Your storage volume grows in increments of 10 GB, up to a maximum of 256

TiB for Engine Version 8.0 and beyond (128 TiB for earlier engine versions). You don't need to provision any excess storage for your cluster to handle future growth.

- With Amazon DocumentDB, you can increase read throughput to support high-volume application requests by creating up to 15 replica instances. Amazon DocumentDB replicas share the same underlying storage, lowering costs and avoiding the need to perform writes at the replica nodes. This capability frees up more processing power to serve read requests and reduces the replica lag time—often down to single digit milliseconds. You can add replicas in minutes regardless of the storage volume size. Amazon DocumentDB also provides a reader endpoint, so the application can connect without having to track replicas as they are added and removed.
- Amazon DocumentDB lets you scale the compute and memory resources for each of your instances up or down. Compute scaling operations typically complete in a few minutes.
- Amazon DocumentDB runs in Amazon Virtual Private Cloud (Amazon VPC), so you can isolate your database in your own virtual network. You can also configure firewall settings to control network access to your cluster.
- Amazon DocumentDB continuously monitors the health of your cluster. On an instance failure, Amazon DocumentDB automatically restarts the instance and associated processes. Amazon DocumentDB doesn't require a crash recovery replay of database redo logs, which greatly reduces restart times. Amazon DocumentDB also isolates the database cache from the database process, enabling the cache to survive an instance restart.
- On instance failure, Amazon DocumentDB automates failover to one of up to 15 Amazon DocumentDB replicas that you create in other Availability Zones. If no replicas have been provisioned and a failure occurs, Amazon DocumentDB tries to create a new Amazon DocumentDB instance automatically.
- The backup capability in Amazon DocumentDB enables point-in-time recovery for your cluster. This feature allows you to restore your cluster to any second during your retention period, up to the last 5 minutes. You can configure your automatic backup retention period up to 35 days. Automated backups are stored in Amazon Simple Storage Service (Amazon S3), which is designed for 99.999999999% durability. Amazon DocumentDB backups are automatic, incremental, and continuous, and they have no impact on your cluster performance.
- With Amazon DocumentDB, you can encrypt your databases using keys that you create and control through AWS Key Management Service (AWS KMS). On a database cluster running with Amazon DocumentDB encryption, data stored at rest in the underlying storage is encrypted. The automated backups, snapshots, and replicas in the same cluster are also encrypted.
- Amazon DocumentDB is authorized under Federal Risk and Authorization Management Program (FedRAMP). It has FedRAMP High authorization for AWS GovCloud (US) Regions and FedRAMP

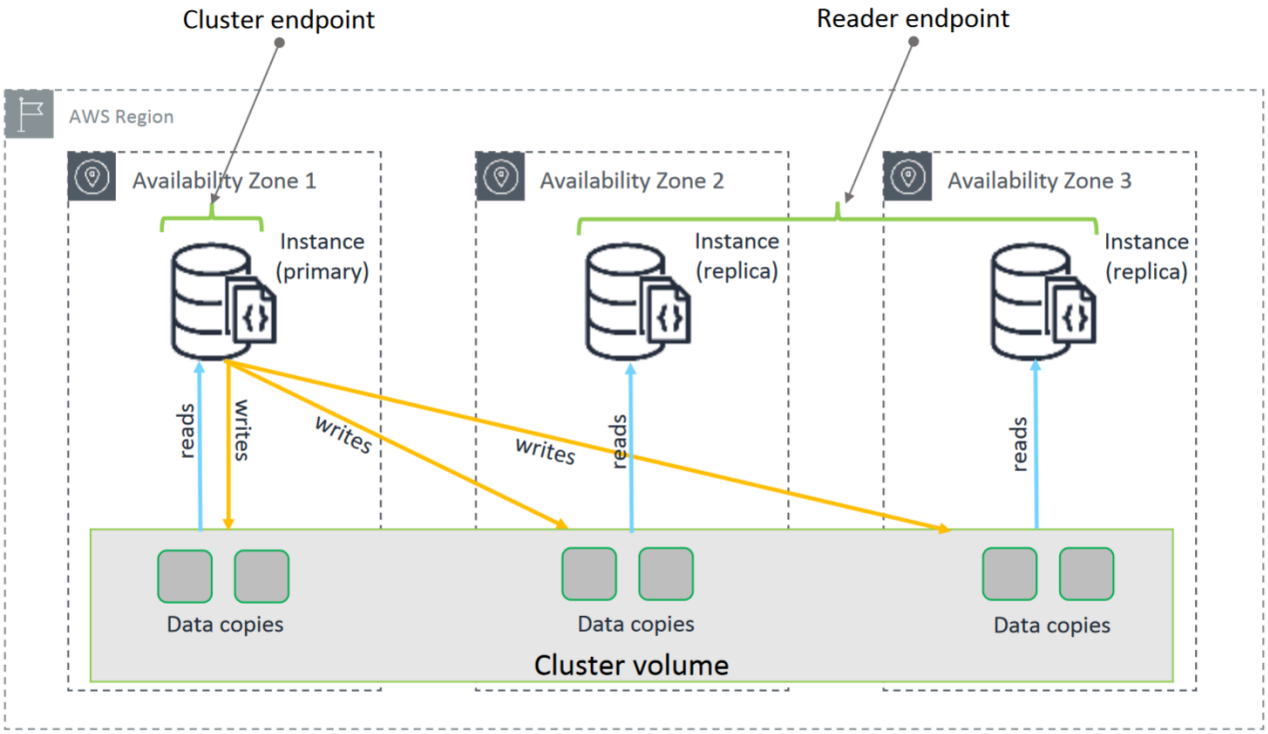
Moderate authorization for AWS US East/West Regions. For details about AWS and compliance efforts, see AWS Services in Scope by Compliance Program.

If you are new to AWS services, use the following resources to learn more:

- AWS offers services for computing, databases, storage, analytics, and other functionality. For an overview of all AWS services, see Cloud Computing with Amazon Web Services.
- AWS provides a number of database services. For guidance on which service is best for your environment, see Databases on AWS.

Clusters

A *cluster* consists of 0 to 16 instances and a cluster storage volume that manages the data for those instances. All writes are done through the primary instance. All instances (primary and replicas) support reads. The cluster's data is stored in the cluster volume with copies in three different Availability Zones.



Amazon DocumentDB 5.0 instance-based clusters support two storage configurations for a database cluster: Amazon DocumentDB standard and Amazon DocumentDB I/O-optimized. For more information see Amazon DocumentDB cluster storage configurations.

Instances

An Amazon DocumentDB instance is an isolated database environment in the cloud. An instance can contain multiple user-created databases. You can create and modify an instance using the AWS Management Console or the AWS CLI.

The computation and memory capacity of an instance are determined by its *instance class*. You can select the instance that best meets your needs. If your needs change over time, you can choose a different instance class. For instance class specifications, see Instance class specifications.

Amazon DocumentDB instances run only in the Amazon VPC environment. Amazon VPC gives you control of your virtual networking environment: You can choose your own IP address range, create subnets, and configure routing and access control lists (ACLs).

Before you can create Amazon DocumentDB instances, you must create a cluster to contain the instances.

Not all instance classes are supported in every AWS Region. The following table shows which instance classes are supported in each Region.

> **Note**
>
> For a complete list of instance types supported by Amazon DocumentDB in each instance class, see Instance class specifications.

Supported instance classes by Region

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
US East (Ohio)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
US East (N. Virginia)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
US West (Oregon)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Africa (Cape Town)			Supported	Supported		Supported	Supported	Supported
South America (São Paulo)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Hong Kong)	Supported		Supported	Supported		Supported	Supported	Supported
Asia Pacific (Taipei)			Supported			Supported	Supported	
Asia Pacific (Hyderabad)			Supported	Supported		Supported	Supported	Supported
Asia Pacific (Malaysia)			Supported			Supported	Supported	Supported
Asia Pacific (Mumbai)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Osaka)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Seoul)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Sydney)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Jakarta)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Melbourne)			Supported	Supported		Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Asia Pacific (New Zealand)			Supported			Supported	Supported	
Asia Pacific (Singapore)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Thailand)			Supported			Supported	Supported	Supported
Asia Pacific (Tokyo)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Canada (Central)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Canada West (Calgary)			Supported	Supported		Supported	Supported	Supported
Europe (Frankfurt)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Zurich)		Supported	Supported	Supported		Supported	Supported	Supported
Europe (Ireland)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
Europe (London)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Milan)			Supported	Supported		Supported	Supported	Supported
Europe (Paris)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Spain)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Stockholm)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Mexico (Central)			Supported			Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Middle East (UAE)			Supported	Supported		Supported	Supported	Supported
China (Beijing)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
China (Ningxia)	Supported		Supported	Supported		Supported	Supported	Supported
Israel (Tel Aviv)			Supported	Supported		Supported	Supported	Supported
AWS GovCloud (US-West)	Supported	Supported	Supported	Supported			Supported	Supported
AWS GovCloud (US-East)		Supported	Supported	Supported		Supported	Supported	Supported

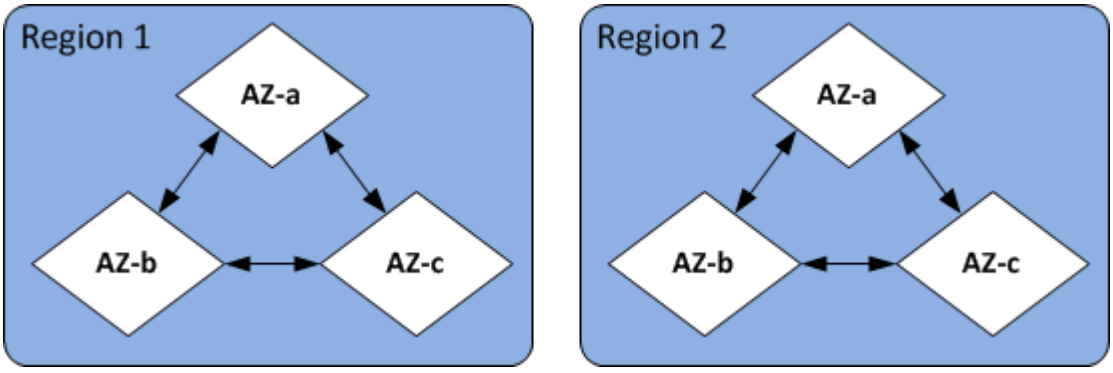
Regions and Availability Zones

AWS Regions and Availability Zones define the physical locations of your cluster and instances.

Regions

AWS Cloud computing resources are housed in highly available data center facilities in different areas of the world (for example, North America, Europe, or Asia). Each data center location is called an *AWS Region*.

Each AWS Region is designed to be completely isolated from the other AWS Regions. Within each are multiple Availability Zones. By launching your nodes in different Availability Zones, you can achieve the greatest possible fault tolerance. The following diagram shows a high-level view of how AWS Regions and Availability Zones work.



Availability zones

Each AWS Region contains multiple distinct locations called *Availability Zones*. Each Availability Zone is engineered to be isolated from failures in other Availability Zones, and to provide inexpensive, low-latency network connectivity to other Availability Zones in the same Region. By launching instances for a given cluster in multiple Availability Zones, you can protect your applications from the unlikely event of an Availability Zone failing.

The Amazon DocumentDB architecture separates storage and compute. For the storage layer, Amazon DocumentDB replicates six copies of your data across three AWS Availability Zones. As an example, if you are launching an Amazon DocumentDB cluster in a Region that only supports two Availability Zones, your data storage will be replicated six ways across three Availability Zones but your compute instances will only be available in two Availability Zones.

The following table lists the number of Availability Zones that you can use in a given AWS Region to provision compute instances for your cluster.

Region Name	Region	Availability Zones (compute)
US East (Ohio)	`us-east-2`	3
US East (N. Virginia)	`us-east-1`	6
US West (Oregon)	`us-west-2`	4
Africa (Cape Town)	`af-south-1`	3
South America (São Paulo)	`sa-east-1`	3

Region Name	Region	Availability Zones (compute)
Asia Pacific (Hong Kong)	ap-east-1	3
Asia Pacific (Taipei)	ap-east-2	3
Asia Pacific (Hyderabad)	ap-south-2	3
Asia Pacific (Malaysia)	ap-southeast-5	3
Asia Pacific (Mumbai)	ap-south-1	3
Asia Pacific (Osaka)	ap-northeast-3	3
Asia Pacific (Seoul)	ap-northeast-2	4
Asia Pacific (Singapore)	ap-southeast-1	3
Asia Pacific (Sydney)	ap-southeast-2	3
Asia Pacific (Jakarta)	ap-southeast-3	3
Asia Pacific (Melbourne)	ap-southeast-4	3
Asia Pacific (New Zealand)	ap-southeast-6	3
Asia Pacific (Thailand)	ap-southeast-7	3
Asia Pacific (Tokyo)	ap-northeast-1	3
Canada (Central)	ca-central-1	3
Canada West (Calgary)	ca-west-1	3
China (Beijing) Region	cn-north-1	3

Region Name	Region	Availability Zones (compute)
China (Ningxia)	`cn-northwest-1`	3
Europe (Frankfurt)	`eu-central-1`	3
Europe (Zurich)	`eu-central-2`	3
Europe (Ireland)	`eu-west-1`	3
Europe (London)	`eu-west-2`	3
Europe (Milan)	`eu-south-1`	3
Europe (Paris)	`eu-west-3`	3
Europe (Spain)	`eu-south-2`	3
Europe (Stockholm)	`eu-north-1`	3
Mexico (Central)	`mx-central-1`	3
Middle East (UAE)	`me-central-1`	3
Israel (Tel Aviv)	`il-central-1`	3
AWS GovCloud (US-West)	`us-gov-west-1`	3
AWS GovCloud (US-East)	`us-gov-east-1`	3

Amazon DocumentDB pricing

Amazon DocumentDB clusters are billed based on the following components:

- **Instance hours (per hour)**—Based on the instance class of the instance (for example, `db.r5.xlarge`). Pricing is listed on a per-hour basis, but bills are calculated down to the second and show times in decimal form. Amazon DocumentDB usage is billed in one second increments, with a minimum of 10 minutes. For more information, see Managing instance classes.

- **I/O requests (per 1 million requests per month)** — Total number of storage I/O requests that you make in a billing cycle.
- **Backup storage (per GiB per month)** — Backup storage is the storage that is associated with automated database backups and any active database snapshots that you have taken. Increasing your backup retention period or taking additional database snapshots increases the backup storage consumed by your database. Backup storage is metered in GB-months and per second does not apply. For more information, see Backing up and restoring in Amazon DocumentDB.
- **Data transfer (per GB)** — Data transfer in and out of your instance from or to the internet or other AWS Regions.

For detailed information, see Amazon DocumentDB pricing.

Free trial

You can try Amazon DocumentDB for free using the 1-month free trial. For more information, see Free trial on the Amazon DocumentDB pricing site, or see the Amazon DocumentDB free trial FAQ.

Monitoring

There are several ways that you can track the performance and health of an instance. You can use the free Amazon CloudWatch service to monitor the performance and health of an instance. You can find performance charts on the Amazon DocumentDB console. You can subscribe to Amazon DocumentDB events to be notified when changes occur with an instance, snapshot, parameter group, or security group.

For more information, see the following:

- Monitoring Amazon DocumentDB with CloudWatch
- Logging Amazon DocumentDB API calls with AWS CloudTrail

Interfaces

There are multiple ways for you to interact with Amazon DocumentDB, including the AWS Management Console and the AWS CLI.

AWS Management Console

The AWS Management Console is a simple web-based user interface. You can manage your clusters and instances from the console with no programming required. To access the Amazon DocumentDB console, sign in to the AWS Management Console and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

AWS CLI

You can use the AWS Command Line Interface (AWS CLI) to manage your Amazon DocumentDB clusters and instances. With minimal configuration, you can start using all of the functionality provided by the Amazon DocumentDB console from your favorite terminal program.

- To install the AWS CLI, see Installing the AWS Command Line Interface.
- To begin using the AWS CLI for Amazon DocumentDB, see AWS Command Line Interface Reference for Amazon DocumentDB.

MongoDB drivers

For developing and writing applications against an Amazon DocumentDB cluster, you can also use the MongoDB drivers with Amazon DocumentDB. For more information, see the MongoDB shell tab in Connecting with TLS enabled or Connecting with TLS disabled.

What's next?

The preceding sections introduced you to the basic infrastructure components that Amazon DocumentDB offers. What should you do next? Depending upon your circumstances, see one of the following topics to get started:

- Get started with Amazon DocumentDB by creating a cluster and instance using CloudFormation Amazon DocumentDB quick start using CloudFormation.
- Get started with Amazon DocumentDB by creating a cluster and instance using the instructions in our Get started guide.
- Get started with Amazon DocumentDB by creating an elastic cluster using the instructions in Get started with Amazon DocumentDB elastic clusters.
- Migrate your MongoDB implementation to Amazon DocumentDB using the guidance at Migrating and upgrading Amazon DocumentDB

Amazon DocumentDB: how it works

Amazon DocumentDB (with MongoDB compatibility) is a fully managed, MongoDB-compatible database service. With Amazon DocumentDB, you can run the same application code and use the same drivers and tools that you use with MongoDB. Amazon DocumentDB is compatible with MongoDB 3.6, 4.0, 5.0, and 8.0.

Topics

- Amazon DocumentDB versions
- Amazon DocumentDB endpoints
- TLS support
- Amazon DocumentDB storage
- Amazon DocumentDB replication
- Amazon DocumentDB reliability
- Read preference options
- TTL deletes
- Billable resources

Amazon DocumentDB versions

> **Note**
>
> Minor version numbering applies to Amazon DocumentDB 5.0 and later. Amazon DocumentDB 3.6 and 4.0 use only major versions with engine patches.

Starting with Amazon DocumentDB 5.0, the engine version uses a four-level numbering scheme: *major.major.minor.patch*.

- **Major version**—Identified by the first two numbers of the version (for example, 5.0). The major version represents the MongoDB compatibility level. Major versions introduce new features, performance improvements, and MongoDB compatibility updates. Upgrading between major versions (for example, from 4.0 to 5.0) requires a major version upgrade (MVU). For more information, see Amazon DocumentDB in-place major version upgrade.

- **Minor version**—Identified by the third number of the version (for example, the 1 in 5.0.1). Minor versions include improvements such as new features, security fixes, and bug fixes within the same major version. You can upgrade to a new minor version manually. For more information, see Amazon DocumentDB minor version upgrade.
- **Patch version**—Engine patches are applied within a minor version and contain critical security fixes and bug fixes. The patch version is tracked by the engine patch version (for example, `3.0.17983`), which you can check by running `db.runCommand({getEngineVersion: 1})`. Patches are applied during your cluster's maintenance window. For more information, see Maintaining Amazon DocumentDB.

When you create a new cluster, you can specify any available engine version. For information about available versions and support dates, see Amazon DocumentDB engine version support dates.

When you use Amazon DocumentDB, you begin by creating a *cluster*. A cluster consists of zero or more database instances and a cluster volume that manages the data for those instances. An Amazon DocumentDB *cluster volume* is a virtual database storage volume that spans multiple Availability Zones. Each Availability Zone has a copy of the cluster data.

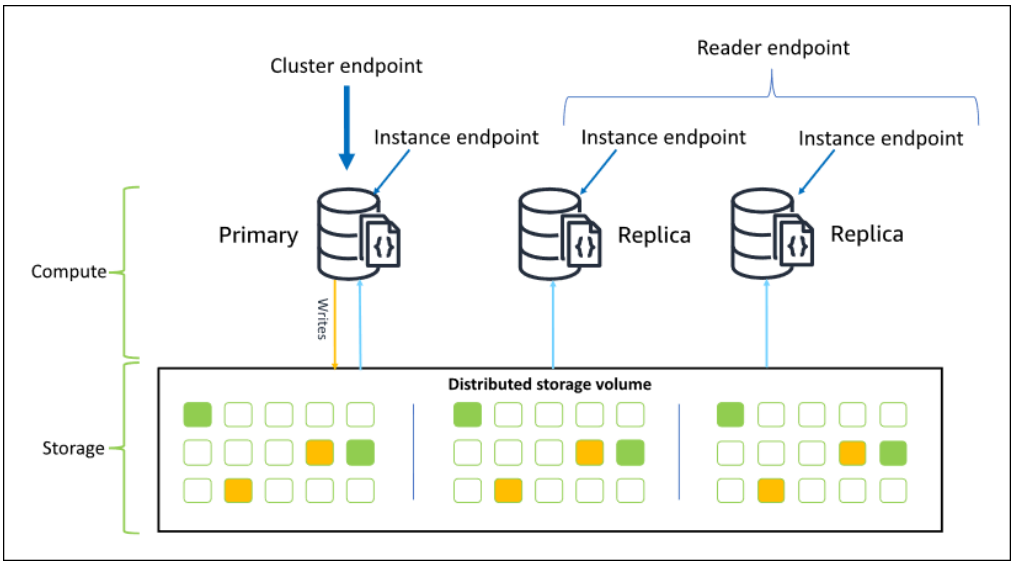
An Amazon DocumentDB cluster consists of two components:

- **Cluster volume**—Uses a cloud-native storage service to replicate data six ways across three Availability Zones, providing highly durable and available storage. An Amazon DocumentDB cluster has exactly one cluster volume, which can store up to 256 TiB of data for Engine Version 8.0 and beyond (128 TiB for earlier engine versions).
- **Instances**—Provide the processing power for the database, writing data to, and reading data from, the cluster storage volume. An Amazon DocumentDB cluster can have 0–16 instances.

Instances serve one of two roles:

- **Primary instance**—Supports read and write operations, and performs all the data modifications to the cluster volume. Each Amazon DocumentDB cluster has one primary instance.
- **Replica instance**—Supports only read operations. An Amazon DocumentDB cluster can have up to 15 replicas in addition to the primary instance. Having multiple replicas enables you to distribute read workloads. In addition, by placing replicas in separate Availability Zones, you also increase your cluster availability.

The following diagram illustrates the relationship between the cluster volume, the primary instance, and replicas in an Amazon DocumentDB cluster:



Cluster instances do not need to be of the same instance class, and they can be provisioned and terminated as desired. This architecture lets you scale your cluster's compute capacity independently of its storage.

When your application writes data to the primary instance, the primary executes a durable write to the cluster volume. It then replicates the state of that write (not the data) to each active replica. Amazon DocumentDB replicas do not participate in processing writes, and thus Amazon DocumentDB replicas are advantageous for read scaling. Reads from Amazon DocumentDB replicas are eventually consistent with minimal replica lag—usually less than 100 milliseconds after the primary instance writes the data. Reads from the replicas are guaranteed to be read in the order in which they were written to the primary. Replica lag varies depending on the rate of data change, and periods of high write activity might increase the replica lag. For more information, see the `ReplicationLag` metrics at Amazon DocumentDB metrics.

Amazon DocumentDB endpoints

Amazon DocumentDB provides multiple connection options to serve a wide range of use cases. To connect to an instance in an Amazon DocumentDB cluster, you specify the instance's endpoint. An *endpoint* is a host address and a port number, separated by a colon.

Connect to your cluster using the cluster endpoint in replica set mode (see Connecting to Amazon DocumentDB as a replica set). Use this approach unless you must connect to the reader endpoint or an instance endpoint. To route requests to your replicas, choose a driver read preference setting

that maximizes read scaling while meeting your application's read consistency requirements. The `secondaryPreferred` read preference enables replica reads and frees up the primary instance to do more work.

The following endpoints are available from an Amazon DocumentDB cluster.

Cluster endpoint

The *cluster endpoint* connects to your cluster's current primary instance. The cluster endpoint can be used for read and write operations. An Amazon DocumentDB cluster has exactly one cluster endpoint.

The cluster endpoint provides failover support for read and write connections to the cluster. If your cluster's current primary instance fails, and your cluster has at least one active read replica, the cluster endpoint automatically redirects connection requests to a new primary instance. When connecting to your Amazon DocumentDB cluster, use the cluster endpoint in replica set mode (see Connecting to Amazon DocumentDB as a replica set).

The following is an example Amazon DocumentDB cluster endpoint:

```
sample-cluster.cluster-123456789012.us-east-1.docdb.amazonaws.com:27017
```

The following is an example connection string using this cluster endpoint:

```
mongodb://username:password@sample-cluster.cluster-123456789012.us-
east-1.docdb.amazonaws.com:27017
```

For information about finding a cluster's endpoints, see Finding a cluster's endpoints.

Reader endpoint

The *reader endpoint* load balances read-only connections across all available replicas in your cluster. A cluster reader endpoint will perform as the cluster endpoint if you are connecting through `replicaSet` mode, meaning in the connection string, the replica set parameter is `&replicaSet=rs0`. In this case, you will be able to perform write operations on the primary. However, if you connect to the cluster specifying `directConnection=true`, then attempting to perform a write operation over a connection to the reader endpoint results in an error. An Amazon DocumentDB cluster has exactly one reader endpoint.

If the cluster contains only one (primary) instance, the reader endpoint connects to the primary instance. When you add a replica instance to your Amazon DocumentDB cluster, the reader endpoint opens read-only connections to the new replica after it is active.

The following is an example reader endpoint for an Amazon DocumentDB cluster:

```
sample-cluster.cluster-ro-123456789012.us-east-1.docdb.amazonaws.com:27017
```

The following is an example connection string using a reader endpoint:

```
mongodb://username:password@sample-cluster.cluster-ro-123456789012.us-
east-1.docdb.amazonaws.com:27017
```

The reader endpoint load balances read-only connections, not read requests. If some reader endpoint connections are more heavily used than others, your read requests might not be equally balanced among instances in the cluster. It is recommended to distribute requests by connecting to the cluster endpoint as a replica set and utilizing the secondaryPreferred read preference option.

For information about finding a cluster's endpoints, see Finding a cluster's endpoints.

Instance endpoint

An *instance endpoint* connects to a specific instance within your cluster. The instance endpoint for the current primary instance can be used for read and write operations. However, attempting to perform write operations to an instance endpoint for a read replica results in an error. An Amazon DocumentDB cluster has one instance endpoint per active instance.

An instance endpoint provides direct control over connections to a specific instance for scenarios in which the cluster endpoint or reader endpoint might not be appropriate. An example use case is provisioning for a periodic read-only analytics workload. You can provision a larger-than-normal replica instance, connect directly to the new larger instance with its instance endpoint, run the analytics queries, and then terminate the instance. Using the instance endpoint keeps the analytics traffic from impacting other cluster instances.

The following is an example instance endpoint for a single instance in an Amazon DocumentDB cluster:

```
sample-instance.123456789012.us-east-1.docdb.amazonaws.com:27017
```

The following is an example connection string using this instance endpoint:

```
mongodb://username:password@sample-instance.123456789012.us-
east-1.docdb.amazonaws.com:27017
```

> **Note**
>
> An instance's role as primary or replica can change due to a failover event. Your applications should never assume that a particular instance endpoint is the primary instance. Don't connect to instance endpoints for production applications. Instead, use the cluster endpoint in replica set mode (see Connecting to Amazon DocumentDB as a replica set). For more advanced control of instance failover priority, see Understanding Amazon DocumentDB cluster fault tolerance.

For information about finding a cluster's endpoints, see Finding an instance's endpoint.

Replica set mode

You can connect to your Amazon DocumentDB cluster endpoint in replica set mode by specifying the replica set name `rs0`. Connecting in replica set mode provides the ability to specify the Read Concern, Write Concern, and Read Preference options. For more information, see Read consistency.

The following is an example connection string connecting in replica set mode:

```
mongodb://username:password@sample-cluster.cluster-123456789012.us-
east-1.docdb.amazonaws.com:27017/?replicaSet=rs0
```

When you connect in replica set mode, your Amazon DocumentDB cluster appears to your drivers and clients as a replica set. Instances added and removed from your Amazon DocumentDB cluster are reflected automatically in the replica set configuration.

Each Amazon DocumentDB cluster consists of a single replica set with the default name `rs0`. The replica set name cannot be modified.

Connecting to the cluster endpoint in replica set mode is the recommended method for general use.

> **Note**
>
> All instances in an Amazon DocumentDB cluster listen on the same TCP port for connections.

TLS support

For more details on connecting to Amazon DocumentDB using Transport Layer Security (TLS), see Encrypting data in transit.

Amazon DocumentDB storage

Amazon DocumentDB data is stored in a *cluster volume*, which is a single, virtual volume that uses solid state drives (SSDs). A cluster volume consists of six copies of your data, which are replicated automatically across multiple Availability Zones in a single AWS Region. This replication helps ensure that your data is highly durable, with less possibility of data loss. It also helps ensure that your cluster is more available during a failover because copies of your data already exist in other Availability Zones. These copies can continue to serve data requests to the instances in your Amazon DocumentDB cluster.

How data storage is billed

Amazon DocumentDB automatically increases the size of a cluster volume as the amount of data increases. An Amazon DocumentDB cluster volume can grow to a maximum size of 256 TiB for Engine Version 8.0 and beyond (128 TiB for earlier engine versions). However, you are only charged for the space that you use in an Amazon DocumentDB cluster volume. Starting with Amazon DocumentDB 4.0, when data is removed, such as by dropping a collection or index, the overall allocated space decreases by a comparable amount. Thus, you can reduce storage charges by deleting collections, indexes, and databases that you no longer need. In Amazon DocumentDB version 3.6, the cluster volume can reuse space that's freed up when you remove data, but the volume itself never decreases in size. As a result in version 3.6, you may not witness any change in storage when you drop a collection or index, even though the freed up space is reused.

> **Note**
>
> With Amazon DocumentDB 3.6, storage costs are based on the storage "high water mark" (the maximum amount that was allocated for the Amazon DocumentDB cluster at any point in time). You can manage costs by avoiding ETL practices that create large

volumes of temporary information, or that load large volumes of new data prior to removing unneeded older data. If removing data from an Amazon DocumentDB cluster results in a substantial amount of allocated but unused space, resetting the high water mark requires doing a logical data dump and restore to a new cluster, using a tool such as `mongodump` or `mongorestore`. Creating and restoring a snapshot does not reduce the allocated storage because the physical layout of the underlying storage remains the same in the restored snapshot.

Note

Using utilities like `mongodump` and `mongorestore` incur I/O charges based on the sizes of the data that is being read and written to the storage volume.

For information about Amazon DocumentDB data storage and I/O pricing, see Amazon DocumentDB (with MongoDB compatibility) pricing and Pricing FAQs.

Amazon DocumentDB replication

In an Amazon DocumentDB cluster, each replica instance exposes an independent endpoint. These replica endpoints provide read-only access to the data in the cluster volume. They enable you to scale the read workload for your data over multiple replicated instances. They also help improve the performance of data reads and increase the availability of the data in your Amazon DocumentDB cluster. Amazon DocumentDB replicas are also failover targets and are quickly promoted if the primary instance for your Amazon DocumentDB cluster fails.

Amazon DocumentDB reliability

Amazon DocumentDB is designed to be reliable, durable, and fault tolerant. (To improve availability, you should configure your Amazon DocumentDB cluster so that it has multiple replica instances in different Availability Zones.) Amazon DocumentDB includes several automatic features that make it a reliable database solution.

Storage auto-repair

Amazon DocumentDB maintains multiple copies of your data in three Availability Zones, greatly reducing the chance of losing data due to a storage failure. Amazon DocumentDB automatically

detects failures in the cluster volume. When a segment of a cluster volume fails, Amazon DocumentDB immediately repairs the segment. It uses the data from the other volumes that make up the cluster volume to help ensure that the data in the repaired segment is current. As a result, Amazon DocumentDB avoids data loss and reduces the need to perform a point-in-time restore to recover from an instance failure.

Survivable cache warming

Amazon DocumentDB manages its page cache in a separate process from the database so that the page cache can survive independently of the database. In the unlikely event of a database failure, the page cache remains in memory. This ensures that the buffer pool is warmed with the most current state when the database restarts.

Crash recovery

Amazon DocumentDB is designed to recover from a crash almost instantaneously, and to continue serving your application data. Amazon DocumentDB performs crash recovery asynchronously on parallel threads so that your database is open and available almost immediately after a crash.

Resource governance

Amazon DocumentDB safeguards resources for critical processes such as health checks. When an instance experiences high memory pressure, Amazon DocumentDB throttles requests. Some operations might be queued until the memory pressure subsides. If memory pressure continues, queued operations might time out.

Monitor for low-memory throttling with the following CloudWatch metrics: `LowMemThrottleQueueDepth`, `LowMemThrottleMaxQueueDepth`, `LowMemNumOperationsThrottled`, `LowMemNumOperationsTimedOut`. For more information, see Monitoring Amazon DocumentDB with CloudWatch.

If you see sustained memory pressure, scale up your instance to provide additional memory for your workload.

Read preference options

Amazon DocumentDB uses a cloud-native shared storage service that replicates data six times across three Availability Zones to provide high levels of durability. Amazon DocumentDB does not rely on replicating data to multiple instances to achieve durability. Your cluster's data is durable whether it contains a single instance or 15 instances.

Topics

- Write durability
- Read isolation
- Read consistency
- High availability
- Scaling reads

Write durability

Amazon DocumentDB uses a unique, distributed, fault-tolerant, self-healing storage system. This system replicates six copies (V=6) of your data across three AWS Availability Zones to provide high availability and durability. When writing data, Amazon DocumentDB ensures that all writes are durably recorded on a majority of nodes before acknowledging the write to the client. If you are running a three-node MongoDB replica set, using a write concern of `{w:3, j:true}` would yield the best possible configuration when comparing with Amazon DocumentDB.

Writes to an Amazon DocumentDB cluster must be processed by the cluster's writer instance. Attempting to write to a reader results in an error. An acknowledged write from an Amazon DocumentDB primary instance is durable, and can't be rolled back. Amazon DocumentDB is highly durable by default and doesn't support a non-durable write option. You can't modify the durability level (that is, write concern). Amazon DocumentDB ignores w=anything and is effectively w: 3 and j: true. You cannot reduce it.

Because storage and compute are separated in the Amazon DocumentDB architecture, a cluster with a single instance is highly durable. Durability is handled at the storage layer. As a result, an Amazon DocumentDB cluster with a single instance and one with three instances achieve the same level of durability. You can configure your cluster to your specific use case while still providing high durability for your data.

Writes to an Amazon DocumentDB cluster are atomic within a single document.

Amazon DocumentDB does not support the `wtimeout` option and will not return an error if a value is specified. Writes to the primary Amazon DocumentDB instance are guaranteed not to block indefinitely.

Read isolation

Reads from an Amazon DocumentDB instance only return data that is durable before the query begins. Reads never return data modified after the query begins execution nor are dirty reads possible under any circumstances.

Read consistency

Data read from an Amazon DocumentDB cluster is durable and will not be rolled back. You can modify the read consistency for Amazon DocumentDB reads by specifying the read preference for the request or connection. Amazon DocumentDB does not support a non-durable read option.

Reads from an Amazon DocumentDB cluster's primary instance are strongly consistent under normal operating conditions and have read-after-write consistency. If a failover event occurs between the write and subsequent read, the system can briefly return a read that is not strongly consistent. All reads from a read replica are eventually consistent and return the data in the same order, and often with less than 100 ms replica lag.

Amazon DocumentDB read preferences

Amazon DocumentDB supports setting a read preference option only when reading data from the cluster endpoint in replica set mode. Setting a read preference option affects how your MongoDB client or driver routes read requests to instances in your Amazon DocumentDB cluster. You can set read preference options for a specific query, or as a general option in your MongoDB driver. (Consult your client or driver's documentation for instructions on how to set a read preference option.)

If your client or driver is not connecting to an Amazon DocumentDB cluster endpoint in replica set mode, the result of specifying a read preference is undefined.

Amazon DocumentDB does not support setting *tag sets* as a read preference.

Supported read preference options

- **`primary`**—Specifying a `primary` read preference helps ensure that all reads are routed to the cluster's primary instance. If the primary instance is unavailable, the read operation fails. A `primary` read preference yields read-after-write consistency and is appropriate for use cases that prioritize read-after-write consistency over high availability and read scaling.

  The following example specifies a `primary` read preference:

```
db.example.find().readPref('primary')
```

- **primaryPreferred**—Specifying a `primaryPreferred` read preference routes reads to the primary instance under normal operation. If there is a primary failover, the client routes requests to a replica. A `primaryPreferred` read preference yields read-after-write consistency during normal operation, and eventually consistent reads during a failover event. A `primaryPreferred` read preference is appropriate for use cases that prioritize read-after-write consistency over read scaling, but still require high availability.

  The following example specifies a `primaryPreferred` read preference:

  ```
  db.example.find().readPref('primaryPreferred')
  ```

- **secondary**—Specifying a `secondary` read preference ensures that reads are only routed to a replica, never the primary instance. If there are no replica instances in a cluster, the read request fails. A `secondary` read preference yields eventually consistent reads and is appropriate for use cases that prioritize primary instance write throughput over high availability and read-after-write consistency.

  The following example specifies a `secondary` read preference:

  ```
  db.example.find().readPref('secondary')
  ```

- **secondaryPreferred**—Specifying a `secondaryPreferred` read preference ensures that reads are routed to a read replica when one or more replicas are active. If there are no active replica instances in a cluster, the read request is routed to the primary instance. A `secondaryPreferred` read preference yields eventually consistent reads when the read is serviced by a read replica. It yields read-after-write consistency when the read is serviced by the primary instance (barring failover events). A `secondaryPreferred` read preference is appropriate for use cases that prioritize read scaling and high availability over read-after-write consistency.

  The following example specifies a `secondaryPreferred` read preference:

```
db.example.find().readPref('secondaryPreferred')
```

- **`nearest`**—Specifying a `nearest` read preference routes reads based solely on the measured latency between the client and all instances in the Amazon DocumentDB cluster. A `nearest` read preference yields eventually consistent reads when the read is serviced by a read replica. It yields read-after-write consistency when the read is serviced by the primary instance (barring failover events). A `nearest` read preference is appropriate for use cases that prioritize achieving the lowest possible read latency and high availability over read-after-write consistency and read scaling.

  The following example specifies a `nearest` read preference:

```
db.example.find().readPref('nearest')
```

High availability

Amazon DocumentDB supports highly available cluster configurations by using replicas as failover targets for the primary instance. If the primary instance fails, an Amazon DocumentDB replica is promoted as the new primary, with a brief interruption during which read and write requests made to the primary instance fail with an exception.

If your Amazon DocumentDB cluster doesn't include any replicas, the primary instance is re-created during a failure. Because promoting an Amazon DocumentDB replica is much faster than re-creating the primary instance, create one or more Amazon DocumentDB replicas as failover targets.

Replicas that are intended for use as failover targets should be of the same instance class as the primary instance. They should be provisioned in different Availability Zones from the primary. You can control which replicas are preferred as failover targets. For best practices on configuring Amazon DocumentDB for high availability, see Understanding Amazon DocumentDB cluster fault tolerance.

Scaling reads

Amazon DocumentDB replicas are ideal for read scaling. They are fully dedicated to read operations on your cluster volume, that is, replicas do not process writes. Data replication happens

within the cluster volume and not between instances. So each replica's resources are dedicated to processing your queries, not replicating and writing data.

If your application needs more read capacity, you can add a replica to your cluster quickly (usually in less than ten minutes). If your read capacity requirements diminish, you can remove unneeded replicas. With Amazon DocumentDB replicas, you pay only for the read capacity that you need.

Amazon DocumentDB supports client-side read scaling through the use of Read Preference options. For more information, see Amazon DocumentDB read preferences.

TTL deletes

Deletes from a TTL index area achieved via a background process are best effort and are not guaranteed within a specific timeframe. Factors like instance size, instance resource utilization, document size, and overall throughput can affect the timing of a TTL delete.

When the TTL monitor deletes your documents, each deletion incurs IO costs, which will increase your bill. If throughput and TTL delete rates increase, you should expect an increase in your bill due to increased IO usage.

When you create a TTL index on an existing collection, you must delete all expired documents before creating the index. The current TTL implementation is optimized for deleting a small fraction of documents in the collection, which is typical if TTL was enabled on the collection from the start, and may result in higher IOPS than necessary if a large number of documents need to be deleted at one go.

If you do not want to create a TTL index to delete documents, you can instead segment documents into collections based on time, and simply drop those collections when the documents are no longer needed. For example: you can create one collection per week and drop it without incurring IO costs. This can be significantly more cost effective than using a TTL index.

Billable resources

Identifying billable Amazon DocumentDB resources

As a fully managed database service, Amazon DocumentDB charges for instances, storage, I/Os, backups, and data transfer. For more information, see Amazon DocumentDB (with MongoDB compatibility) pricing.

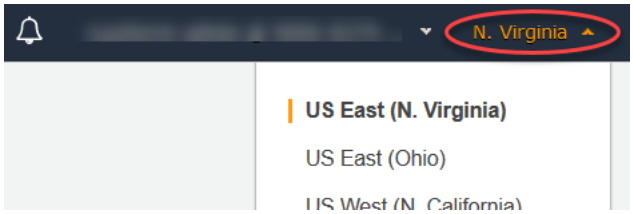
To discover billable resources in your account and potentially delete the resources, you can use the AWS Management Console or AWS CLI.

Using the AWS Management Console

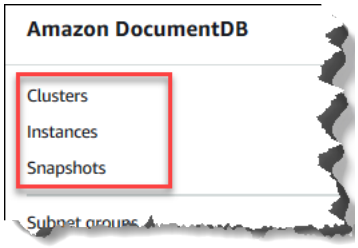
Using the AWS Management Console, you can discover the Amazon DocumentDB clusters, instances, and snapshots that you have provisioned for a given AWS Region.

To discover clusters, instances, and snapshots

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. To discover billable resources in a Region other than your default Region, in the upper-right corner of the screen, choose the AWS Region that you want to search.



3. In the navigation pane, choose the type of billable resource that you're interested in: **Clusters**, **Instances**, or **Snapshots**.



4. All your provisioned clusters, instances, or snapshots for the Region are listed in the right pane. You will be charged for clusters, instances, and snapshots.

Using the AWS CLI

Using the AWS CLI, you can discover the Amazon DocumentDB clusters, instances, and snapshots that you have provisioned for a given AWS Region.

To discover clusters and instances

The following code lists all your clusters and instances for the specified Region. If you want to search for clusters and instances in your default Region, you can omit the `--region` parameter.

Example

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
    --region us-east-1 \
    --query 'DBClusters[?Engine==`docdb`]' | \
       grep -e "DBClusterIdentifier" -e "DBInstanceIdentifier"
```

For Windows:

```
aws docdb describe-db-clusters ^
    --region us-east-1 ^
    --query 'DBClusters[?Engine==`docdb`]' | ^
       grep -e "DBClusterIdentifier" -e "DBInstanceIdentifier"
```

Output from this operation looks something like the following.

```
"DBClusterIdentifier": "docdb-2019-01-09-23-55-38",
        "DBInstanceIdentifier": "docdb-2019-01-09-23-55-38",
        "DBInstanceIdentifier": "docdb-2019-01-09-23-55-382",
"DBClusterIdentifier": "sample-cluster",
"DBClusterIdentifier": "sample-cluster2",
```

To discover snapshots

The following code lists all your snapshots for the specified Region. If you want to search for snapshots in your default Region, you can omit the `--region` parameter.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-snapshots \
  --region us-east-1 \
  --query 'DBClusterSnapshots[?Engine==`docdb`].
[DBClusterSnapshotIdentifier,SnapshotType]'
```

For Windows:

```
aws docdb describe-db-cluster-snapshots ^
  --region us-east-1 ^
```

```
  --query 'DBClusterSnapshots[?Engine==`docdb`].
[DBClusterSnapshotIdentifier,SnapshotType]'
```

Output from this operation looks something like the following.

```
[
    [
        "rds:docdb-2019-01-09-23-55-38-2019-02-13-00-06",
        "automated"
    ],
    [
        "test-snap",
        "manual"
    ]
]
```

You only need to delete `manual` snapshots. `Automated` snapshots are deleted when you delete the cluster.

Deleting unwanted billable resources

To delete a cluster, you must first delete all the instances in the cluster.

- To delete instances, see Deleting an Amazon DocumentDB instance.

> **Important**
>
> Even if you delete the instances in a cluster, you are still billed for the storage and backup usage associated with that cluster. To stop all charges, you must also delete your cluster and manual snapshots.

- To delete clusters, see Deleting an Amazon DocumentDB cluster.
- To delete manual snapshots, see Deleting a cluster snapshot.

What is a document database?

Some developers don't think of their data model in terms of normalized rows and columns. Typically, in the application tier, data is represented as a JSON document because it is more intuitive for developers to think of their data model as a document.

The popularity of document databases has grown because they let you persist data in a database by using the same document model format that you use in your application code. Document databases provide powerful and intuitive APIs for flexible and agile development.

Topics

- Document database use cases
- Understanding documents
- Working with documents

Document database use cases

Your use case drives whether you need a document database or some other type of database for managing your data. Document databases are useful for workloads that require a flexible schema for fast, iterative development. The following are some examples of use cases for which document databases can provide significant advantages:

Topics

- User profiles
- Real-time big data
- Content management

User profiles

Because document databases have a flexible schema, they can store documents that have different attributes and data values. Document databases are a practical solution to online profiles in which different users provide different types of information. Using a document database, you can store each user's profile efficiently by storing only the attributes that are specific to each user.

Suppose that a user elects to add or remove information from their profile. In this case, their document could be easily replaced with an updated version that contains any recently added attributes and data or omits any newly omitted attributes and data. Document databases easily manage this level of individuality and fluidity.

Real-time big data

Historically, the ability to extract information from operational data was hampered by the fact that operational databases and analytical databases were maintained in different environments—

operational and business/reporting respectively. Being able to extract operational information in real time is critical in a highly competitive business environment. By using document databases, a business can store and manage operational data from any source and concurrently feed the data to the BI engine of choice for analysis. There is no requirement to have two environments.

Content management

To effectively manage content, you must be able to collect and aggregate content from a variety of sources, and then deliver it to the customer. Due to their flexible schema, document databases are perfect for collecting and storing any type of data. You can use them to create and incorporate new types of content, including user-generated content, such as images, comments, and videos.

Understanding documents

Document databases are used for storing semistructured data as a document—rather than normalizing data across multiple tables, each with a unique and fixed structure, as in a relational database. Documents stored in a document database use nested key-value pairs to provide the document's structure or schema. However, different types of documents can be stored in the same document database, thus meeting the requirement for processing similar data that is in different formats. For example, because each document is self-describing, the JSON-encoded documents for an online store that are described in the topic Example documents in a document database can be stored in the same document database.

Topics

- SQL vs. non-relational terminology
- Simple documents
- Embedded documents
- Example documents in a document database
- Understanding normalization in a document database

SQL vs. non-relational terminology

The following table compares terminology used by document databases (MongoDB) with terminology used by SQL databases.

SQL	MongoDB
Table	Collection
Row	Document
Column	Field
Primary key	ObjectId
Index	Index
View	View
Nested table or object	Embedded document
Array	Array

Simple documents

All documents in a document database are self-describing. This documentation uses JSON-like formatted documents, although you can use other means of encoding.

A simple document has one or more fields that are all at the same level within the document. In the following example, the fields SSN, LName, FName, DOB, Street, City, State-Province, PostalCode, and Country are all siblings within the document.

```
{
   "SSN": "123-45-6789",
   "LName": "Rivera",
   "FName": "Martha",
   "DOB": "1992-11-16",
   "Street": "125 Main St.",
   "City": "Anytown",
   "State-Province": "WA",
   "PostalCode": "98117",
   "Country": "USA"
}
```

When information is organized in a simple document, each field is managed individually. To retrieve a person's address, you must retrieve `Street`, `City`, `State-Province`, `PostalCode`, and `Country` as individual data items.

Embedded documents

A complex document organizes its data by creating embedded documents within the document. Embedded documents help manage data in groupings and as individual data items, whichever is more efficient in a given case. Using the preceding example, you could embed an `Address` document in the main document. Doing this results in the following document structure:

```
{
   "SSN": "123-45-6789",
   "LName": "Rivera",
   "FName": "Martha",
   "DOB": "1992-11-16",
   "Address":
   {
       "Street": "125 Main St.",
       "City": "Anytown",
       "State-Province": "WA",
       "PostalCode": "98117",
       "Country": "USA"
   }
}
```

You can now access the data in the document as individual fields (`"SSN":`), as an embedded document (`"Address":`), or as a member of an embedded document (`"Address": {"Street":}`).

Example documents in a document database

As stated earlier, because each document in a document database is self-describing, the structure of documents within a document database can be different from one another. The following two documents, one for a book and another for a periodical, are different structurally. Yet both of them can be in the same document database.

The following is a sample book document:

```
{
```

```
    "_id" : "9876543210123",
    "Type": "book",
    "ISBN": "987-6-543-21012-3",
    "Author":
    {
        "LName":"Roe",
        "MI": "T",
        "FName": "Richard"
    },
    "Title": "Understanding Document Databases"
}
```

The following is a sample periodical document with two articles:

```
{
    "_id" : "0123456789012",
    "Publication": "Programming Today",
    "Issue":
    {
        "Volume": "14",
        "Number": "09"
    },
    "Articles" : [
        {
            "Title": "Is a Document Database Your Best Solution?",
            "Author":
            {
                "LName": "Major",
                "FName": "Mary"
            }
        },
        {
            "Title": "Databases for Online Solutions",
            "Author":
            {
                "LName": "Stiles",
                "FName": "John"
            }
        }
    ],
    "Type": "periodical"
}
```

Compare the structure of these two documents. With a relational database, you need either separate "periodical" and "books" tables, or a single table with unused fields, such as "Publication," "Issue," "Articles," and "MI," as `null` values. Because document databases are semistructured, with each document defining its own structure, these two documents can coexist in the same document database with no `null` fields. Document databases are good at dealing with sparse data.

Developing against a document database enables quick, iterative development. This is because you can change the data structure of a document dynamically, without having to change the schema for the entire collection. Document databases are well suited for agile development and dynamically changing environments.

Understanding normalization in a document database

Document databases are not normalized; data found in one document can be repeated in another document. Further, some data discrepancies can exist between documents. For example, consider the scenario in which you make a purchase at an online store and all the details of your purchases are stored in a single document. The document might look something like the following JSON document:

```
{
    "DateTime": "2018-08-15T12:13:10Z",
    "LName" : "Santos",
    "FName" : "Paul",
    "Cart" : [
        {
            "ItemId" : "9876543210123",
            "Description" : "Understanding Document Databases",
            "Price" : "29.95"
        },
        {
            "ItemId" : "0123456789012",
            "Description" : "Programming Today",
            "Issue": {
                "Volume": "14",
                "Number": "09"
            },
            "Price" : "8.95"
        },
        {
            "ItemId": "234567890-K",
            "Description": "Gel Pen (black)",
```

```
            "Price": "2.49"
        }
    ],
    "PaymentMethod" :
    {
        "Issuer" : "MasterCard",
        "Number" : "1234-5678-9012-3456"
    },
    "ShopperId" : "1234567890"
}
```

All this information is stored as a document in a transaction collection. Later, you realize that you forgot to purchase one item. So you again log on to the same store and make another purchase, which is also stored as another document in the transaction collection.

```
{
    "DateTime": "2018-08-15T14:49:00Z",
    "LName" : "Santos",
    "FName" : "Paul",
    "Cart" : [
        {
            "ItemId" : "2109876543210",
            "Description" : "Document Databases for Fun and Profit",
            "Price" : "45.95"
        }
    ],
    "PaymentMethod" :
    {
        "Issuer" : "Visa",
        "Number" : "0987-6543-2109-8765"
    },
    "ShopperId" : "1234567890"
}
```

Notice the redundancy between these two documents—your name and shopper ID (and, if you used the same credit card, your credit card information). But that's okay because storage is inexpensive, and each document completely records a single transaction that can be retrieved quickly with a simple key-value query that requires no joins.

There is also an apparent discrepancy between the two documents—your credit card information. This is only an apparent discrepancy because it is likely that you used a different credit card for each purchase. Each document is accurate for the transaction that it documents.

Working with documents

As a document database, Amazon DocumentDB makes it easy to store, query, and index JSON data. In Amazon DocumentDB, a collection is analogous to a table in a relational database, except there is no single schema enforced upon all documents. Collections let you group similar documents together while keeping them all in the same database, without requiring that they be identical in structure.

Using the example documents from earlier sections, it is likely that you'd have collections for `reading_material` and `office_supplies`. It is the responsibility of your software to enforce which collection a document belongs in.

The following examples use the MongoDB API to show how to add, query, update, and delete documents.

Topics

- Adding documents
- Querying documents
- Updating documents
- Deleting documents

Adding documents

In Amazon DocumentDB, a database is created when first you add a document to a collection. In this example, you are creating a collection named `example` in the `test` database, which is the default database when you connect to a cluster. Because the collection is implicitly created when the first document is inserted, there is no error checking of the collection name. Therefore, a typo in the collection name, such as `eexample` instead of `example`, will create and add the document to `eexample` collection rather than the intended collection. Error checking must be handled by your application.

The following examples use the MongoDB API to add documents.

Topics

- Adding a single document
- Adding multiple documents

Adding a single document

To add a single document to a collection, use the `insertOne( {} )` operation with the document that you want added to the collection.

```
db.example.insertOne(
    {
        "Item": "Ruler",
        "Colors": ["Red","Green","Blue","Clear","Yellow"],
        "Inventory": {
            "OnHand": 47,
            "MinOnHand": 40
        },
        "UnitPrice": 0.89
    }
)
```

Output from this operation looks something like the following (JSON format).

```
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5bedafbcf65ff161707de24f")
}
```

Adding multiple documents

To add multiple documents to a collection, use the `insertMany( [{},...,{}] )` operation with a list of the documents that you want added to the collection. Although the documents in this particular list have different schemas, they can all be added to the same collection.

```
db.example.insertMany(
    [
        {
            "Item": "Pen",
            "Colors": ["Red","Green","Blue","Black"],
            "Inventory": {
                "OnHand": 244,
                "MinOnHand": 72
            }
        },
        {
```

```
            "Item": "Poster Paint",
            "Colors": ["Red","Green","Blue","Black","White"],
            "Inventory": {
                "OnHand": 47,
                "MinOnHand": 50
            }
        },
        {
            "Item": "Spray Paint",
            "Colors": ["Black","Red","Green","Blue"],
            "Inventory": {
                "OnHand": 47,
                "MinOnHand": 50,
                "OrderQnty": 36
            }
        }
    ]
)
```

Output from this operation looks something like the following (JSON format).

```
{
    "acknowledged" : true,
    "insertedIds" : [
            ObjectId("5bedb07941ca8d9198f5934c"),
            ObjectId("5bedb07941ca8d9198f5934d"),
            ObjectId("5bedb07941ca8d9198f5934e")
    ]
}
```

Querying documents

At times, you might need to look up your online store's inventory so that customers can see and purchase what you're selling. Querying a collection is relatively easy, whether you want all documents in the collection or only those documents that satisfy a particular criterion.

To query for documents, use the `find()` operation. The `find()` command has a single document parameter that defines the criteria to use in choosing the documents to return. The output from `find()` is a document formatted as a single line of text with no line breaks. To format the output document for easier reading, use `find().pretty()`. All the examples in this topic use `.pretty()` to format the output.

Use the four documents you inserted into the `example` collection in the preceding two exercises — `insertOne()` and `insertMany()`.

Topics

- Retrieving all documents in a collection
- Retrieving documents that match a field value
- Retrieving documents that match an embedded document
- Retrieving documents that match a field value in an embedded document
- Retrieving documents that match an array
- Retrieving documents that match a value in an array
- Retrieving documents using operators

Retrieving all documents in a collection

To retrieve all the documents in your collection, use the `find()` operation with an empty query document.

The following query returns all documents in the `example` collection.

```
db.example.find( {} ).pretty()
```

Retrieving documents that match a field value

To retrieve all documents that match a field and value, use the `find()` operation with a query document that identifies the fields and values to match.

Using the preceding documents, this query returns all documents where the "Item" field equals "Pen".

```
db.example.find( { "Item": "Pen" } ).pretty()
```

Retrieving documents that match an embedded document

To find all the documents that match an embedded document, use the `find()` operation with a query document that specifies the embedded document name and all the fields and values for that embedded document.

When matching an embedded document, the document's embedded document must have the same name as in the query. In addition, the fields and values in the embedded document must match the query.

The following query returns only the "Poster Paint" document. This is because the "Pen" has different values for "OnHand" and "MinOnHand", and "Spray Paint" has one more field (`OrderQnty`) than the query document.

```
db.example.find({"Inventory": {
    "OnHand": 47,
    "MinOnHand": 50 } } ).pretty()
```

Retrieving documents that match a field value in an embedded document

To find all the documents that match an embedded document, use the `find()` operation with a query document that specifies the embedded document name and all the fields and values for that embedded document.

Given the preceding documents, the following query uses "dot notation" to specify the embedded document and fields of interest. Any document that matches these are returned, regardless of what other fields might be present in the embedded document. The query returns "Poster Paint" and "Spray Paint" because they both match the specified fields and values.

```
db.example.find({"Inventory.OnHand": 47, "Inventory.MinOnHand": 50 }).pretty()
```

Retrieving documents that match an array

To find all documents that match an array, use the `find()` operation with the array name that you are interested in and all the values in that array. The query returns all documents that have an array with that name in which the array values are identical to and in the same order as in the query.

The following query returns only the "Pen" because the "Poster Paint" has an additional color (White), and "Spray Paint" has the colors in a different order.

```
db.example.find( { "Colors": ["Red","Green","Blue","Black"] } ).pretty()
```

Retrieving documents that match a value in an array

To find all the documents that have a particular array value, use the `find()` operation with the array name and the value that you're interested in.

```
db.example.find( { "Colors": "Red" } ).pretty()
```

The preceding operation returns all three documents because each of them has an array named `Colors` and the value "`Red`" somewhere in the array. If you specify the value "`White`," the query would only return "Poster Paint."

Retrieving documents using operators

The following query returns all documents where the "`Inventory.OnHand`" value is less than 50.

```
db.example.find(
        { "Inventory.OnHand": { $lt: 50 } } )
```

For a listing of supported query operators, see Query and projection operators.

Updating documents

Typically, your documents are not static and are updated as part of your application workflows. The following examples show some of the ways that you can update documents.

To update an existing document, use the `update()` operation. The `update()` operation has two document parameters. The first document identifies which document or documents to update. The second document specifies the updates to make.

When you update an existing field — whether that field is a simple field, an array, or an embedded document — you specify the field name and its values. At the end of the operation, it is as though the field in the old document has been replaced by the new field and values.

Topics

- Updating the values of an existing field
- Adding a new field
- Replacing an embedded document
- Inserting new fields into an embedded document
- Removing a field from a document

- Removing a field from multiple documents

Updating the values of an existing field

Use the following four documents that you added earlier for the following updating operations.

```
{
    "Item": "Ruler",
    "Colors": ["Red","Green","Blue","Clear","Yellow"],
    "Inventory": {
        "OnHand": 47,
        "MinOnHand": 40
    },
    "UnitPrice": 0.89
},
{
    "Item": "Pen",
    "Colors": ["Red","Green","Blue","Black"],
    "Inventory": {
        "OnHand": 244,
        "MinOnHand": 72
    }
},
{
    "Item": "Poster Paint",
    "Colors": ["Red","Green","Blue","Black","White"],
    "Inventory": {
        "OnHand": 47,
        "MinOnHand": 50
    }
},
{
    "Item": "Spray Paint",
    "Colors": ["Black","Red","Green","Blue"],
    "Inventory": {
        "OnHand": 47,
        "MinOnHand": 50,
        "OrderQnty": 36
    }
}
```

To update a simple field

To update a simple field, use `update()` with `$set` to specify the field name and new value. The following example changes the `Item` from "Pen" to "Gel Pen".

```
db.example.update(
    { "Item" : "Pen" },
    { $set: { "Item": "Gel Pen" } }
)
```

Results from this operation look something like the following.

```
{
    "Item": "Gel Pen",
    "Colors": ["Red","Green","Blue","Black"],
    "Inventory": {
        "OnHand": 244,
        "MinOnHand": 72
    }
}
```

To update an array

The following example replaces the existing array of colors with a new array that includes `Orange` and drops `White` from the list of colors. The new list of colors is in the order specified in the `update()` operation.

```
db.example.update(
    { "Item" : "Poster Paint" },
    { $set: { "Colors": ["Red","Green","Blue","Orange","Black"] } }
)
```

Results from this operation look something like the following.

```
{
    "Item": "Poster Paint",
    "Colors": ["Red","Green","Blue","Orange","Black"],
    "Inventory": {
        "OnHand": 47,
        "MinOnHand": 50
    }
}
```

Adding a new field

To modify a document by adding one or more new fields, use the `update()` operation with a query document that identifies the document to insert into and the new fields and values to insert using the `$set` operator.

The following example adds the field `UnitPrice` with the value `3.99` to the Spray Paints document. Note that the value `3.99` is numeric and not a string.

```
db.example.update(
    { "Item": "Spray Paint" },
    { $set: { "UnitPrice": 3.99 } }
)
```

Results from this operation look something like the following (JSON format).

```
{
    "Item": "Spray Paint",
    "Colors": ["Black","Red","Green","Blue"],
    "Inventory": {
        "OnHand": 47,
        "MinOnHand": 50,
        "OrderQnty": 36
    },
    "UnitPrice": 3.99
}
```

Replacing an embedded document

To modify a document by replacing an embedded document, use the `update()` operation with documents that identify the embedded document and its new fields and values using the `$set` operator.

Given the following document.

```
db.example.insert({
    "DocName": "Document 1",
    "Date": {
        "Year": 1987,
        "Month": 4,
        "Day": 18
```

```
    }
})
```

To replace an embedded document

The following example replaces the current Date document with a new one that has only the fields `Month` and `Day`; `Year` has been eliminated.

```
db.example.update(
    { "DocName" : "Document 1" },
    { $set: { "Date": { "Month": 4, "Day": 18 } } }
)
```

Results from this operation look something like the following.

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18
    }
}
```

Inserting new fields into an embedded document

To add fields to an embedded document

To modify a document by adding one or more new fields to an embedded document, use the `update()` operation with documents that identify the embedded document and "dot notation" to specify the embedded document and the new fields and values to insert using the `$set` operator.

Given the following document, the following code uses "dot notation" to insert the `Year` and `DoW` fields to the embedded `Date` document, and `Words` into the parent document.

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18
    }
}
```

```
db.example.update(
    { "DocName" : "Document 1" },
    { $set: { "Date.Year": 1987,
              "Date.DoW": "Saturday",
              "Words": 2482 } }
)
```

Results from this operation look something like the following.

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
        "DoW": "Saturday"
    },
    "Words": 2482
}
```

Removing a field from a document

To modify a document by removing a field from the document, use the `update()` operation with a query document that identifies the document to remove the field from, and the `$unset` operator to specify the field to remove.

The following example removes the `Words` field from the preceding document.

```
db.example.update(
    { "DocName" : "Document 1" },
    { $unset: { Words:1 } }
)
```

Results from this operation look something like the following.

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
```

```
        "DoW": "Saturday"
    }
}
```

Removing a field from multiple documents

To modify a document by removing a field from multiple documents, use the `update()` operation with the `$unset` operator and the `multi` option set to `true`.

The following example removes the `Inventory` field from all documents in the example collection. If a document does not have the `Inventory` field, no action is taken on that document. If `multi: true` is omitted, the action is performed only on the first document that meets the criterion.

```
db.example.update(
    {},
    { $unset: { Inventory:1 } },
    { multi: true }
)
```

Deleting documents

To remove a document from your database, use the `remove()` operation, specifying which document to remove. The following code removes "Gel Pen" from your `example` collection.

```
db.example.remove( { "Item": "Gel Pen" } )
```

To remove all documents from your database, use the `remove()` operation with an empty query.

```
db.example.remove( { } )
```

Get started with Amazon DocumentDB

There are many ways to connect and get started with Amazon DocumentDB. This guide is the quickest, simplest, and easiest way for users to get started using our powerful document database. This guide uses AWS CloudShell to connect and query your Amazon DocumentDB cluster directly from the AWS Management Console. New customers who are eligible for the AWS Free Tier can use Amazon DocumentDB and CloudShell for free. If your AWS CloudShell environment or Amazon DocumentDB cluster makes use of resources beyond the free tier, you are charged the normal AWS rates for those resources. This guide will get you started with Amazon DocumentDB in less than five minutes.

Note

The instructions in this guide are specifically for creating and connecting to Amazon DocumentDB instance-based clusters where Amazon DocumentDB and AWS CloudShell are available.

- If you want to create and connect to Amazon DocumentDB elastic clusters, see Get started with Amazon DocumentDB elastic clusters.
- If you are located in AWS China Regions, see Connect Amazon EC2 automatically.

Topics

- Prerequisites
- Step 1: Create a cluster
- Step 2: Connect to your cluster
- Step 3: Insert and query data
- Step 4: Explore
- Amazon DocumentDB quick start using CloudFormation

Prerequisites

Before you create your first Amazon DocumentDB cluster, you must do the following:

Create an Amazon Web Services (AWS) account

Before you can begin using Amazon DocumentDB, you must have an Amazon Web Services (AWS) account. The AWS account is free. You pay only for the services and resources that you use.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://signin.aws.amazon.com/signup?request_type=register.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Set up the needed AWS Identity and Access Management (IAM) permissions.

Access to manage Amazon DocumentDB resources such as clusters, instances, and cluster parameter groups requires credentials that AWS can use to authenticate your requests. For more information, see Identity and Access Management for Amazon DocumentDB.

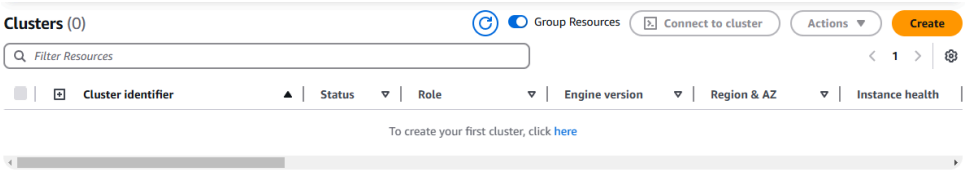
1. In the search bar of the AWS Management Console, enter **`IAM`** and select **IAM** in the drop down menu that appears.
2. Once you're in the IAM console, select **Users** from the navigation pane.
3. Select your username.
4. Choose **Add permissions**.
5. Select **Attach policies directly**.
6. Type `AmazonDocDBFullAccess` in the search bar and select it once it appears in the search results.
7. Choose **Next**.
8. Choose **Add permissions**.

> **Note**
>
> Your AWS account includes a default VPC in each Region. If you choose to use an Amazon VPC, complete the steps in the Create an Amazon VPC topic in the *Amazon VPC User Guide*.

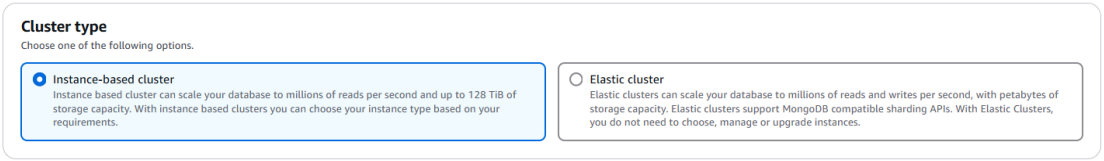
Step 1: Create a cluster

In this step you will create an Amazon DocumentDB cluster.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. On the Amazon DocumentDB management console, under **Clusters**, choose **Create**.



3. On the Create Amazon DocumentDB cluster page, in the **Cluster type** section, choose **Instance-based cluster** (this is the default option).

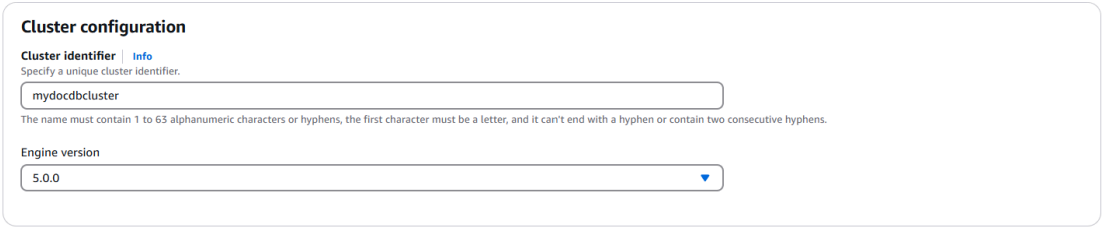


> **Note**
>
> The other option in this category is **Elastic cluster**. To learn more about Amazon DocumentDB elastic clusters, see Using Amazon DocumentDB Elastic Clusters

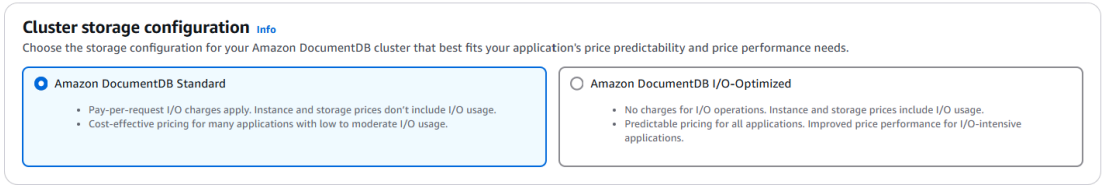
4. In the **Cluster configuration** section:

   a. For **Cluster identifier**, enter a unique name, such as **`mydocdbcluster`**. Note that the console will change all cluster names into lower-case regardless of how they are entered.

   b. For **Engine version**, choose **5.0.0**.



5. In the **Cluster storage configuration** section, choose **Amazon DocumentDB Standard** (this is the default option).



> **Note**
>
> The other option in this category is **Amazon DocumentDB I/O-Optimized**. To learn more about either option, see Amazon DocumentDB cluster storage configurations

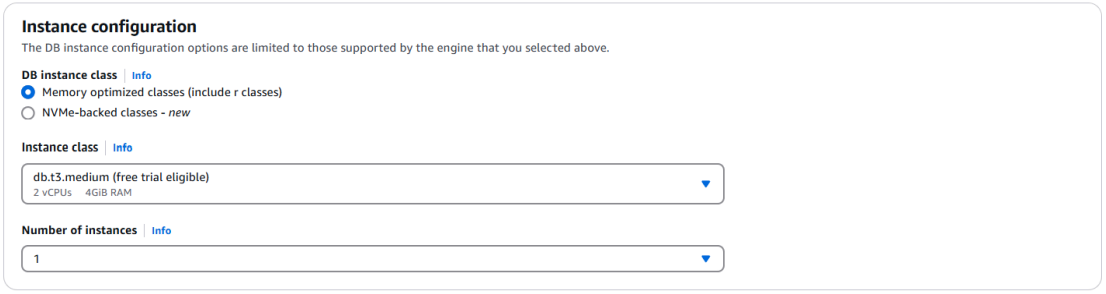
6. In the **Instance configuration** section:

   a. For **DB instance class**, choose **Memory optimized classes (include r classes)** (this is default).

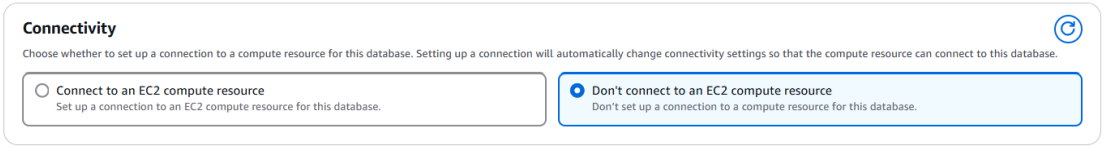
      The other instance option is **NVMe-backed classes**. To learn more, see NVMe-backed instances.

   b. For **Instance class**, choose **db.t3.medium**. This is eligible for the AWS free trial.

   c. For **Number of instances**, choose **1** instance. Choosing one instance helps minimize costs. If this were a production system, we would recommend that you provision three instances for high availability.

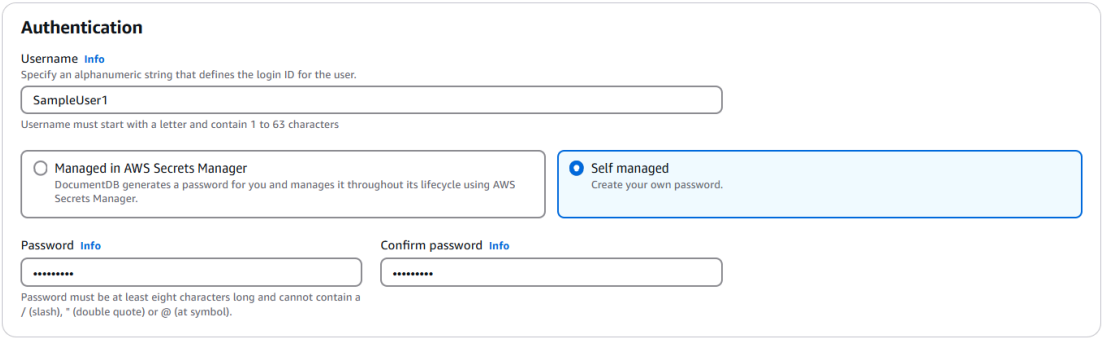


7. In the **Connectivity** section, leave the default setting of **Don't connect to an EC2 compute resource**.



8. In the **Authentication** section, enter a username for the primary user, and then choose **Self managed**. Enter a password, then confirm it.

   If you instead chose **Managed in AWS Secrets Manager**, see [Password management with Amazon DocumentDB and AWS Secrets Manager] for more information.



9. Leave all other options as default and choose **Create cluster**.

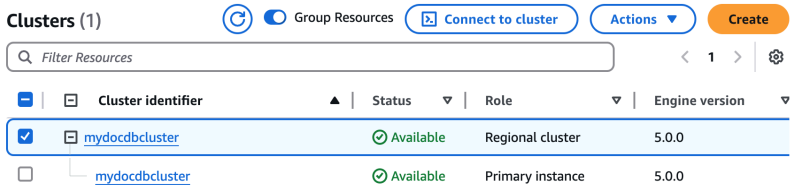
Amazon DocumentDB is now provisioning your cluster, which can take up to a few minutes to finish.

> **Note**
>
> For information about cluster status values, see [Cluster status values] in the Monitoring Amazon DocumentDB chapter.

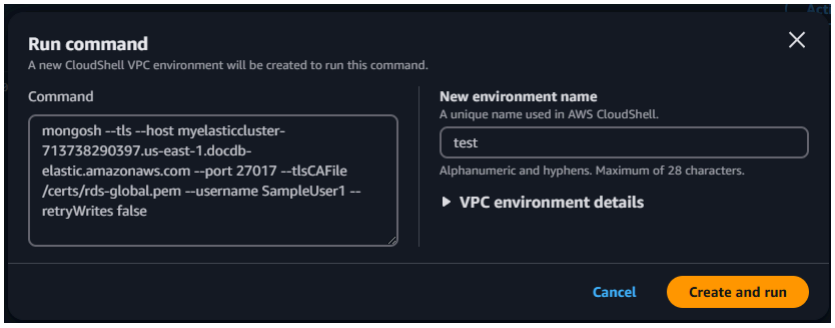
Step 2: Connect to your cluster

Connect to your Amazon DocumentDB cluster using AWS CloudShell.

1. On the Amazon DocumentDB management console, under **Clusters**, locate the cluster you created. Choose your cluster by clicking the check box next to it.



2. Choose **Connect to cluster** (which is next to the **Actions** dropdown menu). This button is enabled only after you have clicked the checkbox next to your cluster, and the status of both the regional cluster and primary instance(s) show as **Available**. The CloudShell **Run command** screen appears.
3. In the **New environment name** field, enter a unique name, such as "test" and choose **Create and run**. VPC environment details are automatically configured for your Amazon DocumentDB database.



4. When prompted, enter the password you created in Step 1: Create an Amazon DocumentDB cluster (sub-step 7).



   After you enter your password and your prompt becomes `rs0 [direct: primary] <env-name>>`, you are successfully connected to your Amazon DocumentDB cluster.

> **Note**
>
> For information about troubleshooting, see Troubleshooting Amazon DocumentDB.

Step 3: Insert and query data

Now that you are connected to your cluster, you can run a few queries to get familiar with using a document database.

1. To insert a single document, enter the following:

```
db.collection.insertOne({"hello":"DocumentDB"})
```

   You get the following output:

```
{
  acknowledged: true,
  insertedId: ObjectId('673657216bdf6258466b128c')
}
```

2. You can read the document that you wrote with the `findOne()` command (because it only returns a single document). Input the following:

```
db.collection.findOne()
```

   You get the following output:

```
{ "_id" : ObjectId("5e401fe56056fda7321fbd67"), "hello" : "DocumentDB" }
```

3. To perform a few more queries, consider a gaming profiles use case. First, insert a few entries into a collection titled `profiles`. Input the following:

```
db.profiles.insertMany([{ _id: 1, name: 'Matt', status: 'active', level: 12, score:
 202 },
      { _id: 2, name: 'Frank', status: 'inactive', level: 2, score: 9 },
      { _id: 3, name: 'Karen', status: 'active', level: 7, score: 87 },
      { _id: 4, name: 'Katie', status: 'active', level: 3, score: 27 }
])
```

You get the following output:

```
{ acknowledged: true, insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4 } }
```

4. Use the `find()` command to return all the documents in the profiles collection. Input the following:

```
db.profiles.find()
```

You will get an output that will match the data you typed in Step 3.

5. Use a query for a single document using a filter. Input the following:

```
db.profiles.find({name: "Katie"})
```

You get the following output:

```
{ "_id" : 4, "name" : "Katie", "status": "active", "level": 3, "score":27}
```

6. Now let's try to find a profile and modify it using the `findAndModify` command. We'll give the user Matt an extra 10 points with the following code:

```
db.profiles.findAndModify({
        query: { name: "Matt", status: "active"},
        update: { $inc: { score: 10 } }
    })
```

You get the following output (note that his score hasn't increased yet):

```
{
    [{_id : 1, name : 'Matt', status: 'active', level: 12, score: 202}]
```

7. You can verify that his score has changed with the following query:

```
db.profiles.find({name: "Matt"})
```

You get the following output:

```
{ "_id" : 1, "name" : "Matt", "status" : "active", "level" : 12, "score" : 212 }
```

Step 4: Explore

Congratulations! You have successfully completed the Get started guide for Amazon DocumentDB instance-based clusters.

What's next? Learn how to fully leverage this database with some of its popular features:

- Managing Amazon DocumentDB
- Scaling
- Backing up and restoring

> **Note**
>
> The cluster you created from this get started exercise will continue to accrue costs unless you delete it. For directions, see Deleting an Amazon DocumentDB Cluster.

Amazon DocumentDB quick start using CloudFormation

This section contains steps and other information to help you get started quickly with Amazon DocumentDB (with MongoDB compatibility) using CloudFormation. For general information about Amazon DocumentDB, see What is Amazon DocumentDB (with MongoDB compatibility).

These instructions use an AWS CloudFormation template to create a cluster and instances in your default Amazon VPC. For instructions on creating these resources yourself, see Get started with Amazon DocumentDB.

> **Important**
>
> The CloudFormation stack that is created by this template creates multiple resources, including resources in Amazon DocumentDB (for example, a cluster and instances) and Amazon Elastic Compute Cloud (for example, a subnet group).
> Some of these resources are not free-tier resources. For pricing information, see Amazon DocumentDB Pricing and Amazon EC2 Pricing. You can delete the stack when you are finished with it to stop any charges.

This CloudFormation stack is intended for tutorial purposes only. If you use this template for a production environment, use stricter IAM policies and security. For information about securing resources, see Amazon VPC Security and Amazon EC2 Network and Security.

Topics

- Prerequisites
- Launching an Amazon DocumentDB CloudFormation stack
- Accessing the Amazon DocumentDB cluster
- Termination protection and deletion protection

Prerequisites

Before you create an Amazon DocumentDB cluster, you must have the following:

- A default Amazon VPC
- The required IAM permissions

Required IAM Permissions

The following permissions allow you to create resources for the CloudFormation stack:

AWS Managed Policies

- `AWSCloudFormationReadOnlyAccess`
- `AmazonDocDBFullAccess`

Additional IAM Permissions

The following policy outlines the additional permissions that are required to create and delete this CloudFormation stack.

In the following examples, replace each *`user input placeholder`* with your resource's information.

Amazon EC2 Key Pair

You must have a key pair (and the PEM file) available in the Region where you will create the CloudFormation stack. If you need to create a key pair, see Creating a Key Pair Using Amazon EC2 in the *Amazon EC2 User Guide*.

Launching an Amazon DocumentDB CloudFormation stack

This section describes how to launch and configure an Amazon DocumentDB CloudFormation stack.

1. Sign in to the AWS Management Console at https://console.aws.amazon.com/.
2. The following table lists the Amazon DocumentDB stack templates for each AWS Region. Choose **Launch Stack** for the AWS Region you want to launch your stack in.

Region	View Template	View in Designer	Launch
US East (Ohio)	View Template	View in Designer	
US East (N. Virginia)	View Template	View in Designer	
US West (Oregon)	View Template	View in Designer	
Asia Pacific (Mumbai)	View Template	View in Designer	
Asia Pacific (Seoul)	View Template	View in Designer	
Asia Pacific (Singapore)	View Template	View in Designer	
Asia Pacific (Sydney)	View Template	View in Designer	
Asia Pacific (Tokyo)	View Template	View in Designer	

Region	View Template	View in Designer	Launch
Canada (Central)	View Template	View in Designer	Launch Stack
Europe (Frankfurt)	View Template	View in Designer	Launch Stack
Europe (Ireland)	View Template	View in Designer	Launch Stack
Europe (London)	View Template	View in Designer	Launch Stack
Europe (Paris)	View Template	View in Designer	Launch Stack

3. **Create stack** — Describes the Amazon DocumentDB template that you selected. Every stack is based on a template — a JSON or YAML file — that contains configuration about the AWS resources you want to include in the stack. Because you chose to launch a stack from the provided templates above, your template has already been configured to create an Amazon DocumentDB stack for the AWS Region you chose.

   When you launch an CloudFormation stack, deletion protection for your Amazon DocumentDB cluster is disabled by default. If you want to enable deletion protection for your cluster, complete the following steps. Otherwise, choose **Next** to continue to the next step.

   To enable deletion protection for your Amazon DocumentDB cluster:

   1. Choose **View in Designer** from the bottom right corner of the **Create stack** page.
   2. Modify the template using the integrated JSON and YAML editor in the resulting CloudFormation Designer page of the console. Scroll to the `Resources` section and modify it to include `DeletionProtection`, as follows. For more information about using CloudFormation Designer, see What Is CloudFormation Designer?.

      JSON:

```
"Resources": {
    "DBCluster": {
        "Type": "AWS::DocDB::DBCluster",
        "DeletionPolicy": "Delete",
        "Properties": {
```

```
            "DBClusterIdentifier": {
                "Ref": "DBClusterName"
            },
            "MasterUsername": {
                "Ref": "MasterUser"
            },
            "MasterUserPassword": {
                "Ref": "MasterPassword"
            },
            "DeletionProtection": "true"
        }
    },
```

YAML:

```
Resources:
  DBCluster:
    Type: 'AWS::DocDB::DBCluster'
    DeletionPolicy: Delete
    Properties:
      DBClusterIdentifier: !Ref DBClusterName
      MasterUsername: !Ref MasterUser
      MasterUserPassword: !Ref MasterPassword
      DeletionProtection: 'true'
```

3. Choose **Create Stack** (

)

   from the top left corner of the page to save your changes and create a stack with these changes enabled.

4. After you save your changes, you will be redirected to the **Create stack** page.
5. Choose **Next** to continue.

4. **Specify stack details** — Enter the stack name and parameters for your template. Parameters are defined in your template and allow you to input custom values when you create or update a stack.

   - Under **Stack name**, enter a name for your stack or accept the provided name. The stack name can include letters (A—Z and a—z), numbers (0—9), and dashes (—).
   - Under **Parameters**, enter the following details:

- **DBClusterName** — Enter a name for your Amazon DocumentDB cluster or accept the provided name.

  Cluster naming constraints:

  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all clusters across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.
- **DBInstanceClass** — From the drop-down list, select the instance class for your Amazon DocumentDB cluster.
- **DBInstanceName** — Enter a name for your Amazon DocumentDB instance or accept the provided name.

  Instance naming constraints:

  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all instances across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.
- **MasterPassword** — The database admin account password.
- **MasterUser** — The database admin account username. The MasterUser must begin with a letter and can contain only alphanumeric characters.

Choose **Next** to save your changes and continue.

5. **Configure stack options** — Configure your stack's tags, permissions, and additional options.
   - **Tags** — Specify tags (key-value) pairs to apply to your resources in your stack. You can add up to 50 unique tags for each stack.
   - **Permissions** — Optional. Choose an IAM role to explicitly define how CloudFormation can create, modify, or delete resources in the stack. If you don't choose a role, CloudFormation uses permissions based on your user credentials. Before you specify a service role, ensure that you have permission to pass it (`iam:PassRole`). The `iam:PassRole` permission specifies which roles you can use.

> **Note**
>
> When you specify a service role, CloudFormation always uses that role for all operations that are performed on that stack. Other users that have permissions to perform operations on this stack will be able to use this role, even if they don't have permission to pass it. If the role includes permissions that the user shouldn't have, you can unintentionally escalate a user's permissions. Ensure that the role grants least privilege.

- **Advanced options** — You can set the following advanced options:
  - **Stack policy** — Optional. Defines the resources that you want to protect from unintentional updates during a stack update. By default, all resources can be updated during a stack update.

    You can enter the stack policy directly as JSON, or upload a JSON file containing the stack policy. For more information, see Prevent Updates to Stack Resources.
  - **Rollback configuration** — Optional. Specify CloudWatch Logs alarms for CloudFormation to monitor when creating and updating the stack. If the operation breaches an alarm threshold, CloudFormation rolls it back.
  - **Notification options** — Optional. Specify topics for Simple Notification System (SNS).
  - **Stack creation options** — Optional. You can specify the following options:
    - **Rollback on failure** — Whether or not the stack should be rolled back if the stack creation fails.
    - **Timeout** —The number of minutes before a stack creation times out.
    - **Termination protection** — Prevents the stack from being accidentally deleted.

      > **Note**
      >
      > CloudFormation termination protection is different from the Amazon DocumentDB concept of deletion protection. For more information, see Termination protection and deletion protection.

Choose **Next** to continue.

6. **Review <stack-name>** — Review your stack template, details, and configuration options. You can also open a **quick-create link** at the bottom of the page to create stacks with the same basic configurations as this one.

   - Choose **Create** to create the stack.
   - Alternatively, you can choose **Create change set**. A change set is a preview of how this stack will be configured before creating the stack. This allows you to examine various configurations before executing the change set.

Accessing the Amazon DocumentDB cluster

Once the CloudFormation stack has been completed, you can use an Amazon EC2 instance to connect to your Amazon DocumentDB cluster. For information about connecting to an Amazon EC2 instance using SSH, see Connect to Your Linux Instance in the *Amazon EC2 User Guide*.

After you are connected, see the following sections, which contain information about using Amazon DocumentDB.

- Step 3: Insert and query data
- Managing Amazon DocumentDB clusters and instances
- Monitoring and logging in Amazon DocumentDB

Termination protection and deletion protection

It is an Amazon DocumentDB best practice to enable deletion protection and termination protection. CloudFormation termination protection is a distinctly different feature from the Amazon DocumentDB deletion protection feature.

- **Termination protection** — You can prevent a stack from being accidentally deleted by enabling termination protection for your CloudFormation stack. If a user attempts to delete a stack with termination protection enabled on it, the deletion fails and the stack remains unchanged. Termination protection is disabled by default when you create a stack using CloudFormation. You can enable termination protection on a stack when you create it. For more information, see Setting CloudFormation Stack Options.
- **Deletion protection** — Amazon DocumentDB also provides the ability to enable deletion protection for a cluster. If a user attempts to delete an Amazon DocumentDB cluster with deletion protection enabled on it, the deletion fails and the cluster remains unchanged. Deletion

protection, when enabled, safeguards against accidental deletes from the Amazon DocumentDB AWS Management Console, AWS CLI, and CloudFormation. For more information on enabling and disabling deletion protection for an Amazon DocumentDB cluster, see Deletion protection.

Migrating and upgrading Amazon DocumentDB

Amazon DocumentDB is a fully managed database service that is compatible with the MongoDB API. You can migrate your data to Amazon DocumentDB from MongoDB databases running on premises or on Amazon Elastic Compute Cloud (Amazon EC2) using the process detailed in this section.

Topics

- Migrate to Amazon DocumentDB using AWS Database Migration Service (DMS): quick start guide
- Amazon DocumentDB migration runbook
- Migration from Couchbase Server
- Amazon DocumentDB engine version support dates
- Amazon DocumentDB in-place major version upgrade
- Amazon DocumentDB minor version upgrade
- Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service
- Using a long-term support (LTS) release
- Amazon DocumentDB Extended Support

Migrate to Amazon DocumentDB using AWS Database Migration Service (DMS): quick start guide

Topics

- Prepare the DMS source
- Setup DMS
- Enable DocumentDB compression
- Create a replication task
- Monitor progress
- Additional information

Prepare the DMS source

See Enabling change streams to enable DocumentDB change streams or enable the MongoDB Oplog to support DMS Change Data Capture (CDC).

- The DMS source must retain all ongoing changes until DMS full load completes for all included collections.
- DocumentDB change streams are time based. Make sure your `change_stream_log_retention_duration` setting is large enough to cover the time to complete the full load.
- The MongoDB Oplog is a fixed size. Make sure it is sized to hold all operations during full load.

Setup DMS

Create DMS instance, source, and target endpoints and test each endpoint.

Enable DocumentDB compression

Amazon DocumentDB 5.0: Enable compression by attaching a custom parameter group to your Amazon DocumentDB cluster and setting the `default_collection_compression` parameter to `enabled`. See Managing collection-level document compression for more information.

Amazon DocumentDB 8.0: Zstd compression is enabled by default. See Managing dictionary-based compression in Amazon DocumentDB 8.0 for more information.

Create a replication task

1. In the DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Tasks**.
2. Choose **Create task**.
3. On the **Create task** page, in the **Task configuration** section:
   - Enter a unique and meaningful **Task identifier** (for example, "mongodb-docdb-replication").
   - Choose the source endpoint you created previously in the **Source database endpoint** drop-down menu.
   - Choose the target endpoint you created previously in the **Target database endpoint** drop-down menu.

- For **Task type**, choose **Migrate and replicate**.

4. In the **Settings** section:

- For **Task logs**, check the **Turn on CloudWatch** logs box.
- For **Editing mode** (at the top of the section), choose **JSON editor** and set the following attributes:
  - Set `ParallelApplyThreads` to 5 (under `TargetMetadata`). This enables ~1000 insert/update/delete ops per second in CDC.
  - Set `MaxFullLoadSubTasks` to 16 (under `FullLoadSettings`). Consider increasing this depending on your instance size.
  - For large collections (over 100 GB), enable auto-partition (under Table Mapping and under the `parallel-load` attribute):
    - "type": "partitions-auto"
    - "number-of-partitions": 16

Monitor progress

Use the AWS DMS console or create a custom dashboard (dashboarder tool) to track migration. Focus on the following metrics:

- **FullLoadThroughputBandwidthTarget** — Measures the network bandwidth (in KB/second) used by DMS when transferring data to the target database during the full load phase of migration.
- **CDCLatencyTarget** — Measures the time delay (in seconds) bewteen when a change occurs in the source database and when that change is applied to the target database.
- **CDCThroughputRowsTarget** — Measures the number of rows per second that DMS is applying to the target database during the ongoing replication phase of migration.

Additional information

For more information about Amazon DocumentDB and AWS DMS, see:

- Amazon DocumentDB migration runbook
- Migrating from MongoDB to Amazon DocumentDB

Amazon DocumentDB migration runbook

This runbook provides a comprehensive guide for migrating a MongoDB database to Amazon DocumentDB using AWS Database Migration Service (DMS). It is designed to support database administrators, cloud engineers, and developers throughout the end-to-end migration journey—from initial discovery to post-migration validation.

Given the differences in implementation and supported features between MongoDB and Amazon DocumentDB, this runbook emphasizes a structured and systematic approach. It outlines essential pre-migration assessments, highlights compatibility considerations, and details the key tasks required to ensure a successful migration with minimal disruption.

The runbook is organized into the following topics:

- **Compatibility** — Understand the supported MongoDB features and data types in Amazon DocumentDB, and identify potential incompatibilities.
- **Workload discovery** — Analyze existing MongoDB workloads, including read/write patterns, data volumes, and performance baselines.
- **Index migration** — Analyze strategies for extracting and transforming MongoDB indexes for optimal performance in Amazon DocumentDB.
- **User migration** — Detail the approach for migrating database users, roles, and access controls to Amazon DocumentDB.
- **Data migration** — Cover various methods for data migration using AWS DMS, including full load and change data capture (CDC).
- **Monitoring** — Detail various monitoring approaches when migrating using DMS or native tools.
- **Validation** — Provide procedures for data integrity checks, functional validation, and performance comparison post-migration.

By following the guidance in this runbook, teams can ensure a smooth, secure, and efficient transition to Amazon DocumentDB, while preserving application functionality and minimizing risk.

Compatibility

Topics

- Core feature compatibility
- Amazon DocumentDB compatibility assessment tool

When migrating from MongoDB to Amazon DocumentDB, a thorough initial assessment and feature compatibility check is essential for a successful migration. This process begins with a comprehensive inventory of your MongoDB features, including aggregation pipeline operators, query patterns, indexes, and data models.

Since Amazon DocumentDB is compatible with MongoDB 3.6, 4.0, 5.0, and 8.0 API's, applications using newer MongoDB-specific features may require refactoring. Critical areas to evaluate include sharding mechanisms(Amazon DocumentDB uses a different approach), transaction implementations, change streams functionality, and index types (particularly sparse and partial indexes).

Performance characteristics also differ, with Amazon DocumentDB optimized for enterprise workloads with predictable performance. Testing should involve running representative workloads against both systems to identify query patterns that might need optimization.

Monitoring execution plans to detect potential performance gaps is important during the assessment phase. This helps create a clear migration roadmap, identifying necessary application changes and establishing realistic timelines for a smooth transition.

Core feature compatibility

Comprehensive feature support

- **CRUD operations** — Enjoy full support for all basic create, read, update, and delete operations, including bulk and query operators - providing seamless application compatibility.
- **Rich indexing capabilities** — Leverage comprehensive support for single field, compound, TTL, partial, sparse, and 2dsphere indexes, to optimize your query performance and text indexes (version 5) for text-based lookups.
- **Enterprise-grade replication** — Benefit from a robust automatic failover mechanism with read replicas for superior high availability without operational overhead.
- **Advanced backup solutions** — Rest easy with automated backup system featuring Point-in-Time Recovery (PITR) and on-demand manual snapshots for data protection.

Enhanced AWS-integrated features

- **Streamlined aggregation** — Take advantage of the most commonly used aggregation stages (`$match`, `$group`, `$sort`, `$project`, etc.) with optimized performance for enterprise workloads.

- **Transaction support** — Implement multi-document and multi-collection transactions, perfect for most business application needs.
- **Real-time data tracking** — Enable change streams by a simple command and increase change stream retention period through a simple parameter group setting for real-time data change monitoring.
- **Location-based services** — Implement geospatial applications with support for `$geoNear` operator and 2dsphere indexes.
- **Text search capabilities** — Utilize built-in text search functionality for content discovery needs.

Modern architecture advantages

- **Cloud-native design** — Enjoy AWS-optimized architecture that replaces legacy features like MapReduce with more efficient aggregation pipeline operations.
- **Enhanced security** — Benefit from AWS Identity and Access Management (IAM), SCRAM-SHA-1, SCRAM-SHA-256, X.509 certificate authentication, and password-based authentication.
- **Predictable performance** — Experience consistent performance optimized specifically for enterprise workloads.

For a comprehensive overview of Amazon DocumentDB's capabilities, refer to the Supported MongoDB APIs, operations, and data types in Amazon DocumentDB and Functional differences: Amazon DocumentDB and MongoDB to maximize your database's potential.

Amazon DocumentDB does not support all the indexes offered by MongoDB. We provide a free index tool on GitHub to check the compatibility. Run the index tool to assess incompatibility and plan workarounds accordingly.

Amazon DocumentDB compatibility assessment tool

The Amazon DocumentDB Compatibility Tool on GitHub analyzes your MongoDB workload and reports operators, commands, and features that are unsupported or require changes in Amazon DocumentDB. It supports target versions from 3.6 through 8.0.1 (default: 8.0.1).

Recommended: URI mode (MongoDB 5.0+)

URI mode connects directly to your MongoDB instance, samples operations in real time, and requires no profiling or log access.

Alternative methods

- **Log-based analysis** — Parses MongoDB profiler logs. Captures actual runtime behavior but requires profiling to be enabled and only covers the logging period.
- **Source code analysis** — Scans application source code for MongoDB API usage. Provides comprehensive coverage but may flag unused code paths and cannot detect dynamically constructed queries.

Prerequisites

- Python 3.7+
- Clone the tool: `git clone https://github.com/awslabs/amazon-documentdb-tools.git`
- Install dependencies: `pip install -r compat-tool/requirements.txt`

For full usage details and all available options, see the README on GitHub.

Workload discovery

Migrating from MongoDB to Amazon DocumentDB requires a thorough understanding of the existing database workload. Workload discovery is the process of analyzing your database usage patterns, data structures, query performance, and operational dependencies to ensure a seamless transition with minimal disruption. This section outlines the key steps involved in workload discovery to facilitate an effective migration from MongoDB to Amazon DocumentDB.

Topics

- Assessing the existing MongoDB deployment
- Identifying data model differences
- Query and performance analysis
- Security and access control review
- Operational and monitoring considerations

Assessing the existing MongoDB deployment

Before migration, it is crucial to evaluate the current MongoDB environment, including:

- **Cluster architecture** — Identify the number of nodes, replica sets, and sharding configurations. When migrating from MongoDB to Amazon DocumentDB, understanding your MongoDB sharding configuration is important because Amazon DocumentDB does not support user-controlled sharding. Applications designed for a sharded MongoDB environment will need architectural changes, as Amazon DocumentDB uses a different scaling approach with its storage-based architecture. You'll need to adapt your data distribution strategy and possibly consolidate sharded collections when moving to Amazon DocumentDB.
- **Storage and data volume** — Measure the total data size and index size of your cluster. Complement this with the Oplog review tool to understand write patterns and data growth velocity. For more information about sizing your cluster, see Instance sizing.
- **Workload patterns** — Analyze read and write throughput, query execution frequency, and indexing efficiency.
- **Operational dependencies** — Document all applications, services, and integrations relying on MongoDB.

Identifying data model differences

Although Amazon DocumentDB is MongoDB-compatible, there are differences in supported features, such as:

- **Transactions** — Amazon DocumentDB supports ACID transactions but with some Limitations.
- **Schema design** — Ensure that document structures, embedded documents, and references align with Amazon DocumentDB's best practices.

Query and performance analysis

Understanding query behavior helps optimize migration and post-migration performance. Key areas to analyze include:

- **Slow queries** — Identify queries with high execution time using MongoDB's profiling tools.
- **Query patterns** — Categorize common query types, including CRUD operations and aggregations.
- **Index usage** — Assess whether indexes are effectively utilized or need optimization in Amazon DocumentDB. To assess index usage and optimize performance in Amazon DocumentDB, use the `$indexStats` aggregation pipeline stage combined with the `explain()` method on your critical queries. Start by running `db.collection.aggregate([{$indexStats{}}])` to

identify which indexes are being used. You can do more detailed analysis by executing you most frequent queries with `explainPlan`.

- **Concurrency & workload distribution** — Evaluate read and write ratios, connection pooling, and performance bottlenecks.

Security and access control review

Authentication and authorization

- **MongoDB RBAC to Amazon DocumentDB IAM and RBAC** — Map MongoDB's role-based access control users and roles to AWS Identity and Access Management (IAM) policies and Amazon DocumentDB SCRAM authentication users.
- **User migration strategy** — Plan for migrating database users, custom roles, and privileges to Amazon DocumentDB's supported authentication mechanisms.
- **Privilege differences** — Identify MongoDB privileges without direct Amazon DocumentDB equivalents (for example, cluster administration roles).
- **Application authentication** — Update connection strings and credential management for Amazon DocumentDB's password policies. You can use secrets manager to store your credentials and rotate passwords.
- **Service account management** — Establish processes for managing service account credentials in AWS Secrets Manager.
- **Least privilege implementation** — Review and refine access controls to implement least privilege principles in the new environment.

Encryption

Ensure encryption at rest and in transit aligns with compliance requirements.

Network configuration

Plan for Virtual Private Cloud (VPC) setup and security group rules.

Operational and monitoring considerations

To maintain system reliability, workload discovery should also include:

- **Backup and restore strategy** — Evaluate existing backup methods and Amazon DocumentDB's backup capabilities.

- **AWS Backup integration** — Leverage AWS Backup for centralized backup management across AWS services including Amazon DocumentDB.
- **CloudWatch metrics** — Map MongoDB monitoring metrics to Amazon DocumentDB CloudWatch metrics for CPU, memory, connections, and storage.
- **Performance Insights** — Implement Amazon DocumentDB Performance Insights to visualize database load and analyze performance issues with detailed query analytics.
- **Profiler** — Configure Amazon DocumentDB profiler to capture slow-running operations (similar to MongoDB's profiler but with Amazon DocumentDB-specific settings).
  - Enable through parameter groups with appropriate thresholds.
  - Analyze profiler data to identify optimization opportunities
- **CloudWatch Events** — Set up event-driven monitoring for Amazon DocumentDB cluster events.
  - Configure notifications for backup events, maintenance windows, and failovers.
  - Integrate with Amazon SNS for alerting and AWS Lambda for automated responses.
- **Audit logging** — Plan for audit logging configuration to track user activity and security-relevant events.
- **Enhanced monitoring** — Enable enhanced monitoring for granular OS-level metrics at 1-second intervals.

Create new target cluster

Create an Amazon DocumentDB cluster following the steps in Creating an Amazon DocumentDB cluster. When configuring the cluster for migration, apply these recommendations:

- **Instance class:** Choose R8G for best performance. If R8G is unavailable in your target Region, use R6G. Choose as large an instance as possible for best full-load throughput. Scale down after migration is complete.
- **Engine version:** latest (8.0.1)
- **Number of instances:** Choose 1 instance to minimize costs during migration. Scale to 3 instances for high availability after the full-load migration is complete.
- **Network settings:** Ensure your Amazon DocumentDB security group allows inbound connections from the DMS replication instance's security group (for online migrations) or from the EC2 migration host (for offline migrations using mongorestore).

Index migration

Migrating from MongoDB to Amazon DocumentDB involves transferring not just data but also indexes to maintain query performance and optimize database operations. This section outlines the detailed step-by-step process for migrating indexes from MongoDB to Amazon DocumentDB while ensuring compatibility and efficiency.

Using the Amazon DocumentDB index tool

Clone the index tool on GitHub

```
git clone https://github.com/awslabs/amazon-documentdb-tools.git
cd amazon-documentdb-tools/index-tool
```

```
pip install -r requirements.txt
```

Export indexes from MongoDB (if migrating from MongoDB)

```
python3 migrationtools/documentdb_index_tool.py \
  --dump-indexes \
  --dir index_export \
  --uri 'mongodb://<username>:<password>@<source_endpoint>:27017'
```

Verify indexes

```
python3 migrationtools/documentdb_index_tool.py \
  --show-issues \
  --dir index_export
```

Import indexes

```
python3 migrationtools/documentdb_index_tool.py \
  --restore-indexes \
  --skip-incompatible \
  --dir index_export \
  --uri 'mongodb://<username>:<password>@<target_endpoint>:27017/?
tls=true&tlsCAFile=global-bundle.pem&replicaSet=rs0&retryWrites=false'
```

User migration

Migrating users from MongoDB to Amazon DocumentDB is essential for maintaining access control, authentication, and database security. This section outlines detailed steps to successfully migrate MongoDB users while preserving their roles and permissions using the Amazon DocumentDB export user tool.

Using Amazon DocumentDB export users tool

The Export Users tool exports users and roles from MongoDB or Amazon DocumentDB to JavaScript files, which can then be used to recreate them in another cluster.

Prerequisites

```
# Clone the repository
git clone https://github.com/awslabs/amazon-documentdb-tools.git
cd amazon-documentdb-tools/migration/export-users
```

```
# Install required dependencies
pip install pymongo
```

Step 1: Export users and roles

```
# Export users and roles to JavaScript files
python3 docdbExportUsers.py \
  --users-file mongodb-users.js \
  --roles-file mongodb-roles.js \
  --uri "mongodb://<username>:<password>@<source_endpoint>:27017"
```

Step 2: Edit the Users File

Passwords are not exported for security reasons. Open the `mongodb-users.js` file and add a password for each user by replacing the pwd value in each `createUser` statement:

```
db.getSiblingDB("admin").createUser({
  user: "appuser",
  pwd: "REPLACE_THIS_PASS",
  roles: [
    { role: "readWrite", db: "mydb" }
```

```
  ]
})
```

Step 3: Restore Custom Roles to Amazon DocumentDB

```
# Import roles first
mongosh \
  --tls \
  --host <target_endpoint>:27017 \
  --tlsCAFile global-bundle.pem \
  --username <username> \
  --password <password> \
  mongodb-roles.js
```

Step 4: Restore Users to Amazon DocumentDB

```
# Import users after roles are created
mongosh \
  --tls \
  --host <target_endpoint>:27017 \
  --tlsCAFile global-bundle.pem \
  --username <username> \
  --password <password> \
  mongodb-users.js
```

Important notes

- Passwords are not exported for security reasons and must be manually added to the users.js file.
- Roles must be imported before users to ensure proper role assignments.
- The tool generates JavaScript files that can be directly executed with the mongosh shell.
- Custom roles and their privileges are preserved during migration.
- This approach allows for review and modification of user permissions before importing.

This method provides a secure and flexible approach to migrating users and roles from MongoDB to Amazon DocumentDB while allowing for password resets during the migration process.

Data migration

Topics

- Online migration
- Offline migration

Online migration

This section provides detailed steps to perform an online migration from MongoDB to Amazon DocumentDB using AWS DMS to enable minimal downtime and continuous replication. To begin, you set up an Amazon DocumentDB cluster as the target and ensure your MongoDB instance is properly configured as the source, typically requiring replica set mode for change data capture. Next, you create a DMS replication instance and define source and target endpoints with the necessary connection details. After validating the endpoints, you configure and start a migration task that can include full data load, ongoing replication, or both.

Create new target cluster

See Create new target cluster.

Configure source

MongoDB and Amazon DocumentDB can both serve as migration sources, depending on your scenario:

- **MongoDB as source** — Common when migrating from an on-premises or a self-managed MongoDB to an Amazon DocumentDB or other AWS database services. Requires running in replica set mode with an adequately sized oplog (make sure it is sized to hold all operations during Full Load) to support change data capture during migration.
- **Amazon DocumentDB as source** — Typically used for cross-Region replication, version upgrades, or migrating to other database services like MongoDB Atlas. Requires Enabling change streams by setting the `change_stream_log_retention_duration` parameter in the cluster parameter group to capture ongoing changes during migration. Make sure your `change_stream_log_retention_duration` setting is large enough to cover the time needed to complete the Full Load.

Before starting migration, configure your source to allow AWS DMS access.

Create a MongoDB user with proper permissions:

```
db.createUser({
```

```
  user: "dmsUser",
  pwd: "<password>",
  roles: [{ role: "readAnyDatabase", db: "admin" }]
})
```

Configure network and authentication.

When configuring network connectivity for MongoDB to DMS migration:

EC2-hosted MongoDB source

- Modify the EC2 security group to allow inbound traffic from the DMS replication instance security group.
- Add a rule for TCP port 27017 (or your custom MongoDB port).
- Use the DMS replication instance's security group ID as the source for precise access control.
- Ensure the EC2 instance's subnet has a route to the DMS replication instance's subnet.

On-premises MongoDB source

- Configure your firewall to allow inbound connections from the DMS replication instance's public IP addresses.
- If using Direct Connect or a VPN, ensure proper routing between your network and the VPC containing the DMS instance.
- Test connectivity using telnet or nc commands from the DMS subnet to your MongoDB server.

MongoDB Atlas source

- Add a DMS replication instance IP addresses to the MongoDB Atlas IP allowlist.
- Configure VPC peering between AWS VPC and MongoDB Atlas VPC if Atlas is running on AWS.
- Set up AWS PrivateLink for private connectivity (Enterprise tier), if running on another cloud provider.
- Create a dedicated user with appropriate read/write permissions.
- Use a MongoDB Atlas connection string with SSL Mode set to "verify-full".
- Ensure sufficient oplog size for migration duration.

Amazon DocumentDB source

Configure your source Amazon DocumentDB security group to allow inbound traffic from the DMS replication instance security group.

Create DMS replication instance

Use DMS Buddy on GitHub to create optimal migration infrastructure with optimal DMS settings and instance sizes. If you prefer to configure manually, follow these steps:

1. In the AWS DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Provisioned instances**.
2. Enter replication instance details:
   - **Instance name**: Choose a unique name.
   - **Instance class**: Select based on workload. Example: dms.r7i.large (small workloads), dms.r7i.4xlarge (large workloads).
   - **Engine version**: 3.5.4
   - **Allocated storage**: Default is 50 GB (increase if needed). This is determined by document size, updates/second and full load duration.
   - **Multi-AZ Deployment**: Enable for high availability, if needed.
   - Choose the same VPC as Amazon DocumentDB.
   - Ensure **Security groups** allow inbound traffic from source and Amazon DocumentDB.
3. Choose **Create replication instance** and wait for the status to be available.

Create DMS endpoints

Create a source endpoint

For a MongoDB source

1. In the DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Endpoints**.
2. Choose **Create endpoint**.
3. On the **Create endpoint** page, choose **Source endpoint**.
4. In the **Endpoint configuration** section:
   - Enter a unique and meaningful **Endpoint identifier** (for example, "mongodb-source").
   - Choose **MongoDB** as the **Source engine**.

- For **Access to endpoint database**, choose **Provide access information manually**.
- For **Server name**, enter your *MongoDB server DNS name/IP address*.
- For **Port**, enter **27017** (default MongoDB port).
- For **Authentication mode**, choose the appropriate mode for your application (password/SSL) (default is secrets manager).
- If **Authentication mode** is **Password**, provide:
  - **Username** and **Password**: Enter MongoDB credentials.
  - **Database name**: Your source database name.
  - **Authentication mechanism**: SCRAM-SHA-1 (default) or appropriate mechanism

5. For **Metadata mode**, leave the default setting of **document**.
6. Additional connection attributes:

- authSource=admin (if authentication database is different)
- replicaSet=<your-replica-set-name> (required for CDC)

For an Amazon DocumentDB source

1. In the DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Endpoints**.
2. Choose **Create endpoint**.
3. On the **Create endpoint** page, choose **Source endpoint**.
4. In the **Endpoint configuration** section:

- Enter a unique and meaningful **Endpoint identifier** (for example, "docdb-source").
- Choose **Amazon DocumentDB** as the **Source engine**.
- For **Access to endpoint database**, choose **Provide access information manually**.
- For **Server name**, enter your *source Amazon DocumentDB cluster endpoint*.
- For **Port**, enter **27017** (default Amazon DocumentDB port).
- For **SSL mode**, choose **verify-full** (recommended for Amazon DocumentDB).
- For **CA Certificate**, choose the Amazon RDS root CA certificate.
- For **Authentication mode**, choose the appropriate mode for your application (password/SSL) (default is secrets manager).
- If **Authentication mode** is **Password**, provide:

- **Username** and **Password**: Enter Amazon DocumentDB credentials.
- **Database name**: Your source database name.
- **Authentication mechanism**: SCRAM-SHA-1 (default) or appropriate mechanism

5. For **Metadata mode**, leave the default setting of **document**.

Create a target endpoint (Amazon DocumentDB)

1. In the DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Endpoints**.
2. Choose **Create endpoint**.
3. On the **Create endpoint** page, choose **Target endpoint**.
4. In the **Endpoint configuration** section:

   - Enter a unique and meaningful **Endpoint identifier** (for example, "docdb-target").
   - Choose **Amazon DocumentDB** as the **Target engine**.
   - For **Access to endpoint database**, choose the method you want to use to authenticate access to the database:
     - If you choose **AWS Secrets Manager**, choose the secret where you store your Amazon DocumentDB credentials in the **Secret** field.
     - If you choose **Provide access information manually**:
       - For **Server name**, enter your *`target Amazon DocumentDB cluster endpoint`*.
       - For **Port**, enter **27017** (default Amazon DocumentDB port).
       - For **SSL mode**, choose **verify-full** (recommended for Amazon DocumentDB).
       - For **CA Certificate**, download and specify the CA certificate bundle for SSL verification.
       - For **Authentication mode**, choose the appropriate mode for your application (password/SSL) (default is secrets manager).
       - If **Authentication mode** is **Password**, provide:
         - **Username** and **Password**: Enter Amazon DocumentDB credentials.
         - **Database name**: Your source database name.
         - **Authentication mechanism**: SCRAM-SHA-1 (default) or appropriate mechanism

5. For **Metadata mode**, leave the default setting of **document**.

Create replication task

1. In the DMS console, in the navigation pane, choose **Migrate or replicate**, then choose **Tasks**.
2. Choose **Create task**.
3. On the **Create task** page, in the **Task configuration** section:
   - Enter a unique and meaningful **Task identifier** (for example, "mongodb-docdb-replication").
   - Choose the source endpoint you created previously in the **Source database endpoint** drop-down menu.
   - Choose the target endpoint you created previously in the **Target database endpoint** drop-down menu.
   - Choose the replication instance you created previously in the **Provisioned instance** drop-down menu.
   - For **Task type**, choose **Migrate and replicate**.
   - For **How long do you intend to continue replicating from source to target?**, choose **For a limited time**.
4. In the **Settings** section:
   - For **Target table preparation mode**, choose **Do nothing**. Otherwise, the collections and indexes that were already created will be deleted.
   - For **Task logs**, check the **Turn on CloudWatch** logs box.
   - Leave default values for all other settings.
5. Back at the top of the **Settings** section, for **Editing mode**, choose **JSON editor** and set the following attributes:

```
{
  "TargetMetadata": {
    "ParallelApplyThreads": 5
  },
  "FullLoadSettings": {
    "MaxFullLoadSubTasks": 16
  }
}
```

Set `MaxFullLoadSubTasks` based on your collection count and instance capacity. Use lower values (2–4) for fewer or smaller collections, and higher values (8–16) for many large collections. For automated sizing, see DMS Buddy on GitHub.

6. In the **Table mappings** section, add a new selection rule:
   - For **Schema name**, add the source database to migrate. Use % to specify multiple databases.
   - For **Schema table** name, add the source collection to migrate. Use % to specify multiple collections.
   - For **Action**, leave the default setting of **Include**
7. For large collections (over 100GB), add **Table settings rule**:
   - For **Schema name**, add the source database to migrate. Use % to specify multiple databases.
   - For **Schema table** name, add the source collection to migrate. Use % to specify multiple collections.
   - For **Number of partitions**, set based on your workload. For a small number of large collections, set partitions ≥ `MaxFullLoadSubTasks` to keep all workers busy. For many collections loading in parallel, set partitions ≤ `MaxFullLoadSubTasks` since the collection count itself provides sufficient parallelism.
8. In the **Premigration assessment** section, make sure it is turned off. Premigration assessment is not supported for Amazon DocumentDB as a target and must be disabled.

Offline migration

This section outlines the process to perform an offline migration from a self-managed MongoDB instance to Amazon DocumentDB using native MongoDB tools: `mongodump` and `mongorestore`.

Prerequisites

Source MongoDB requirements

- Access to the source MongoDB instance with appropriate permissions.
- Install `mongodump`. if needed (it is installed during a MongoDB installation).
- Make sure there is enough disk space for the dump files.

Target Amazon DocumentDB requirements

- Make sure you have an Amazon DocumentDB cluster provisioned.
- Ensure there is an EC2 instance in the same VPC as Amazon DocumentDB to facilitate the migration.
- Network connectivity must be available between your source environment and Amazon DocumentDB.
- **mongorestore** must be installed on the migration EC2 instance.
- Appropriate IAM permissions must be configured to access Amazon DocumentDB,

General requirements

- AWS CLI must be configured (if using AWS services for intermediate storage)
- Sufficient bandwidth must be available for data transfer.
- Downtime window should be approved (if doing a live migration, consider other approaches)

Prepare an Amazon DocumentDB cluster

See Create new target cluster.

Perform the data dump (mongodump)

Choose one of the following options to create a dump file:

- **Option 1: Basic**

```
mongodump \
  --uri="mongodb://<user>:<password>@<source_endpoint>:27017/<database>" \
  --out=/path/to/dump
```

- **Option 2: Better control and performance**

  Compresses the output with gzip, dumps collections in parallel, enables TLS, and reads from a secondary replica if available:

```
mongodump \
  --uri="mongodb://<user>:<password>@<source_endpoint>:27017" \
  --out=/path/to/dump \
  --gzip \
  --numParallelCollections=4 \
```

```
  --tls \
  --authenticationDatabase=admin \
  --readPreference=secondaryPreferred
```

- **Option 3: Large databases**

  Dumps a specific database and collection, filters documents by query, and outputs to a single compressed archive:

```
mongodump \
  --host=<source_endpoint> \
  --port=27017 \
  --username=<user> \
  --password=<password> \
  --db=<specific_db> \
  --collection=<specific_collection> \
  --query='{ "date": { "$gt": "2020-01-01" } }' \
  --archive=/path/to/archive.gz \
  --gzip \
  --tls
```

Transfer dump files to restoration environment

Choose an appropriate method based on your dump size:

- **Small** — Directly copy to your migration machine (EC2 instance you created earlier):

```
scp -r /path/to/dump user@migration-machine:/path/to/restore
```

- **Medium** — Use Amazon S3 as intermediate storage:

```
aws s3 cp --recursive /path/to/dump s3://your-bucket/mongodb-dump/
```

- **Large** — For very large databases, consider AWS DataSync or a physical transfer.

Restore data to Amazon DocumentDB (mongorestore)

Before starting the restore process, create the indexes in Amazon DocumentDB. You can utilize the Amazon DocumentDB Index tool to export and import indexes.

Choose one of the following options to restore data:

- **Option 1: Basic restore**

```
mongorestore \
  --uri="mongodb://<user>:<password>@<target_endpoint>:27017" \
  --tls \
  --tlsCAFile=global-bundle.pem \
  /path/to/dump
```

- **Option 2: Better control and performance**

  Restores gzip-compressed dumps with parallel collection and document insertion. Skips index restoration since indexes are pre-created:

```
mongorestore \
  --uri="mongodb://<user>:<password>@<target_endpoint>:27017" \
  --tls \
  --tlsCAFile=global-bundle.pem \
  --gzip \
  --numParallelCollections=4 \
  --numInsertionWorkersPerCollection=4 \
  --noIndexRestore \
  /path/to/dump
```

- **Option 3: Large databases or specific controls**

  Restores from a compressed archive with namespace filtering to include or exclude specific databases and collections. Skips index restoration since indexes are pre-created:

```
mongorestore \
  --host=<target_endpoint> \
  --port=27017 \
  --username=<user> \
  --password=<password> \
  --tls \
  --tlsCAFile=global-bundle.pem \
  --archive=/path/to/archive.gz \
  --gzip \
  --nsInclude="db1.*" \
  --nsExclude="db1.sensitive_data" \
  --noIndexRestore
```

Monitoring

This section provides a detailed monitoring process to track the progress, performance, and health of an ongoing migration.

The monitoring steps apply regardless of the migration method (AWS DMS, mongodump/mongorestore, or other tools).

AWS DMS Migration monitoring (if applicable)

Monitor the following key CloudWatch metrics:

Full load phase metrics

- **FullLoadThroughputBandwidthTarget** — Network bandwidth (KB/second) during full load
- **FullLoadThroughputRowsTarget** — Number of rows/documents loaded per second
- **FullLoadThroughputTablesTarget** — Number of tables/collections completed per minute
- **FullLoadProgressPercent** — Percentage of full load completed
- **TablesLoaded** — Number of tables/collections successfully loaded
- **TablesLoading** — Number of tables/collections currently loading
- **TablesQueued** — Number of tables/collections waiting to be loaded
- **TablesErrored** — Number of tables/collections that failed to load

CDC phase metrics

- **CDCLatencyTarget** — Time delay (seconds) between source change and target application
- **CDCLatencySource** — Time delay (seconds) between change in source and DMS reading it
- **CDCThroughputRowsTarget** — Rows per second applied during ongoing replication
- **CDCThroughputBandwidthTarget** — Network bandwidth (KB/second) during CDC
- **CDCIncomingChanges** — Number of change events received from source
- **CDCChangesMemoryTarget** — Memory used (MB) for storing changes on target side

Resource metrics

- **CPUUtilization** — CPU usage of the replication instance

- **FreeableMemory** — Available memory on the replication instance
- **FreeStorageSpace** — Available storage on the replication instance
- **NetworkTransmitThroughput** — Network throughput for the replication instance
- **NetworkReceiveThroughput** — Network throughput for the replication instance

Error metrics

- **ErrorsCount** — Total number of errors during migration
- **TableErrorsCount** — Number of table-specific errors
- **RecordsErrorsCount** — Number of record-specific errors

Create CloudWatch alarms for critical metrics like `CDCLatencyTarget` and `CPUUtilization` to receive notifications if migration performance degrades.

DMS logs (CloudWatch logs)

1. Go to Amazon CloudWatch Logs console.
2. Find and choose on your log group. It will look similar to "dms-tasks –".
3. Look for log streams that might contain error information:
   - Streams with "error" in the name
   - Streams with task IDs or endpoint names
   - The most recent log streams during the time of your migration
4. Within these streams, search for keywords like:
   - "error"
   - "exception"
   - "failed"
   - "warning"

DMS task status (using AWS CLI)

```
aws dms describe-replication-tasks \
  --filters Name=replication-task-id,Values=<task_id> \
```

```
  --query "ReplicationTasks[0].Status"
```

Expected status flow:

creating → ready → running → stopping → stopped (or failed)

Monitor using `docdb-dashboarder`

The `docdb-dashboarder` tool provides comprehensive monitoring for Amazon DocumentDB clusters by automatically generating CloudWatch dashboards with essential performance metrics. These dashboards display critical cluster-level metrics (replica lag, operation counters), instance-level metrics (CPU, memory, connections), and storage metrics (volume usage, backup storage). For migration scenarios, the tool offers specialized dashboards that track migration progress with metrics like CDC replication lag and operation rates. The dashboards can monitor multiple clusters simultaneously and include support for NVMe-backed instances. By visualizing these metrics, teams can proactively identify performance bottlenecks, optimize resource allocation, and ensure smooth operation of their Amazon DocumentDB deployments. The tool eliminates the need for manual dashboard creation while providing consistent monitoring across all environments. For setup instructions and advanced configuration options, refer to the Amazon DocumentDB Dashboarder Tool GitHub repository.

Validation

Topics

- Validation checklist
- Schema and index validation
- Data sampling and field-level validation
- Validation using DataDiffer tool

This section provides a detailed validation process to ensure data consistency, integrity, and application compatibility after migrating.

The validation steps apply regardless of the migration method (AWS DMS, mongodump/mongorestore, or other tools).

Validation checklist

Verify that the number of documents in each collection matches between source and target:

MongoDB source

```
mongosh \
  --host <source_endpoint> \
  --port 27017 \
  --username <user> \
  --password <password> \
  --eval "db.<collection>.countDocuments()"
```

Amazon DocumentDB target

```
mongosh \
  --host <target_endpoint> \
  --port 27017 \
  --username <user> \
  --password <password> \
  --tls \
  --tlsCAFile global-bundle.pem \
  --eval "db.<collection>.countDocuments()"
```

Schema and index validation

Ensure that:

- all collections exist in the target.
- indexes are correctly replicated.
- schema definitions (if enforced) are identical.

Check collections (source vs. target)

```
mongosh --host <source_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --eval "db = db.getSiblingDB('<database>'); db.getCollectionNames();"

mongosh --host <target_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --tls --tlsCAFile global-bundle.pem \
```

```
  --eval "db = db.getSiblingDB('<database>'); db.getCollectionNames();"
```

Check indexes (source vs. target)

```
mongosh --host <source_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --eval "db.<collection>.getIndexes()"

mongosh --host <target_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --tls --tlsCAFile global-bundle.pem \
  --eval "db.<collection>.getIndexes()"
```

Compare the list of collections to ensure there are no missing or extra collections.

Verify indexes by checking index names, key definitions, unique constraints, and TTL indexes (if any).

Check schema validation rules (if using schema validation in MongoDB)

```
mongosh --host <source_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --eval "db.getCollectionInfos({name: '<collection>'})[0].options.validator"

mongosh --host <target_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --tls --tlsCAFile global-bundle.pem \
  --eval "db.getCollectionInfos({name: '<collection>'})[0].options.validator"
```

Data sampling and field-level validation

You can randomly sample documents and compare fields between source and target.

Manual sampling

Fetch five random documents (source):

```
mongosh --host <source_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --eval "db.<collection>.aggregate([{ \$sample: { size: 5 } }])"
```

Fetch the same document IDs (target):

```
mongosh --host <target_endpoint> \
  --port 27017 \
  --username <username> \
  --password <password> \
  --tls --tlsCAFile global-bundle.pem \
  --eval "db.<collection>.find({ _id: { \$in: [<list_of_ids>] } })"
```

Automatic sampling

```
import pymongo

# Connect to source and target
source_client = pymongo.MongoClient("<source_uri>")
target_client = pymongo.MongoClient("<target_uri>",
                                    tls=True,
                                    tlsCAFile='global-bundle.pem')
source_db = source_client["<db_name>"]
target_db = target_client["<db_name>"]

# Compare 100 random documents
for doc in source_db["<collection>"].aggregate([{"$sample": {"size": 100}}]):
    target_doc = target_db["<collection>"].find_one({"_id": doc["_id"]})
    if target_doc != doc:
        print(f"Mismatch in _id: {doc['_id']}")
    else:
        print(f"Match: {doc['_id']}")
```

Validation using DataDiffer tool

The DataDiffer tool on GitHub provides a reliable way to compare data between source and target databases.

Prerequisites

The following prerequisites must be met before installing the DataDiffer tool:

- Python 3.7+
- PyMongo library
- Network connectivity to both source MongoDB and target Amazon DocumentDB clusters

Setup and installation

Clone the repository and navigate to the DataDiffer directory

```
git clone https://github.com/awslabs/amazon-documentdb-tools.git
cd amazon-documentdb-tools/migration/data-differ
```

Install required dependencies

```
pip install -r requirements.txt
```

Running data validation

Create a configuration file (e.g., config.json) with connection details

```
{
  "source": {
    "uri": "mongodb://<username>:<password>@<source_endpoint>:27017/?replicaSet=rs0",
    "db": "your_database",
    "collection": "your_collection"
  },
  "target": {
    "uri": "mongodb://<username>:<password>@<target_endpoint>:27017/?
tls=true&tlsCAFile=global-bundle.pem&replicaSet=rs0&retryWrites=false",
    "db": "your_database",
    "collection": "your_collection"
  },
  "options": {
    "batch_size": 1000,
    "threads": 4,
    "sample_size": 0,
    "verbose": true
  }
```

```
}
```

Run the DataDiffer tool

```
python differ.py --config config.json
```

For large collections, use sampling to validate a subset of data

```
python differ.py --config config.json --sample-size 10000
```

To validate multiple collections, create separate configuration files or use the batch mode

```
python differ.py --batch-config batch_config.json
```

Interpreting results

The tool will output:

- Total documents in source and target
- Number of matching documents
- Number of missing documents
- Number of documents with differences
- Detailed report of differences (if any)

Best practices

The following are best practices when using the DataDiffer tool:

- **Run in phases** — First validate document counts, then sample key documents, and finally run a full comparison, if needed.
- **Check for schema differences** — Amazon DocumentDB has some limitations compared to MongoDB. The tool will highlight incompatible data types or structures.
- **Validate during quiet periods** — Run validation when write operations are minimal to ensure consistency.
- **Monitor resource usage** — The comparison process can be resource-intensive. Adjust batch size and thread count accordingly.

- **Validate indexes** — After data validation, ensure all required indexes have been created on the target Amazon DocumentDB cluster.
- **Document validation results** — Keep a record of validation results for each collection as part of your migration documentation.

Migration from Couchbase Server

Topics

- Introduction
- Comparison to Amazon DocumentDB
- Discovery
- Planning
- Migration
- Validation

Introduction

This guide presents the key points to consider when migrating from Couchbase Server to Amazon DocumentDB. It explains considerations for the discovery, planning, execution, and validation phases of your migration. It also explains how to perform offline and online migrations.

Comparison to Amazon DocumentDB

	Couchbase Server	Amazon DocumentDB
Data Organization	In versions 7.0 and later, data is organized into buckets, scopes, and collections. In earlier versions, data is organized into buckets.	Data is organized into databases and collections.
Compatibility	There are separate APIs for each service (e.g. data, index, search, etc.). Secondary lookups use SQL ++ (formerly known an N1QL); a query language based on ANSI-	Amazon DocumentDB is compatible with the MongoDB API.

	Couchbase Server	Amazon DocumentDB
	standard SQL so it is familiar to many developers.	
Architecture	Storage is attached to each cluster instance. You cannot scale compute independently of storage.	Amazon DocumentDB is designed for the cloud and to avoid the limitations of traditional database architectures. The compute and storage layers are separated in Amazon DocumentDB and the compute layer can be scaled independently of storage.
Add read capacity on demand	Clusters can be scaled out by adding instances. Since storage is attached to the instance where the service is running, the time it takes to scale out is dependent on the amount of data that needs to be moved to the new instance, or rebalanced.	You can achieve read scaling for your Amazon DocumentDB cluster by creating up to 15 Amazon DocumentDB replicas in the cluster. There is no impact to the storage layer.
Recover quickly from node failure	Clusters have automatic failover capabilities but the time to get the cluster back to full strength is dependent on the amount of data that needs to be moved to the new instance.	Amazon DocumentDB can failover the primary typically within 30 seconds and restore the cluster back to full strength in 8-10 minutes regardless of the amount of data in the cluster.
Scale storage as data grows	For self-managed clusters storage and IOs do not scale automatically.	Amazon DocumentDB storage and IOs scale automatically.

	Couchbase Server	Amazon DocumentDB
Backup data without affecting performance	Backups are performed by the backup service and are not enabled by default. Since storage and compute are not separated there can be an impact to performance.	Amazon DocumentDB backups are enabled by default and cannot be turned off. Backups are handled by the storage layer, so they are zero-impact on the compute layer. Amazon DocumentDB supports restoring from a cluster snapshot and restoring to a point in time.
Data durability	There can be a maximum of 3 replica copies of data in a cluster for a total of 4 copies. Each instance where the data service is running will have active and 1, 2, or 3 replica copies of the data.	Amazon DocumentDB maintains 6 copies of data no matter how many compute instances there are with a write quorum of 4 and persist true. Clients receive an acknowledgement after the storage layer has persisted 4 copies of the data.
Consistency	Immediate consistency for K/V operations is supported. The Couchbase SDK routes K/V requests to the specific instance that contains the active copy of the data so once an update is acknowledged, the client is guaranteed to read that update. Replication of updates to other services (index, search, analytics, eventing) is eventually consistent.	Amazon DocumentDB replicas are eventually consistent. If immediate consistency reads are required, the client can read from the primary instance.
Replication	Cross-Data Center Replication (XDCR) provides filtered, active-passive/active-active replication of data in many:many topologies.	Amazon DocumentDB global clusters provide active-passive replication in 1:many (up to 10) topologies.

Discovery

Migrating to Amazon DocumentDB requires a thorough understanding of the existing database workload. Workload discovery is the process of analyzing your Couchbase cluster configuration and operational characteristics – data set, indexes, and workload – to help ensure a seamless transition with minimal disruption.

Cluster configuration

Couchbase uses a service-centric architecture where each capability corresponds to a service. Execute the following command against your Couchbase cluster to determine which services are being used (see Getting Information on Nodes):

```
curl -v -u <administrator>:<password> \
  http://<ip-address-or-hostname>:<port>/pools/nodes | \
  jq '[.nodes[].services[]] | unique'
```

Sample output:

```
[
  "backup",
  "cbas",
  "eventing",
  "fts",
  "index",
  "kv",
  "n1ql"
]
```

Couchbase services include the following:

Data service (kv)

The data service provides read/write access to data in memory and on disk.

Amazon DocumentDB supports K/V operations on JSON data via the MongoDB API.

Query service (n1ql)

The query service supports the querying of JSON data via SQL++.

Amazon DocumentDB supports the querying of JSON data via the MongoDB API.

Index service (index)

The index service creates and maintains indexes on data, enabling faster querying.

Amazon DocumentDB supports a default primary index and the creation of secondary indexes on JSON data via the MongoDB API.

Search service (fts)

The search service supports the creation of indexes for full text search.

The native full text search feature of Amazon DocumentDB allows you to perform basic text queries on large textual data sets using special purpose text indexes by using the MongoDB API. For advanced capabilities such as fuzzy search, multilingual search, and cross-collection search, use Amazon DocumentDB zero-ETL integration with Amazon OpenSearch Service.

Analytics service (cbas)

The analytics service supports analyzing JSON data in near real-time.

Amazon DocumentDB supports ad-hoc queries on JSON data via the MongoDB API. You can also run complex queries on your JSON data in Amazon DocumentDB using Apache Spark running on Amazon EMR.

Eventing service (eventing)

The eventing service executes user-defined business logic in response to data changes.

Amazon DocumentDB automates event-driven workloads by invoking AWS Lambda functions each time that data changes with your Amazon DocumentDB cluster.

Backup service (backup)

The backup service schedules full and incremental data backups and merges of previous data backups.

Amazon DocumentDB continuously backs up your data to Amazon S3 with a retention period of 1–35 days so that you can quickly restore to any point within the backup retention period. Amazon

DocumentDB also takes automatic snapshots of your data as part of this continuous backup process. You can also manage backup and restore of Amazon DocumentDB with AWS Backup..

Operational characteristics

Use the Discovery Tool for Couchbase to get the following information about your data set, indexes, and workload. This information will help you size your Amazon DocumentDB cluster.

Data set

The tool retrieves the following bucket, scope, and collection information:

1. bucket name
2. bucket type
3. scope name
4. collection name
5. total size (bytes)
6. total items
7. item size (bytes)

Indexes

The tool retrieves the following index statistics and all index definitions for all buckets. Note that primary indexes are excluded since Amazon DocumentDB automatically creates a primary index for each collection.

1. bucket name
2. scope name
3. collection name
4. index name
5. index size (bytes)

Workload

The tool retrieves K/V and N1QL query metrics. K/V metric values are gathered at the bucket level and SQL++ metrics are gathered at the cluster level.

The tool command line options are as follows:

```
python3 discovery.py \
  --username <source cluster username> \
  --password <source cluster password> \
  --data_node <data node IP address or DNS name> \
  --admin_port <administration http REST port> \
  --kv_zoom <get bucket statistics for specified interval> \
  --tools_path <full path to Couchbase tools> \
  --index_metrics <gather index definitions and SQL++ metrics> \
  --indexer_port <indexer service http REST port> \
  --n1ql_start <start time for sampling> \
  --n1ql_step <sample interval over the sample period>
```

Here is an example command:

```
python3 discovery.py \
  --username username \
  --password ******** \
  --data_node "http://10.0.0.1" \
  --admin_port 8091 \
  --kv_zoom week \
  --tools_path "/opt/couchbase/bin" \
  --index_metrics true \
  --indexer_port 9102 \
  --n1ql_start -60000 \
  --n1ql_step 1000
```

K/V metric values will be based on samples every 10 minutes for the past week (see HTTP method and URI). SQL++ metric values will based on samples every 1 seconds for the past 60 seconds (see General Labels). The output of the command will be in the following files:

collection-stats.csv – bucket, scope, and collection information

```
bucket,bucket_type,scope_name,collection_name,total_size,total_items,document_size
beer-sample,membase,_default,_default,2796956,7303,383
gamesim-sample,membase,_default,_default,114275,586,196
pillowfight,membase,_default,_default,1901907769,1000006,1902
travel-sample,membase,inventory,airport,547914,1968,279
travel-sample,membase,inventory,airline,117261,187,628
travel-sample,membase,inventory,route,13402503,24024,558
travel-sample,membase,inventory,landmark,3072746,4495,684
```

```
travel-sample,membase,inventory,hotel,4086989,917,4457
...
```

index-stats.csv – index names and sizes

```
bucket,scope,collection,index-name,index-size
beer-sample,_default,_default,beer_primary,468144
gamesim-sample,_default,_default,gamesim_primary,87081
travel-sample,inventory,airline,def_inventory_airline_primary,198290
travel-sample,inventory,airport,def_inventory_airport_airportname,513805
travel-sample,inventory,airport,def_inventory_airport_city,487289
travel-sample,inventory,airport,def_inventory_airport_faa,526343
travel-sample,inventory,airport,def_inventory_airport_primary,287475
travel-sample,inventory,hotel,def_inventory_hotel_city,497125
...
```

kv-stats.csv – get, set, and delete metrics for all buckets

```
bucket,gets,sets,deletes
beer-sample,0,0,0
gamesim-sample,0,0,0
pillowfight,369,521,194
travel-sample,0,0,0
```

n1ql-stats.csv – SQL++ select, delete, and insert metrics for the cluster

```
selects,deletes,inserts
0,132,87
```

indexes-<bucket-name>.txt – index definitions of all indexes in the bucket. Note that primary indexes are excluded since Amazon DocumentDB automatically creates a primary index for each collection.

```
CREATE INDEX `def_airportname` ON `travel-sample`(`airportname`)
CREATE INDEX `def_city` ON `travel-sample`(`city`)
CREATE INDEX `def_faa` ON `travel-sample`(`faa`)
CREATE INDEX `def_icao` ON `travel-sample`(`icao`)
CREATE INDEX `def_inventory_airport_city` ON `travel-
sample`.`inventory`.`airport`(`city`)
CREATE INDEX `def_inventory_airport_faa` ON `travel-
sample`.`inventory`.`airport`(`faa`)
```

```
CREATE INDEX `def_inventory_hotel_city` ON `travel-sample`.`inventory`.`hotel`(`city`)
CREATE INDEX `def_inventory_landmark_city` ON `travel-
sample`.`inventory`.`landmark`(`city`)
CREATE INDEX `def_sourceairport` ON `travel-sample`(`sourceairport`)
...
```

Planning

In the planning phase you will determine Amazon DocumentDB cluster requirements and mapping of the Couchbase buckets, scopes, and collections to Amazon DocumentDB databases and collections.

Amazon DocumentDB cluster requirements

Use the data gathered in the discovery phase to size your Amazon DocumentDB cluster. See Instance sizing for more information about sizing your Amazon DocumentDB cluster.

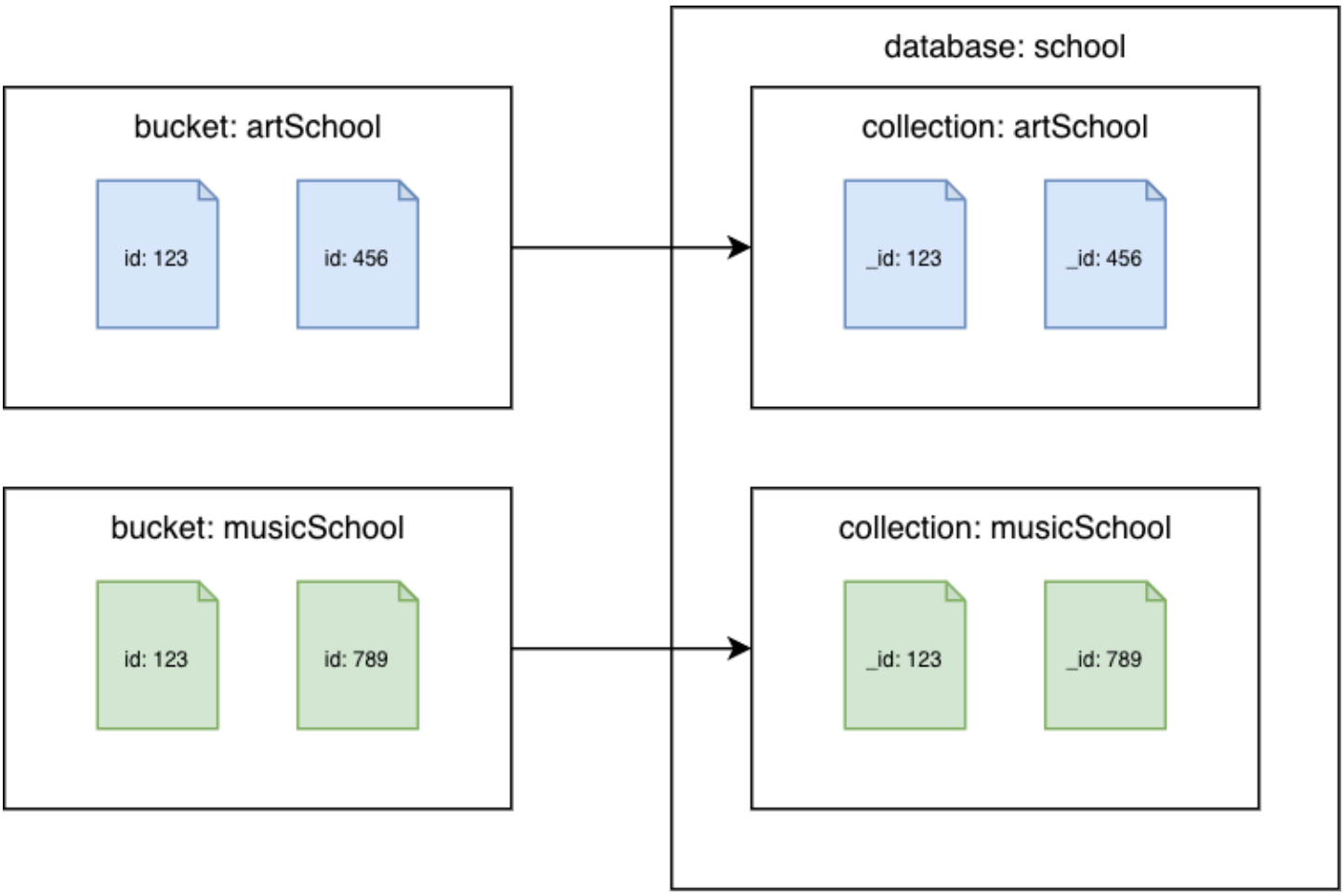
Mapping buckets, scopes, and collections to databases and collections

Determine the databases and collections that will exist in your Amazon DocumentDB cluster(s). Consider the following options depending on how data is organized in your Couchbase cluster. These are not the only options, but they provide starting points for you to consider.

Couchbase Server 6.x or earlier

Couchbase buckets to Amazon DocumentDB collections

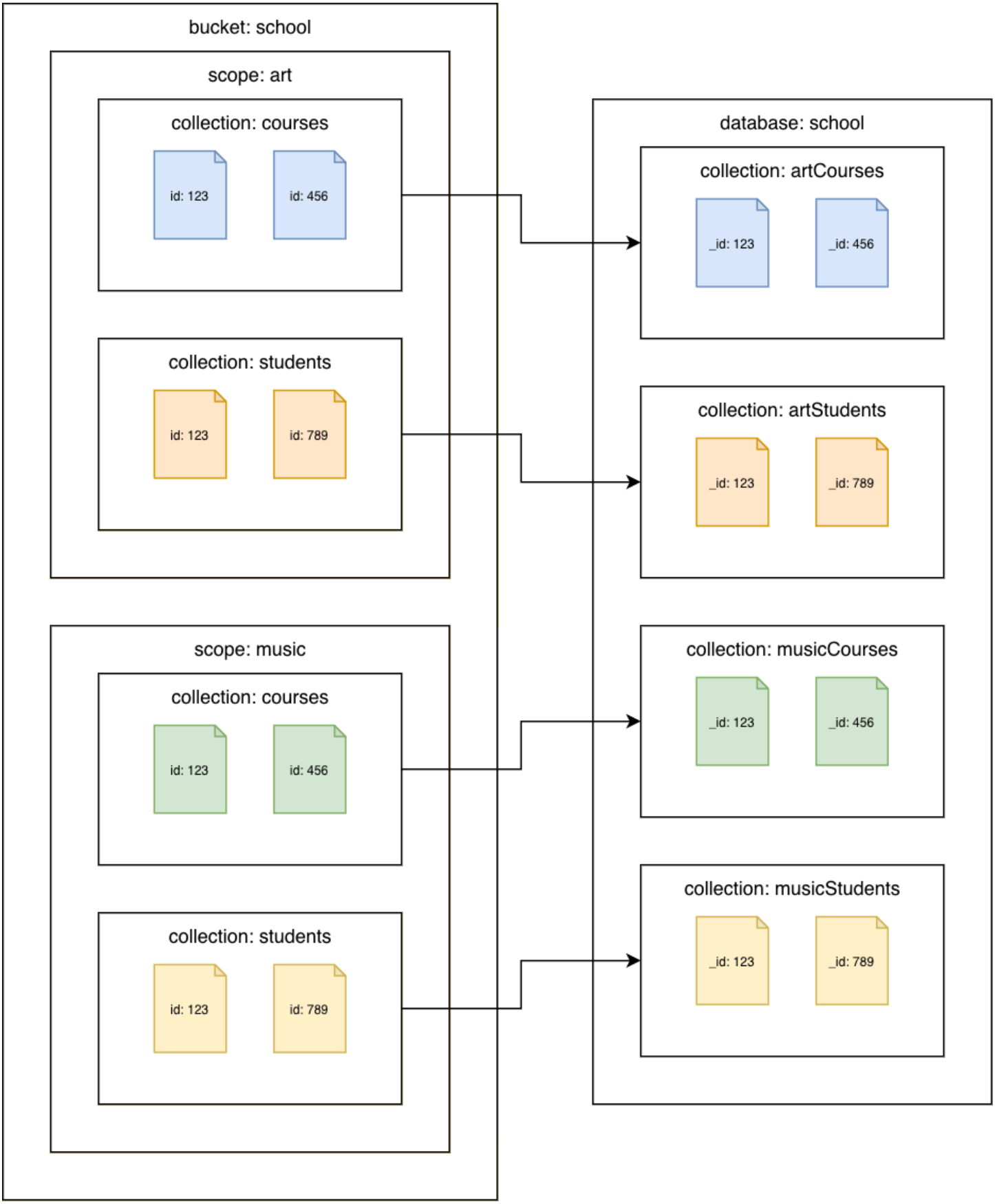
Migrate each bucket to a different Amazon DocumentDB collection. In this scenario, the Couchbase document `id` value will be used as the Amazon DocumentDB `_id` value.



Couchbase Server 7.0 or later

Couchbase collections to Amazon DocumentDB collections

Migrate each collection to a different Amazon DocumentDB collection. In this scenario, the Couchbase document `id` value will be used as the Amazon DocumentDB `_id` value.


bucket: school
scope: art
collection: courses
id: 123
id: 456
collection: students
id: 123
id: 789
scope: music
collection: courses
id: 123
id: 456
collection: students
id: 123
id: 789
database: school
collection: artCourses
_id: 123
_id: 456
collection: artStudents
_id: 123
_id: 789
collection: musicCourses
_id: 123
_id: 456
collection: musicStudents
_id: 123
_id: 789

Migration

Index migration

Migrating to Amazon DocumentDB involves transferring not just data but also indexes to maintain query performance and optimize database operations. This section outlines the detailed step-by-step process for migrating indexes to Amazon DocumentDB while ensuring compatibility and efficiency.

Use Amazon Q (available in the AWS Management Console or your IDE) to convert SQL++ `CREATE INDEX` statements to Amazon DocumentDB `createIndex()` commands.

1. Upload the **indexes-<bucket name>.txt** file(s) created by the Discovery Tool for Couchbase.
2. Enter the following prompt:

   ```
   Convert the Couchbase CREATE INDEX statements to Amazon DocumentDB
   createIndex commands
   ```

Amazon Q will generate equivalent Amazon DocumentDB `createIndex()` commands. Note that you may need to update the collection names based on how you mapped the Couchbase buckets, scopes, and collections to Amazon DocumentDB collections.

For example:

indexes-beer-sample.txt

```
CREATE INDEX `beerType` ON `beer-sample`(`type`)
CREATE INDEX `code` ON `beer-sample`(`code`) WHERE (`type` = "brewery")
```

Example Amazon Q output (excerpt):

```
db.beerSample.createIndex(
  { "type": 1 },
  {
    "name": "beerType",
    "background": true
  }
)

db.beerSample.createIndex(
  { "code": 1 },
```

```
  {
    "name": "code",
    "background": true,
    "partialFilterExpression": { "type": "brewery" }
  }
)
```

For any indexes that Amazon Q is not able to convert, refer to Managing Amazon DocumentDB indexes and Indexes and index properties for more information.

Refactor code to use the MongoDB APIs

Clients use the Couchbase SDKs to connect to Couchbase Server. Amazon DocumentDB clients use MongoDB drivers to connect to Amazon DocumentDB. All the languages supported by the Couchbase SDKs are also supported by MongoDB drivers. See MongoDB Drivers for more information on the driver for your language.

Because the APIs are different between Couchbase Server and Amazon DocumentDB, you will need to refactor your code to use the appropriate MongoDB APIs. You can use Amazon Q to convert the K/V API calls and SQL++ queries to the equivalent MongoDB APIs:

1. Upload the source code file(s).
2. Enter the following prompt:

```
Convert the Couchbase API code to Amazon DocumentDB API code
```

Using the Hello Couchbase Python code sample, Amazon Q generates the following (excerpt):

```
from pymongo import MongoClient

# Connection parameters
database_name = "travel-sample"

# Connect to Amazon DocumentDB cluster
client = MongoClient('<Amazon DocumentDB connection string>',
                     tls=True,
                     tlsCAFile='global-bundle.pem')

# Get reference to database and collection
db = client['travel-sample']
airline_collection = db['airline']
```

```
# upsert document function
def upsert_document(doc):
    print("\nUpsert Result: ")
    try:
        # key will equal: "airline_8091"
        key = doc["type"] + "_" + str(doc["id"])
        doc['_id'] = key # Amazon DocumentDB uses _id as primary key
        result = airline_collection.update_one(
            {'_id': key},
            {'$set': doc},
            upsert=True
        )
        print(f"Modified count: {result.modified_count}")
    except Exception as e:
        print(e)

# get document function
def get_airline_by_key(key):
    print("\nGet Result: ")
    try:
        result = airline_collection.find_one({'_id': key})
        print(result)
    except Exception as e:
        print(e)

# query for document by callsign
def lookup_by_callsign(cs):
    print("\nLookup Result: ")
    try:
        result = airline_collection.find(
            {'callsign': cs},
            {'name': 1, '_id': 0}
        )
        for doc in result:
            print(doc['name'])
    except Exception as e:
        print(e)

# Test document
airline = {
    "type": "airline",
    "id": 8091,
    "callsign": "CBS",
```

```
    "iata": None,
    "icao": None,
    "name": "Couchbase Airways",
}

upsert_document(airline)
get_airline_by_key("airline_8091")
lookup_by_callsign("CBS")
```

Refer to Connecting programmatically to Amazon DocumentDB for examples of connecting to Amazon DocumentDB in Python, Node.js, PHP, Go, Java, C#/.NET, R, and Ruby.

Select the migration approach

When migrating data to Amazon DocumentDB, there are two options:

1. offline migration
2. online migration

> **Note**
>
> Amazon DocumentDB uses the `_id` field as the primary key for each document. The migration approaches in this guide store the Couchbase document key in the `_id` field. For offline migration, this is done via the `--include-key _id` option in `cbexport`. For online migration, this is done via the Kafka connector's `ProvidedInKeyStrategy`.

Offline migration

Consider an offline migration when:

- **Downtime is acceptable:** Offline migration involves stopping write operations to the source database, exporting the data, and then importing it to Amazon DocumentDB. This process incurs downtime for your application. If your application or workload can tolerate this period of unavailability, offline migration is a viable option.
- **Migrating smaller datasets or conducting proofs of concept:** For smaller datasets, the time required for the export and import process is relatively short, making offline migration a quick and simple method. It is also well-suited for development, testing, and proof-of-concept environments where downtime is less critical.

- **Simplicity is a priority:** The offline method, using cbexport and mongoimport, is generally the most straightforward approach to migrate data. It avoids the complexities of change data capture (CDC) involved in online migration methods.
- **No ongoing changes need to be replicated:** If the source database is not actively receiving changes during the migration, or if those changes are not critical to be captured and applied to the target during the migration process, then an offline approach is appropriate.

Topics

- Couchbase Server 6.x or earlier
- Couchbase Server 7.0 or later

Couchbase Server 6.x or earlier

Couchbase bucket to Amazon DocumentDB collection

Export data using cbexport json to create a JSON dump of all data in the bucket. For the `--format` option you can use `lines` or `list`.

```
cbexport json \
  --cluster <source cluster endpoint> \
  --bucket <bucket name> \
  --format <lines | list> \
  --username <username> \
  --password <password> \
  --output export.json \
  --include-key _id
```

Import the data to an Amazon DocumentDB collection using mongoimport with the appropriate option to import the lines or list:

lines:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
  --sslCAFile global-bundle.pem \
```

```
  --file export.json
```

list:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
  --sslCAFile global-bundle.pem \
  --jsonArray \
  --file export.json
```

Couchbase Server 7.0 or later

To perform an offline migration, use the cbexport and mongoimport tools:

Couchbase bucket with default scope and default collection

Export data using cbexport json to create a JSON dump of all collections in the bucket. For the `--format` option you can use `lines` or `list`.

```
cbexport json \
  --cluster <source cluster endpoint> \
  --bucket <bucket name> \
  --format <lines | list> \
  --username <username> \
  --password <password> \
  --output export.json \
  --include-key _id
```

Import the data to an Amazon DocumentDB collection using mongoimport with the appropriate option to import the lines or list:

lines:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
```

```
  --sslCAFile global-bundle.pem \
  --file export.json
```

list:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
  --sslCAFile global-bundle.pem \
  --jsonArray \
  --file export.json
```

Couchbase collections to Amazon DocumentDB collections

Export data using cbexport json to create a JSON dump of each collection. Use the `--include-data` option to export each collection. For the `--format` option you can use `lines` or `list`. Use the `--scope-field` and `--collection-field` options to store the name of the scope and collection in the specified fields in each JSON document.

```
cbexport json \
  --cluster <source cluster endpoint> \
  --bucket <bucket name> \
  --include-data <scope name>.<collection name> \
  --format <lines | list> \
  --username <username> \
  --password <password> \
  --output export.json \
  --include-key _id \
  --scope-field "_scope" \
  --collection-field "_collection"
```

Because cbexport added the `_scope` and `_collection` fields to every exported document, you can remove them from every document in the export file via search and replace, `sed`, or whatever method you prefer.

Import the data for each collection to an Amazon DocumentDB collection using mongoimport with the appropriate option to import the lines or list:

lines:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
  --sslCAFile global-bundle.pem \
  --file export.json
```

list:

```
mongoimport \
  --db <database> \
  --collection <collection> \
  --uri "<Amazon DocumentDB cluster connection string>" \
  --ssl \
  --sslCAFile global-bundle.pem \
  --jsonArray \
  --file export.json
```

Online migration

Consider an online migration when you need to minimize downtime and ongoing changes need to be replicated to Amazon DocumentDB in near-real time.

See How to perform a live migration from Couchbase to Amazon DocumentDB to learn how to perform a live migration to Amazon DocumentDB. The documentation walks you through deploying the solution and performing a live migration a bucket to an Amazon DocumentDB cluster.

Topics

- Couchbase Server 6.x or earlier
- Couchbase Server 7.0 or later

Couchbase Server 6.x or earlier

Couchbase bucket to Amazon DocumentDB collection

The migration utility for Couchbase is pre-configured to perform an online migration of a Couchbase bucket to an Amazon DocumentDB collection. Looking at the sink connector

configuration, the `document.id.strategy` parameter is configured to use the message key value as the `_id` field value (see Sink Connector Id Strategy Properties):

```
ConnectorConfiguration:
  document.id.strategy:
 'com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy'
```

Couchbase Server 7.0 or later

Couchbase bucket with default scope and default collection

The migration utility for Couchbase is pre-configured to perform an online migration of a Couchbase bucket to an Amazon DocumentDB collection. Looking at the sink connector configuration, the `document.id.strategy` parameter is configured to use the message key value as the `_id` field value (see Sink Connector Id Strategy Properties):

```
ConnectorConfiguration:
  document.id.strategy:
 'com.mongodb.kafka.connect.sink.processor.id.strategy.ProvidedInKeyStrategy'
```

Couchbase collections to Amazon DocumentDB collections

Configure the source connector to stream each Couchbase collection in each scope to a separate topic (see Source Configuration Options). For example:

```
ConnectorConfiguration:
  # add couchbase.collections configuration
  couchbase.collections: '<scope 1>.<collection 1>, <scope 1>.<collection 2>, ...'
```

Configure the sink connector to stream from each topic to a separate Amazon DocumentDB collection (see Sink Connector Configuration Properties). For example:

```
ConnectorConfiguration:
  # remove collection configuration
```

```
  #collection: 'test'

  # modify topics configuration
  topics: '<bucket>.<scope 1>.<collection 1>, <bucket>.<scope 1>.<collection 2>, ...'

  # add topic.override.%s.%s configurations for each topic
  topic.override.<bucket>.<scope 1>.<collection 1>.collection: '<collection>'
  topic.override.<bucket>.<scope 1>.<collection 2>.collection: '<collection>'
```

Validation

This section provides a detailed validation process to verify data consistency and integrity after migrating to Amazon DocumentDB. The validation steps apply regardless of the migration method.

Topics

- Verify that all collections exist in the target
- Verify document count between source and target clusters
- Compare documents between source and target clusters

Verify that all collections exist in the target

Couchbase source

option 1: query workbench

```
SELECT RAW `path`
  FROM system:keyspaces
  WHERE `bucket` = '<bucket>'
```

option 2: cbq tool

```
cbq \
  -e <source cluster endpoint> \
  -u <username> \
  -p <password> \
  -q "SELECT RAW `path`
       FROM system:keyspaces
       WHERE `bucket` = '<bucket>'"
```

Amazon DocumentDB target

mongosh (see Connect to your Amazon DocumentDB cluster):

```
db.getSiblingDB('<database>')
db.getCollectionNames()
```

Verify document count between source and target clusters

Couchbase source

Couchbase Server 6.x or earlier

option 1: query workbench

```
SELECT COUNT(*)
FROM `<bucket>`
```

option 2: cbq

```
cbq \
  -e <source cluster endpoint> \
  -u <username> \
  -p <password> \
  -q "SELECT COUNT(*)
      FROM `<bucket>`"
```

Couchbase Server 7.0 or later

option 1: query workbench

```
SELECT COUNT(*)
FROM `<bucket>`.`<scope>`.`<collection>`
```

option 2: cbq

```
cbq \
  -e <source cluster endpoint> \
  -u <username> \
  -p <password> \
  -q "SELECT COUNT(*)
```

```
        FROM `<bucket>`.`<scope>`.`<collection>`"
```

Amazon DocumentDB target

mongosh (see Connect to your Amazon DocumentDB cluster):

```
db = db.getSiblingDB('<database>')
db.getCollection('<collection>').countDocuments()
```

Compare documents between source and target clusters

Couchbase source

Couchbase Server 6.x or earlier

option 1: query workbench

```
SELECT META().id as _id, *
FROM `<bucket>`
LIMIT 5
```

option 2: cbq

```
cbq \
  -e <source cluster endpoint> \
  -u <username> \
  -p <password> \
  -q "SELECT META().id as _id, *
       FROM `<bucket>`
       LIMIT 5"
```

Couchbase Server 7.0 or later

option 1: query workbench

```
SELECT META().id as _id, *
FROM `<bucket>`.`<scope>`.`<collection>`
LIMIT 5
```

option 2: cbq

```
cbq \
```

```
  -e <source cluster endpoint> \
  -u <username> \
  -p <password> \
  -q "SELECT META().id as _id, *
       FROM `<bucket>`.`<scope>`.`<collection>`
       LIMIT 5"
```

Amazon DocumentDB target

mongosh (see Connect to your Amazon DocumentDB cluster):

```
db = db.getSiblingDB('<database>')
db.getCollection('<collection>').find({
  _id: {
    $in: [
      <_id 1>, <_id 2>, <_id 3>, <_id 4>, <_id 5>
    ]
  }
})
```

Amazon DocumentDB engine version support dates

You can use the following dates to plan your testing and upgrade cycles.

Release calendar for Amazon DocumentDB major versions

Amazon DocumentDB currently supports the following major versions. To upgrade your engine version, see Upgrading a major engine version or Upgrading using AWS DMS.

Engine version	Release date	End of standard support	Start of Extended Support (year 1 pricing)	Start of Extended Support (year 3 pricing)	End of Extended Support
Version 3.6	9 January 2019	30 March 2026	31 March 2026	31 March 2028	30 March 2029
Version 4.0	9 November 2020	N/A	N/A	N/A	N/A

Engine version	Release date	End of standard support	Start of Extended Support (year 1 pricing)	Start of Extended Support (year 3 pricing)	End of Extended Support
Version 5.0	1 March 2023	N/A	N/A	N/A	N/A
Version 8.0	14 November 2025	N/A	N/A	N/A	N/A

Amazon DocumentDB supports multiple engine versions under standard support. You can continue running a version past its end of standard support date for an Extended Support fee. For more information, see Amazon DocumentDB Extended Support and Amazon DocumentDB pricing.

Release calendar for Amazon DocumentDB minor versions

Amazon DocumentDB currently supports the following minor versions. The release schedule will vary depending on additional features or fixes. Minor versions can reach end of standard support before corresponding major versions do. To upgrade your engine version, see Upgrading a minor engine version.

> **Note**
>
> Minor versions are available starting on major version 5.0.

Engine version	Release date
Version 5.0.0 (LTS)	16 February 2026
Version 5.0.1	8 June 2026

Amazon DocumentDB designates certain versions as Long-Term Support (LTS) releases. For more information, see Using a long-term support (LTS) release.

Amazon DocumentDB in-place major version upgrade

You can perform an in-place major version upgrade (MVU) of your Amazon DocumentDB cluster while keeping the same endpoints, storage, and tags. Your applications continue to work without modifications. This feature is available at no additional cost in all AWS Regions where Amazon DocumentDB is available.

Important

Your cluster is unavailable during the upgrade and reboots multiple times. Do not connect to, read from, or write to the cluster after starting the upgrade. Downtime varies depending on the number of collections, indexes, databases, and instances. Perform the upgrade during your maintenance window or low-utilization hours.
Once upgraded, you cannot downgrade to a previous version. You can restore your pre-upgrade snapshot to a new cluster if needed.

Topics

- Supported upgrade paths
- Prerequisites
- Best practices
- What changes after upgrading to Amazon DocumentDB 8.0
- Post-upgrade considerations for clusters upgraded from 3.6 or 4.0
- Performing the upgrade
- Troubleshooting

Supported upgrade paths

The following table lists the target engine versions you can select for each source version. There is no requirement to first upgrade to the `.0` minor version and then apply a minor version upgrade.

Source version	Supported target versions
3.6	Any published 5.0 or 8.0 minor version
4.0	Any published 5.0 or 8.0 minor version

Source version	Supported target versions
5.0.0	8.0.0, 8.0.1, 8.0.2
5.0.1	8.0.0, 8.0.1, 8.0.2
5.0.2	8.0.2

For post-upgrade considerations, see the section called "Post-upgrade considerations for clusters upgraded from 3.6 or 4.0" and the section called "What changes after upgrading to Amazon DocumentDB 8.0".

To view the minor versions available in your AWS Region, use the AWS CLI command `aws docdb describe-db-engine-versions`. For a list of released minor versions, see Release notes.

Note

In-place MVU is not supported for global clusters or elastic clusters. To upgrade a global cluster, remove the secondary clusters, convert the primary to a regional cluster, perform the MVU, then recreate the global cluster by adding secondary clusters using the same names to retain your endpoints. You will incur I/O charges while the upgraded primary replicates data to the new secondaries. For detailed steps, see Removing a cluster from an Amazon DocumentDB global cluster.

Prerequisites

Important

Scale up burstable instances before upgrading. If your cluster uses burstable instance types (for example, `db.t3.medium` or `db.t4g.medium`), scale up the primary instance to at least `db.r5.large` or `db.r6g.large` before initiating the upgrade. Burstable instances may not have sufficient CPU and memory to complete the upgrade process, which can result in upgrade failures and extended cluster unavailability. You can scale back down after the upgrade completes.

Scale up the serverless writer instance before upgrading. If your cluster has a serverless writer instance, the maximum DCU in the `ServerlessV2ScalingConfiguration` must

be set to 10 or higher before initiating the upgrade. Clusters with a maximum DCU below 10 cannot be upgraded because they do not have sufficient CPU or memory to complete the upgrade.

Check your partial indexes before upgrading (upgrade from Amazon DocumentDB 5.0 to 8.0). Before starting the upgrade, review your partial indexes for a `partialFilterExpression` that uses the `$type` operator to match certain BSON types: specifically the `binData`, `bool`, `date`, `double`, `decimal`, and `array` type aliases, or a numeric type code such as `{ "$type" : 1 }`. Such partial indexes cause the upgrade to fail, even though the index works normally for reads and writes. Use `db.collection.getIndexes()` to review the `partialFilterExpression` of your partial indexes. Drop the affected partial index before starting the upgrade and recreate it after the upgrade completes. If you are unsure whether an index is affected, contact AWS support before upgrading.

- **Instance type** — Amazon DocumentDB 4.0+ does not support db.r4 instances. Modify any `db.r4.*` instances to `db.r5.*` instances or newer before upgrading. See Modifying an Amazon DocumentDB instance and Supported instance classes by Region.
- **OS patches** — Apply any pending OS maintenance actions on all instances before upgrading. See Amazon DocumentDB operating system updates.

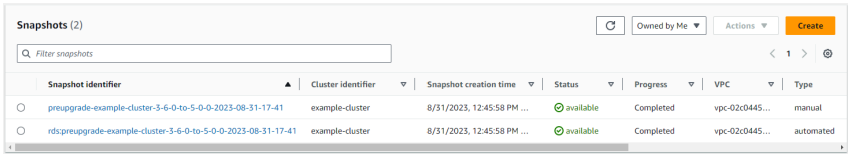
> **Note**
>
> Pending cluster-level engine patches may hide instance OS patches. Apply engine patches first if needed. See Performing a patch update to a cluster's engine version.

- **Index limits on burstable instances (t-family instances)** — If you have more than 3,000 indexes on burstable instances, scale up the primary to at least db.r5.large before upgrading. You can scale back down after the upgrade completes.

Instance	Maximum indexes for MVU
db.t4g.medium	3K
db.t3.medium	10K

- **Parameter group** — Have a custom cluster parameter group for the target version ready before upgrading. If one is not specified, the default parameter group for the target version will be used (for example, `default.docdb5.0` or `default.docdb8.0`).
- **Manual snapshot** — Create a manual snapshot before upgrading. The upgrade process creates an automatic snapshot named `preupgrade-<name>-<version>-<timestamp>`, but always create your own backup. See Creating a manual cluster snapshot.

> **Note**
>
> The auto snapshot created by the upgrade process will not be automatically deleted after the in-place major version upgrade has completed. This snapshot will not incur any charges as long as it is within the retention period. You can choose to delete this snapshot once you have verified a successful upgrade of your cluster.



Best practices

Before upgrade — testing with a clone

1. Use Cloning a volume for an Amazon DocumentDB cluster to create a clone of your cluster. You will not incur storage costs unless you modify data on the clone.
2. Match the instance count of the clone to the target cluster for a realistic time estimate.
3. Perform the MVU on the clone and fully test for functional differences.
4. Check if an upgrade is already scheduled by running `describe-db-clusters` and looking for `PendingModifiedValues.EngineVersion`. If you have modified the cluster and selected to apply it in the next maintenance window, the schedule will not be visible in the console but you can view it in the AWS CLI:

```
aws docdb describe-db-clusters \
  --region us-east-1 \
  --db-cluster-identifier mydocdbcluster
```

```
"PendingModifiedValues": {
    "EngineVersion": "5.0.0"
},
```

5. If testing is successful, proceed with the upgrade on your production cluster.

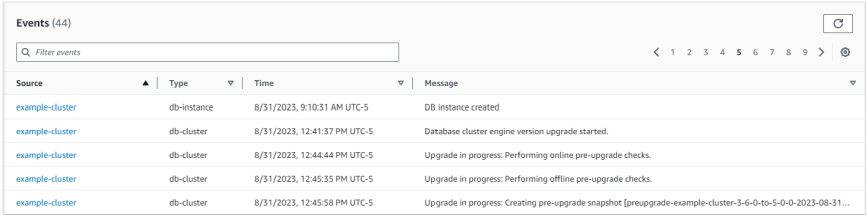
During the upgrade

You can monitor progress of your in-place major version upgrade by subscribing to cluster maintenance events. When the upgrade completes, you will receive the "Database cluster major version has been upgraded" event. This and other events occurring during the upgrade appear in the **Events and Tags** section of the cluster detail page in the Amazon DocumentDB console. The cluster status then changes from `upgrading` to `available`.

The following events are generated during the upgrade:

1. Database cluster engine major version upgrade started. Cluster remains online.
2. Upgrade preparation in progress: Starting online upgrade prechecks.
3. Upgrade preparation in progress: Completed online upgrade prechecks.
4. Taking database cluster offline while the primary instance completes the patch/upgrade process.
5. Upgrade preparation in progress: Starting offline upgrade prechecks.
6. Upgrade preparation in progress: Completed offline upgrade prechecks.
7. Upgrade in progress: Creating pre-upgrade snapshot [preupgrade-<cluster-name>-<version-from>-to-<version-to>-<timestamp>].
8. Upgrade in progress: Cloning volume.
9. Upgrade in progress: Upgrading writer.
10. Upgrade in progress: Upgrading readers.
11. Database cluster engine major version has been upgraded.

Events are visible in the console under the **Events** page:



From the AWS CLI, you can run `aws docdb describe-events` to monitor upgrade progress. To receive notifications automatically, use `aws docdb create-event-subscription` to subscribe to events and route them to Amazon SNS for email, push messages, or other delivery methods. For more information, see Subscribing to Amazon DocumentDB events.

```
aws docdb describe-events
  --source-identifier mydocdbcluster
  --source-type db-cluster
```

The command returns output similar to the following:

```
{
    "Events": [
        {
            "SourceIdentifier": "mydocdbcluster",
            "SourceType": "db-cluster",
            "Message": "Database cluster engine version upgrade started.",
            "EventCategories": [
                "maintenance"
            ],
            "Date": "2023-07-11T23:20:32.444000+00:00",
            "SourceArn": "arn:aws:rds:us-east-1:xxxx:cluster:mycluster"
        }
    ]
}
```

After the upgrade

Warning

Immediately after the in-place major version upgrade, your Amazon DocumentDB cluster repopulates index metadata that the database engine uses to optimize query execution plans. Query performance returns to expected levels once this process completes. It typically finishes in a few minutes but can take up to two hours depending on the number of indexes on your cluster.

Do not reboot, failover, or scale up/down your writer instance during this time, as it may disrupt the index metadata recalculation. Wait until you observe expected query performance before making such changes.

Track progress via the following cluster events:

- Post-upgrade cluster status: Index metadata refresh process started
- Post-upgrade cluster status: Index metadata refresh process completed in X seconds

Contact AWS support if the index metadata refresh process hasn't completed within three hours, or if you continue to experience performance issues after the process completes.

1. **Take a manual snapshot** of the upgraded cluster in case you need to restore to the post-upgrade state. The automatic snapshot process will resume as soon as the in-place major version upgrade completes.
2. **Tag clusters upgraded from 3.6.** Add a tag to differentiate clusters upgraded from 3.6. (see the section called "Post-upgrade considerations for clusters upgraded from 3.6 or 4.0").
3. **Update your driver.** To use new features (for example, collation, views, or Zstd compression in 8.0), upgrade to the corresponding MongoDB API version. For more information, see What's new in Amazon DocumentDB 8.0.
4. **Test thoroughly.** Validate your application against the upgraded cluster.

What changes after upgrading to Amazon DocumentDB 8.0

After performing a major version upgrade to Amazon DocumentDB 8.0 (from Amazon DocumentDB 5.0, or directly from 3.6 or 4.0), the following features are enabled or changed:

- **Collation.** Amazon DocumentDB 8.0 supports collation. After the upgrade, new collections and their indexes, and new indexes on existing collections, have collation enabled by default.
- **Text index.** New text indexes are created using Text Index V2, which uses an updated text search parser for improved MongoDB compatibility. Existing text indexes are not affected.
- **Query planner version.** If you did not have a custom parameter group, a new default parameter group is created for Amazon DocumentDB 8.0 with Planner Version 3 automatically selected. With query planner version 3, views are also available.
- **Compression.** Amazon DocumentDB 8.0 supports dictionary-based document compression using the Zstd algorithm. After the upgrade, new collections are created with Zstd compression enabled by default. Existing collections from 5.0 retain their compression settings. To take advantage of Zstd compression on existing collections, you can modify their compression

settings. For more information, see Managing dictionary-based compression in Amazon DocumentDB 8.0.

- **Index rebuild.** If you are upgrading from Amazon DocumentDB 5.0 to Amazon DocumentDB 8.0, no index rebuild is needed. If you are upgrading directly from Amazon DocumentDB 3.6 or 4.0 to 8.0, rebuild your indexes as described in the section called "Post-upgrade considerations for clusters upgraded from 3.6 or 4.0".

> **Important**
>
> Amazon DocumentDB 8.0 requires TLS 1.2 or higher. TLS 1.0 and TLS 1.1 are no longer supported.

> **Note**
>
> For a full list of functional differences, see Amazon DocumentDB compatibility with MongoDB.

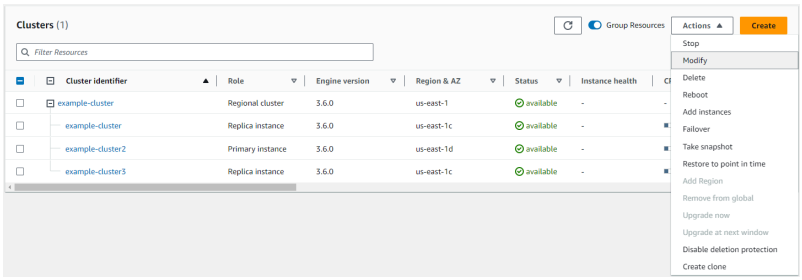
Post-upgrade considerations for clusters upgraded from 3.6 or 4.0

- **Index rebuild.** An MVU retains original indexes. Amazon DocumentDB 5.0 and later have improved index maintenance and garbage collection, especially for low-cardinality indexes. After upgrading from 3.6 or 4.0 (to either 5.0 or 8.0), rebuild your indexes to ensure optimal query performance (optional, involves additional I/O). See Index maintenance using `reIndex`.
- **Subdocument numeric comparison (3.6 only).** Clusters upgraded from 3.6 inherit the 3.6 behavior where numeric types in subdocuments are not compared across types. For example, `{a: {b: NumberLong(1)}}` does not equal `{a: {b: 1}}` in 3.6, but they are equal in 4.0 and later. This behavior affects any clusters upgraded from 3.6.

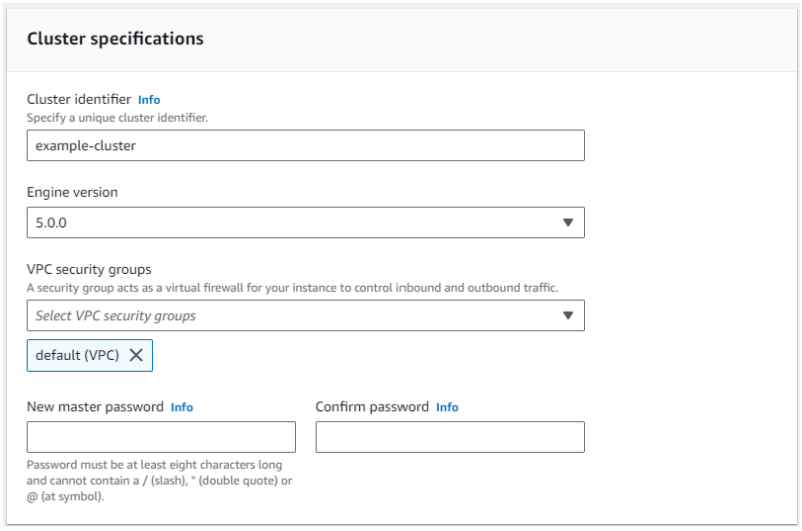
Performing the upgrade

Using the AWS Management Console

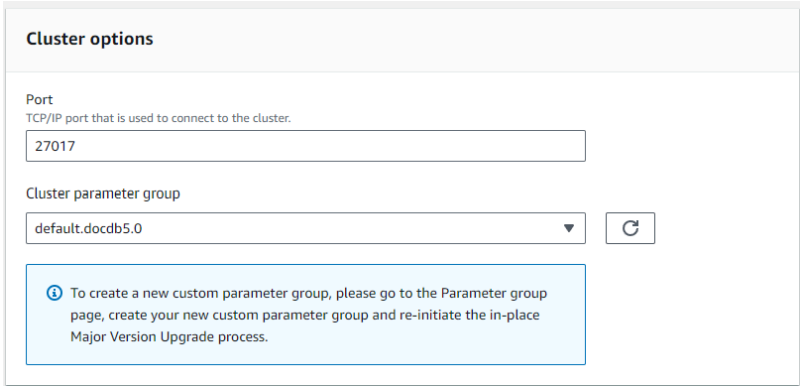
1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the **Clusters** table, select the source cluster, choose **Actions**, and then **Modify**.



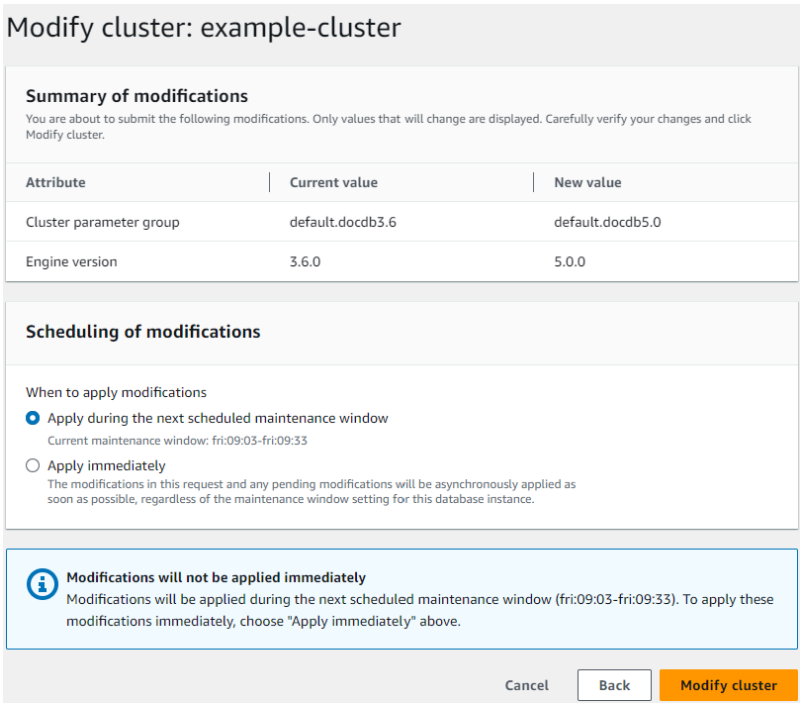
3. In **Cluster specifications**, choose the target version (for example, **5.0.0** or **8.0.0**) from the **Engine version** dropdown.



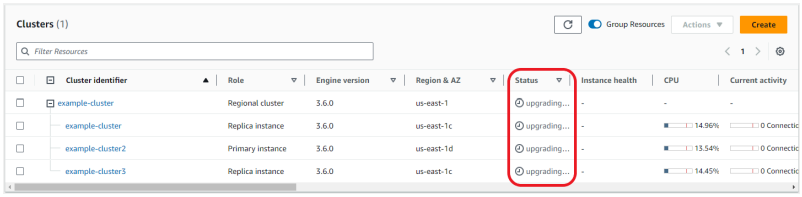
4. In **Cluster options**, select your cluster parameter group for the target engine version. You can use the default (for example, **default.docdb5.0** or **default.docdb8.0**) or a custom parameter group you created.



5. Choose **Continue**, select your scheduling preference (apply immediately or next maintenance window), then choose **Modify cluster**.



6. Monitor the cluster status in the clusters table as it changes to **upgrading**:



Using the AWS CLI

Use [modify-db-cluster](#) with the `--allow-major-version-upgrade` flag:

```
aws docdb modify-db-cluster \
  --db-cluster-identifier mydocdbcluster \
  --allow-major-version-upgrade \
  --engine-version 8.0.0 \
  --apply-immediately \
  --cluster-parameter-group mydocdbparametergroup \
  --region us-east-1
```

Replace each *placeholder* with your cluster's information.

Troubleshooting

- **Pre-upgrade check failure.** Before the upgrade begins, Amazon DocumentDB runs pre-upgrade validation checks. The following are common causes of pre-check failures:
  - **The upgrade could not proceed because the index count exceeds the limit for the instance type** — Upgrade to a larger instance type before retrying the upgrade. For index limits by instance type, see Prerequisites.
- **Upgrade failure and rollback.** If the upgrade fails, it automatically attempts a rollback. A successful rollback generates the event: "Database cluster is in a state that cannot be upgraded." Your cluster returns to its pre-upgrade state and you can continue using it. Contact AWS support to troubleshoot before re-attempting.
- **Post-upgrade performance.** Temporary performance degradation and high CPU utilization may occur while the index metadata refresh runs. If degradation persists beyond 3 hours, contact AWS support.

For additional assistance, contact AWS Support.

Amazon DocumentDB minor version upgrade

> **Note**
>
> This topic covers *minor* version upgrades within the same major version (for example, upgrading from 5.0.0 to 5.0.1). For information about upgrading to a different major version, see Amazon DocumentDB in-place major version upgrade.

You can upgrade the minor engine version of your Amazon DocumentDB cluster (for example, from 5.0.0 to 5.0.1) by modifying the cluster. Minor version upgrades can be applied immediately or during the next maintenance window.

Before performing a minor version upgrade

Before performing a minor version upgrade, the following actions may be helpful:

- Perform the upgrade during a period of low traffic to reduce the impact of any brief downtime.
- Verify the target minor version is a valid upgrade target for your current version by using the `describe-db-engine-versions` command.

- For global clusters, secondary clusters must be upgraded before the primary cluster. For more information, see Global clusters patching.

> **Important**
>
> - Upgrades are **one-way operations** — you cannot downgrade after upgrading.
> - Your cluster will experience brief downtime during the upgrade process.

Performing a minor version upgrade

Using the AWS Management Console

To perform a minor version upgrade using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the **Clusters** table, select the cluster you want to upgrade, choose **Actions**, and then choose **Modify**.
3. On the **Modify cluster** dialog, in the **Cluster specifications** section, choose the target minor version from the **Engine version** drop-down menu.
4. Scroll down and choose **Continue**.
5. In the **Scheduling of modifications** section, choose to apply the change immediately or during the next maintenance window.

   Then choose **Modify cluster**.

Using the AWS CLI

Use the `modify-db-cluster` command with the `--engine-version` parameter:

```
aws docdb modify-db-cluster \
    --db-cluster-identifier mydocdbcluster \
    --engine-version 5.0.1 \
    --apply-immediately \
    --region us-east-1
```

In the example above, replace each *user input placeholder* with your cluster's information.

> **Note**
>
> To apply the upgrade during the next maintenance window instead of immediately, use `--no-apply-immediately` in place of `--apply-immediately`.

Verifying the minor version upgrade

After the upgrade completes, verify the engine version using the AWS CLI:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier mydocdbcluster \
    --query 'DBClusters[0].EngineVersion'
```

Output from this operation looks something like the following:

```
"5.0.1"
```

Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service

> **Important**
>
> Amazon DocumentDB does not follow the same support lifecycles as MongoDB and MongoDB's end-of-life schedule does not apply to Amazon DocumentDB.

You can upgrade your Amazon DocumentDB cluster to a higher version with minimal downtime using AWS DMS. AWS DMS is a fully managed service that makes it easy to migrate from older Amazon DocumentDB versions, relational databases and non-relational databases to your target Amazon DocumentDB cluster.

Topics

- Step 1: Enable change streams
- Step 2: Modify the change streams retention duration
- Step 3: Migrate your indexes
- Step 4: Create an AWS DMS replication instance
- Step 5: Create an AWS DMS source endpoint
- Step 6: Create an AWS DMS target endpoint
- Step 7: Create and run a migration task
- Step 8: Changing the application endpoint to the target Amazon DocumentDB cluster

Step 1: Enable change streams

To perform a minimal downtime migration, AWS DMS requires access to the cluster's change streams. Amazon DocumentDB change streams provide a time-ordered sequence of update events that occur within your cluster's collections and databases. Reading from the change stream enables AWS DMS to perform change data capture (CDC) and apply incremental updates to the target Amazon DocumentDB cluster.

To enable change streams for all collections on a specific database, authenticate to your Amazon DocumentDB cluster using the mongo shell and execute the following commands:

```
db.adminCommand({modifyChangeStreams: 1,
    database: "db_name",
    collection: "",
    enable: true});
```

Step 2: Modify the change streams retention duration

Next, modify the change stream retention period based on how long you would like to retain change events in the change stream. For example, if you expect your Amazon DocumentDB cluster migration using AWS DMS to take 12 hours, you should set the change stream retention to a value greater than 12 hours. The default retention period for your Amazon DocumentDB cluster is three hours. You can modify the change stream log retention duration for your Amazon DocumentDB cluster to be between one hour and seven days using the AWS Management Console or the AWS CLI. For more information, see Modifying the change stream log retention duration.

Step 3: Migrate your indexes

Create the same indexes on your target Amazon DocumentDB cluster that you have on your source Amazon DocumentDB cluster. Although AWS DMS handles the migration of data, it does not migrate indexes. To migrate the indexes, use the Amazon DocumentDB Index Tool to export indexes from the source Amazon DocumentDB cluster. You can get the tool by creating a clone of the Amazon DocumentDB tools GitHub repo and following the instructions in README.md. You can run the tool from an Amazon EC2 instance or an AWS Cloud9 environment running in the same Amazon VPC as your Amazon DocumentDB cluster.

In the following example, replace each *user input placeholder* with your own information.

The following code dumps indexes from your source Amazon DocumentDB cluster:

```
python migrationtools/documentdb_index_tool.py --dump-indexes
--uri mongodb://sample-user:user-password@sample-source-cluster.node.us-
east 1.docdb.amazonaws.com:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false'
--dir ~/index.js/

2020-02-11 21:51:23,245: Successfully authenticated to database: admin2020-02-11
 21:46:50,432: Successfully connected to instance docdb-40-xx.cluster-xxxxxxxx.us-
east-1.docdb.amazonaws.com:27017
2020-02-11 21:46:50,432: Retrieving indexes from server...2020-02-11 21:46:50,440:
 Completed writing index metadata to local folder: /home/ec2-user/index.js/
```

Once your indexes are successfully exported, restore those indexes in your target Amazon DocumentDB cluster. To restore the indexes that you exported in the preceding step, use the Amazon DocumentDB Index Tool. The following command restores the indexes in your target Amazon DocumentDB cluster from the specified directory.

```
python migrationtools/documentdb_index_tool.py --restore-indexes
--uri mongodb://sample-user:user-password@sample-destination-
cluster.node.us-east 1.docdb.amazonaws.com:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false'
--dir ~/index.js/

2020-02-11 21:51:23,245: Successfully authenticated to database: admin2020-02-11
 21:51:23,245: Successfully connected to instance docdb-50-xx.cluster-xxxxxxxx.us-
east-1.docdb.amazonaws.com:27017
2020-02-11 21:51:23,264: testdb.coll: added index: _id
```

To confirm that you restored the indexes correctly, connect to your target Amazon DocumentDB cluster with the mongo shell and list the indexes for a given collection. See the following code:

```
mongo --ssl
--host docdb-xx-xx.cluster-xxxxxxxx.us-east-1.docdb.amazonaws.com:27017
--sslCAFile rds-ca-2019-root.pem --username documentdb --password documentdb

db.coll.getIndexes()
```

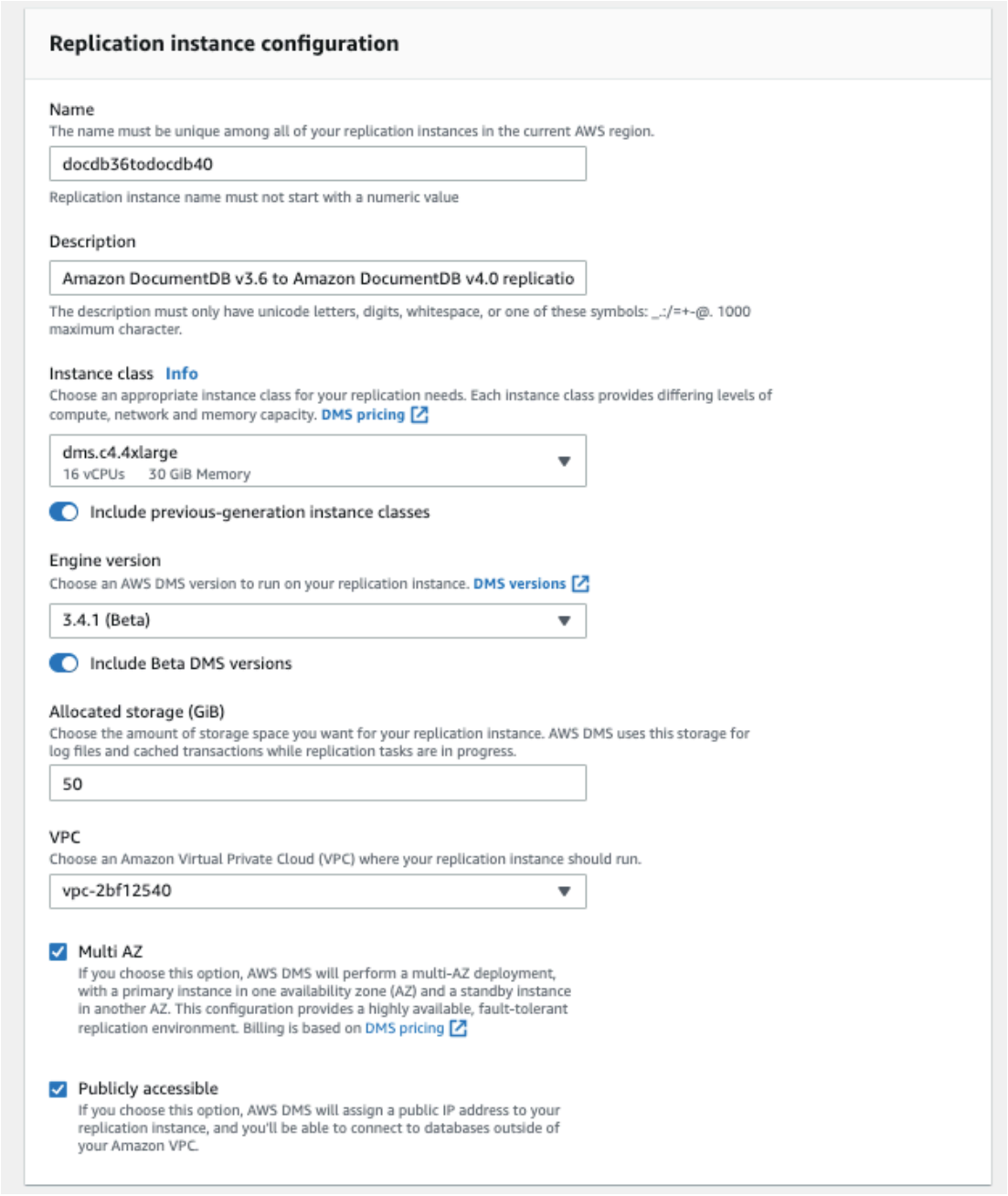
Step 4: Create an AWS DMS replication instance

An AWS DMS replication instance connects and reads data from your source Amazon DocumentDB cluster and writes it your target Amazon DocumentDB cluster. The AWS DMS replication instance can perform both bulk load and CDC operations. Most of this processing happen in memory. However, large operations might require some buffering on disk. Cached transactions and log files are also written to disk. Once the data is migrated, the replication instance also streams any change events to make sure the source and target are in sync.

To create an AWS DMS replication instance:

1. Open the AWS DMS console.
2. In the navigation pane, choose **Replication instances**.
3. Choose **Create replication instance** and enter the following information:

   - For Name, enter a name of your choice. For example, `docdb36todocdb40`.
   - For **Description**, enter a description of your choice. For listitem, Amazon DocumentDB 3.6 to Amazon DocumentDB 4.0 replication instance.
   - For **Instance class**, choose the size based on your needs.
   - For **Engine version**, choose `3.4.1`.
   - For **Amazon VPC**, choose the Amazon VPC that houses your source and target Amazon DocumentDB clusters.
   - For **Allocated storage** (GiB), use the default of 50 GiB. If you have a high write throughput workload, increase this value to match your workload.
   - For **Multi-AZ**, choose **Yes** if you need high availability and failover support.
   - For **Publicly accessible**, enable this option.



4. Choose **Create replication instance**.

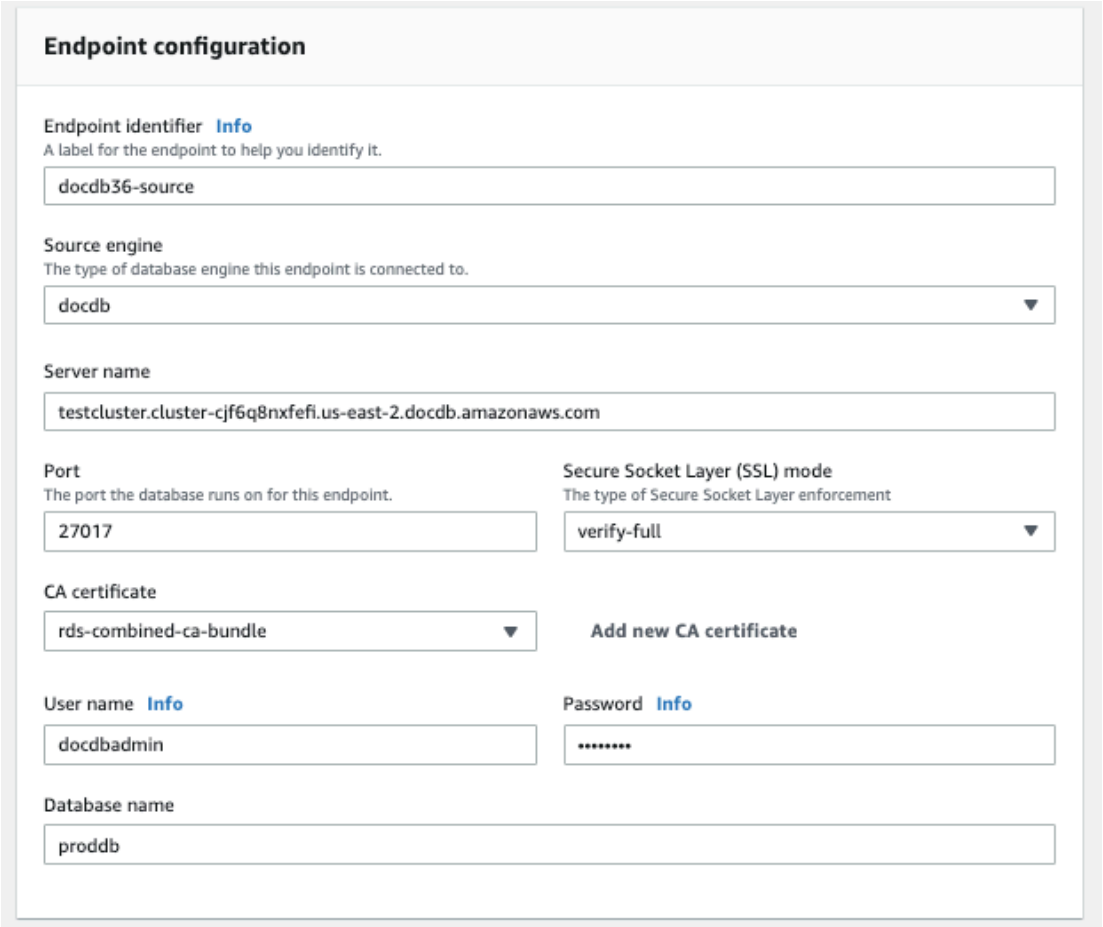
Step 5: Create an AWS DMS source endpoint

The source endpoint is used for the source Amazon DocumentDB cluster.

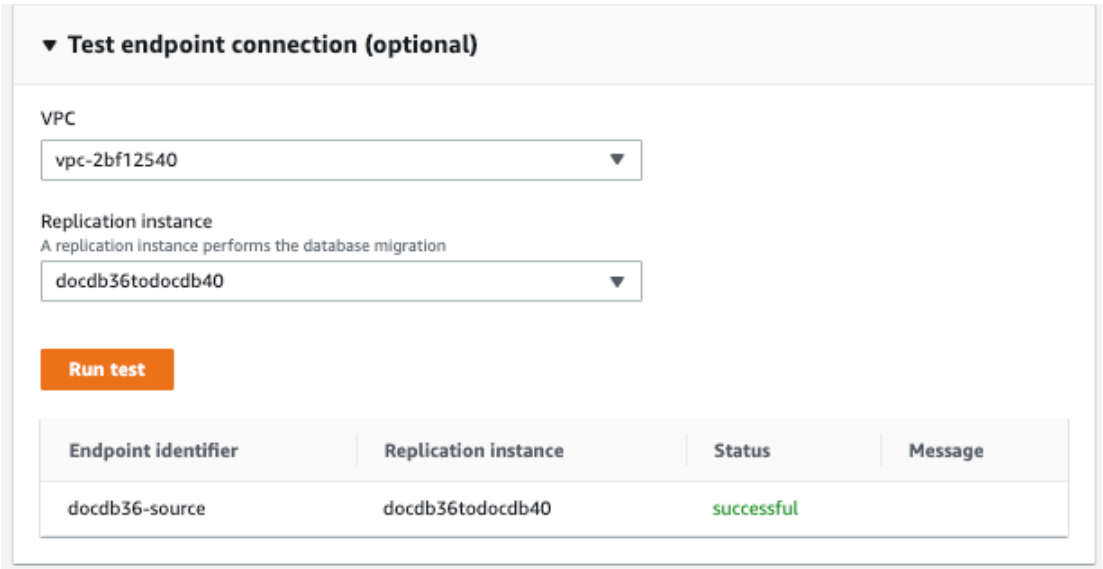
To create a source endpoint

1. Open the AWS DMS console.
2. In the navigation pane, choose **Endpoints**.
3. Choose `Create endpoint` and enter the following information:

   - For **Endpoint type**, choose **Source**.
   - >For **Endpoint identifier**, enter a name that's easy to remember, for example `docdb-source`.
   - For **Source engine**, choose `docdb`.
   - For **Server name**, enter the DNS name of your source Amazon DocumentDB cluster.
   - For **Port**, enter the port number of your source Amazon DocumentDB cluster.
   - For **SSL mode**, choose `verify-full`.
   - For **CA certificate**, choose **Add new CA certificate**. Download the new CA certificate to create TLS connections bundle. For **Certificate identifier**, enter `rds-combined-ca-bundle`. For **Import certificate file**, choose **Choose file** and navigate to the `.pem` file that you previously downloaded. Select and open the file. Choose **Import certificate**, then choose `rds-combined-ca-bundle` from the **Choose a certificate** drop down
   - For **User name**, enter the primary username of your source Amazon DocumentDB cluster.
   - For **Password**, enter the primary password of your source Amazon DocumentDB cluster.
   - For **Database name**, enter the database name you are looking to upgrade.



4. Test your connection to verify it was successfully setup.



5. Choose **Create Endpoint**.

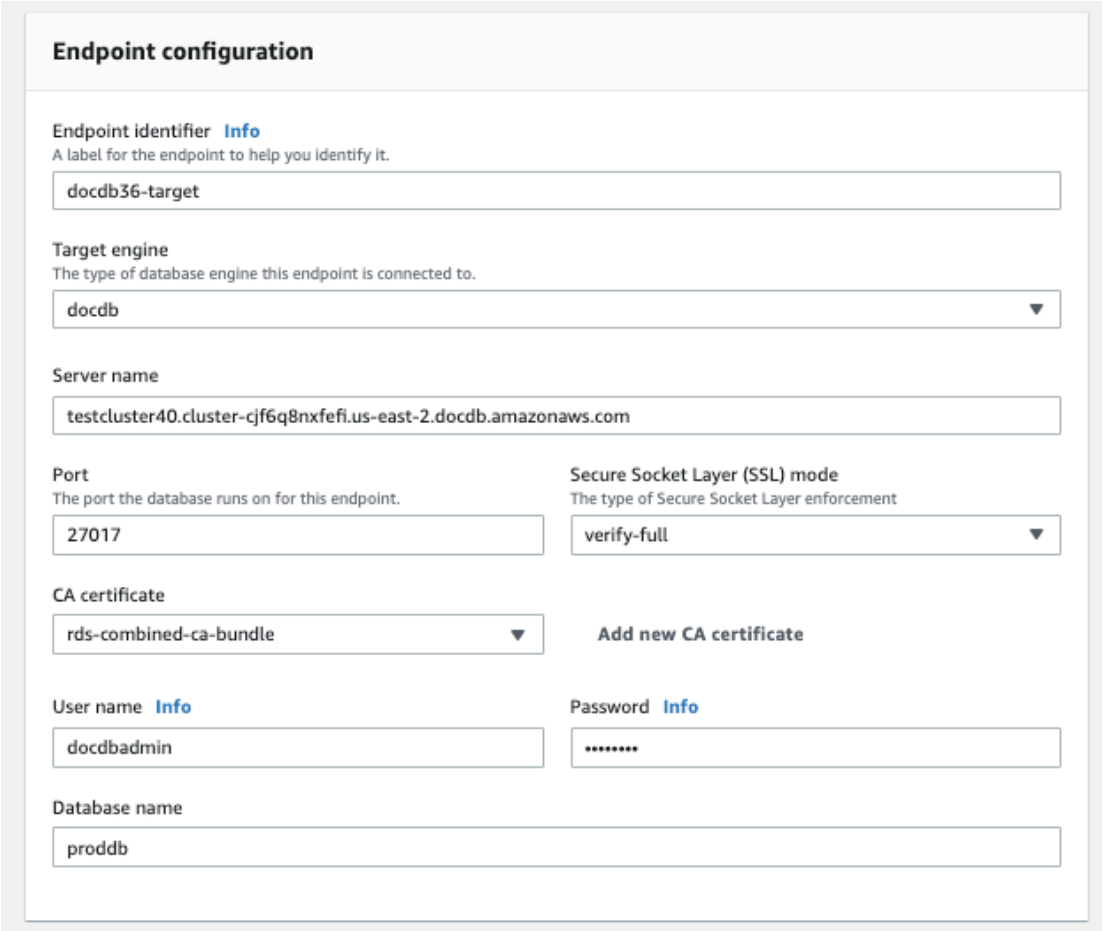
> **Note**
>
> AWS DMS can only migrate one database at a time.

Step 6: Create an AWS DMS target endpoint

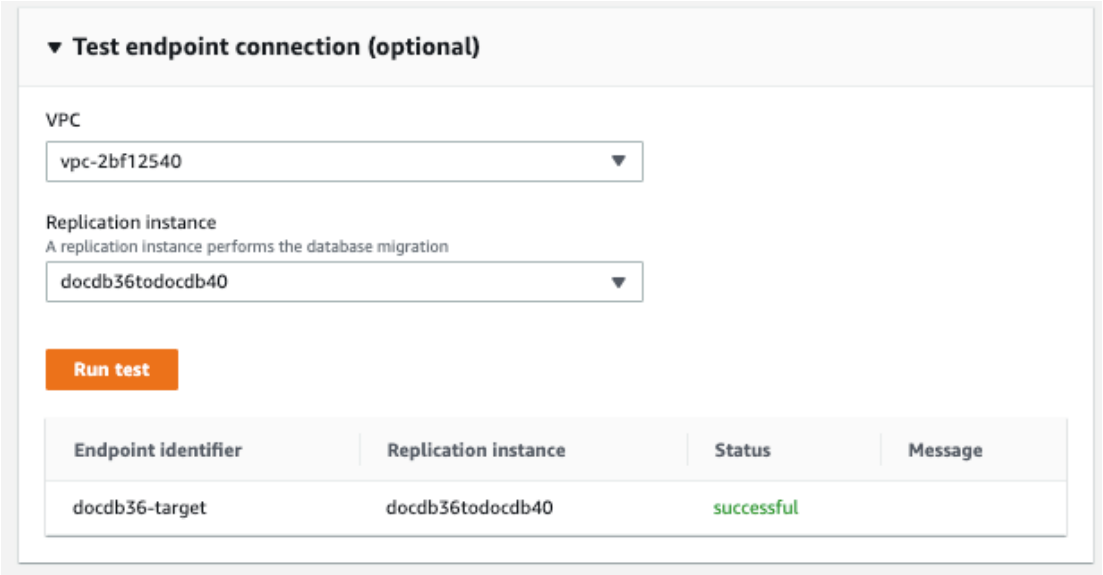
The target endpoint is for your target Amazon DocumentDB cluster.

To create a target endpoint:

1. Open the AWS DMS console.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create endpoint** and enter the following information:
   - For **Endpoint type**, choose **Target**.
   - For **Endpoint identifier**, enter a name that's easy to remember, for example `docdb-target`.
   - For **Source engine**, choose `docdb`.
   - For **Server name**, enter the DNS name of your target Amazon DocumentDB cluster.
   - For **Port**, enter the port number of your target Amazon DocumentDB cluster.
   - For **SSL mode**, choose `verify-full`.
   - For **CA certificate**, choose the existing `rds-combined-ca-bundle` certificate from the **Choose a certificate** drop down.
   - For **User name**, enter the primary username of your target Amazon DocumentDB cluster.
   - For **Password**, enter the primary password of your target Amazon DocumentDB cluster.
   - For **Database name**, enter the same database name you used to setup your source endpoint.



4. Test your connection to verify it was successfully set up.



5. Choose **Create Endpoint**.

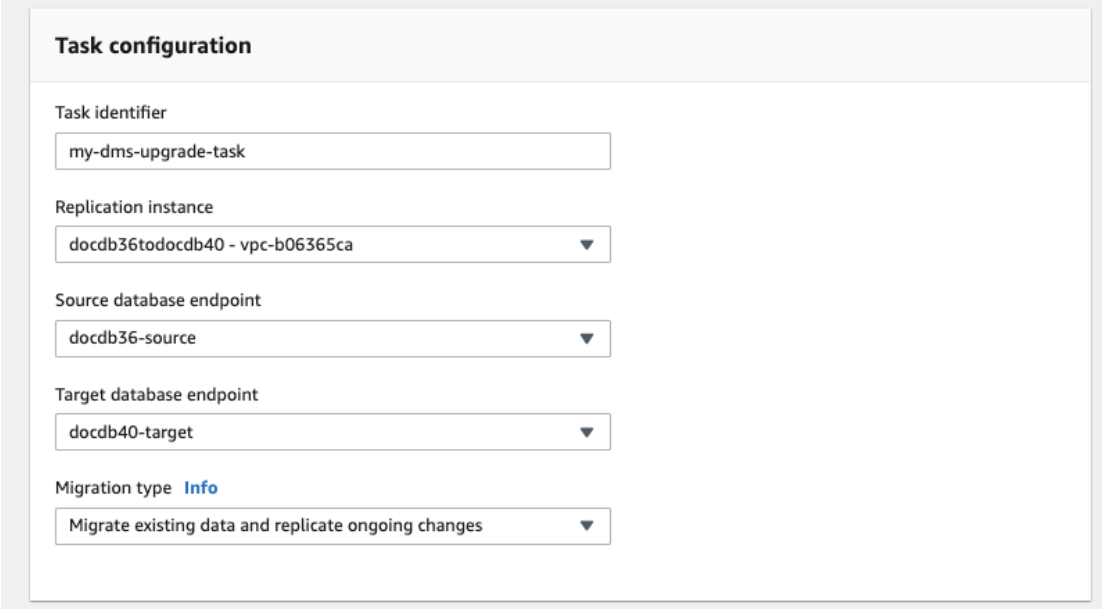
Step 7: Create and run a migration task

An AWS DMS task binds the replication instance with your source and target instance. When you create a migration task, you specify the source endpoint, target endpoint, replication instance, and any desired migration settings. An AWS DMS task can be created with three different migration types - migrate existing data, migrate existing data, and replicate ongoing changes or replicate data changes only. Since the purpose of this walk-through is to upgrade an Amazon DocumentDB cluster with minimal downtime, the steps utilize the option to migrate existing data and replicate ongoing changes. With this option, AWS DMS captures changes while migrating your existing data. AWS DMS continues to capture and apply changes even after the bulk data has been loaded. Eventually the source and target databases will be in sync, allowing for a minimal downtime migration.

Below are the steps to create a migration task for a minimal downtime migration:

1. Open the AWS DMS console.
2. In the navigation pane, choose **Database migration tasks**.
3. Choose **Create database migration task** and enter the following information in the **Task configuration** section:

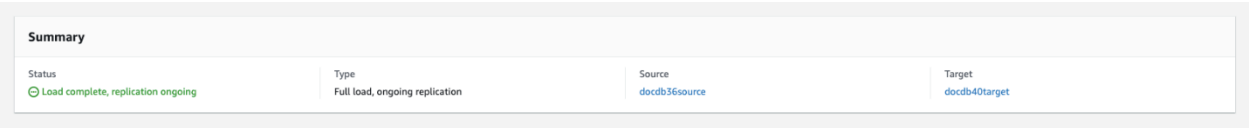
   - For **Task identifier**, enter a name that's easy to remember, for example `my-dms-upgrade-task`.
   - For **Descriptive Amazon Resource Name (ARN)**, enter a user-friendly name to override the default DMS ARN.
   - For **Replication instance**, choose the replication instance that you created in Step 4: Create an AWS DMS replication instance.
   - For **Source database endpoint**, choose the source endpoint that you created in Step 5: Create an AWS DMS source endpoint.
   - For **Target database endpoint**, choose the target endpoint that you created in Step 6: Create an AWS DMS target endpoint.
   - For **Migration type**, choose **Migrate and replicate**.



4. Enter the following information in the **Task settings** section:

   - For **Target table preparation mode** section, choose **Do nothing**. This will ensure that the indexes created in step 3 will not be dropped.
   - For **Task logs** sub-section, select **Turn on CloudWatch logs**.
   - For **Migration task startup configuration**, choose **Automatically on create**. This will start the migration task automatically once you create it.
   - Choose **Create database migration task**.

AWS DMS now begins migrating data from your source Amazon DocumentDB cluster to your target Amazon DocumentDB cluster. The task status should change from Starting to Running. You can monitor the progress by choosing Tasks in the AWS DMS console. After several minutes/ hours (depending on the size of your migration), the status should change from to Load complete, replication ongoing. This means that AWS DMS has completed a full load migration of your source Amazon DocumentDB cluster to a target Amazon DocumentDB cluster and is now replicating change events.



Eventually your source and target will be in sync. You can verify whether they are in sync by running a `count()` operation on your collections to verify all change events have migrated.

Step 8: Changing the application endpoint to the target Amazon DocumentDB cluster

After the full load is complete and the CDC process is replicating continuously, you are ready to change your application's database connection endpoint from your source Amazon DocumentDB cluster to your target Amazon DocumentDB cluster.

Using a long-term support (LTS) release

Amazon DocumentDB designates certain versions as Long-Term Support (LTS) releases. Database clusters that use LTS can stay on the same version longer and undergo fewer upgrade cycles than clusters that use non-LTS releases. LTS versions will be patched only in the event of critical stability and security patches without introducing new features. Additional optional patches may be released such as metrics to debug critical failures. These will be cumulatively applied with required patches.

Current LTS Release

The current LTS release for Amazon DocumentDB is:

- Minor Version 5.0.0

To determine the version your cluster is on, see the section called "How to check if my existing engine version 5.0.0 cluster is on LTS".

For more information about support timelines and release cycles for the LTS versions, see Amazon DocumentDB engine version support dates. Amazon DocumentDB will announce the next LTS version through standard communication channels.

> **Note**
>
> An LTS release for Amazon DocumentDB 8.0 has not yet been designated. If you require LTS, use engine version 5.0.0 with Patch Version 3.0.17983 or later.

Who should use LTS

Amazon DocumentDB LTS releases are designed for clusters with limited upgrade windows. LTS is ideal for production environments where database availability directly impacts business operations.

Consider LTS releases if:

- Your application has strict uptime requirements that limit upgrade opportunities
- You have all required database engine features and enhancements your application needs

Consider standard releases instead if:

- Your application can handle minimal interruption
- You want the latest capabilities and enhancements as soon as they're released

> ⚠ **Important**
>
> - Upgrades are **one-way operations** - you cannot downgrade after upgrading
> - Your cluster will experience brief downtime during the upgrade process

How to create a new LTS cluster

To create an LTS cluster, follow these steps:

1. In the Management Console, create a new engine version 5.0.0 cluster
2. Verify you're on engine version 5.0.0 after cluster creation using the AWS CLI:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier mydocdbcluster \
    --query 'DBClusters[0].EngineVersion'
```

Output from this operation looks something like the following:

```
"5.0.0"
```

How to upgrade from engine version 3.6.0 or 4.0.0 to a 5.0.0 LTS cluster

To upgrade to an LTS cluster, follow these steps:

1. In the Management Console, major version upgrade (MVU) your engine version 3.6.0 or 4.0.0 cluster to 5.0.0. For instructions on performing an MVU, see Amazon DocumentDB in-place major version upgrade.
2. Verify you're on engine version 5.0.0 after upgrade using the AWS CLI:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier mydocdbcluster \
    --query 'DBClusters[0].EngineVersion'
```

How to check if my existing engine version 5.0.0 cluster is on LTS

To check if your engine version 5.0.0 cluster is running an LTS version:

1. Verify you're on engine version 5.0.0 using the AWS CLI:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier mydocdbcluster \
    --query 'DBClusters[0].EngineVersion'
```

2. If needed, upgrade your existing engine version 5.0.0 cluster. For instructions on performing a patch update, see Performing a patch update to a cluster's engine version.

Amazon DocumentDB Extended Support

Amazon DocumentDB Extended Support allows you to continue running an engine version past the end of the standard support date for an additional cost. Your clusters are enrolled into Extended Support once past the end of the standard support date. If you do not complete upgrades by the end of the engine version standard support date, your clusters are billed at the Extended Support rate.

Topics

- Extended Support overview
- Extended Support charges

- Responsibilities

Amazon DocumentDB Extended Support provides the following updates and technical support:

- Security updates for critical and high CVEs for your instance or cluster
- Bug fixes and patches for critical issues
- The ability to open support cases and receive troubleshooting help within the standard Amazon DocumentDB service level agreement

This paid offering gives you more time to upgrade to a supported engine version. For example, Amazon DocumentDB end of standard support date for version 3.6 is March 30, 2026, however you aren't ready to upgrade to version 5.0 or 8.0 before that date. In this case, Amazon DocumentDB enrolls your version 3.6 cluster in Extended Support on March 31, 2026 so you can continue to use Amazon DocumentDB version 3.6. Starting July 1, 2026, AWS charges you an Extended Support premium for Amazon DocumentDB version 3.6 clusters.

Amazon DocumentDB Extended Support is available for up to 3 years past the engine version standard support date. After this time, if you haven't upgraded your engine version to a supported version, then Amazon DocumentDB will upgrade your engine version. Upgrade to a supported engine version as soon as possible.

Amazon DocumentDB version 3.6 end of standard support date and Extended Support dates:

- **Release of version 3.6** — 9 January 2019
- **End of version 3.6 standard support** — 30 March 2026
- **Start of version 3.6 Extended Support year 1 pricing** — 1 July 2026
- **Start of version 3.6 Extended Support year 3 pricing** — 31 March 2028
- **End of version 3.6 Extended Support** — 30 March 2029

Extended Support overview

Amazon DocumentDB upgrades your cluster to the last engine version released (before the end of the Extended Support date), if your cluster is not already on that version. The upgrade period will not occur until after the standard support date for that engine version.

You can create new clusters with an engine version that has reached the end of standard support date, if you already have a cluster on that engine version. Amazon DocumentDB enrolls these new clusters in Amazon DocumentDB Extended Support and charges you for this offering. If you upgrade to an engine version that's still under Amazon DocumentDB standard support before version 3.6's end of standard support date, you will not be charged an Extended Support premium.

You can end enrollment in Amazon DocumentDB Extended Support at any time. To end enrollment, upgrade each enrolled cluster to a newer engine version that is still under Amazon DocumentDB standard support. The end of Amazon DocumentDB Extended Support enrollment will be effective the day that you complete an upgrade to a newer engine version that is still under Amazon DocumentDB standard support.

For more information about Amazon DocumentDB end of standard support dates and Amazon DocumentDB end of Extended Support dates, see Amazon DocumentDB engine version support dates.

Extended Support charges

You will incur charges for all clusters running on Extended Support beginning July 1, 2026. For the Amazon DocumentDB version 3.6 end of standard support date, see Amazon DocumentDB engine version support dates.

The additional charge for Amazon DocumentDB Extended Support stops when you take one of the following actions:

- Upgrade to an engine version that's covered under standard support
- Delete the cluster that's running version 3.6 past the end of standard support date

For example, Amazon DocumentDB version 3.6 enters Extended Support on March 31, 2026, but charges do not start until July 1, 2026. If you upgrade your Amazon DocumentDB version 3.6 cluster to Amazon DocumentDB version 5.0 on July 31, 2026, you will only be charged for 30 days of Extended Support on Amazon DocumentDB version 3.6. For more information, see Amazon DocumentDB pricing.

Responsibilities

The following are the Amazon DocumentDB responsibilities and your responsibilities with Amazon DocumentDB Extended Support.

Amazon DocumentDB responsibilities

After the Amazon DocumentDB version 3.6 end of standard support date, Amazon DocumentDB will apply patches, bug fixes, and upgrades for version 3.6 clusters enrolled in Extended Support.

Your responsibilities

You are responsible for upgrading your engine to a newer engine version before Amazon DocumentDB version 3.6's end of Extended Support date. If you do not upgrade your cluster, after the Amazon DocumentDB version 3.6 end of Extended Support date, Amazon DocumentDB will attempt to upgrade your cluster to the latest engine version that's supported under Amazon DocumentDB standard support.

Amazon DocumentDB features and configurations

The following tables compare features, capabilities, and supported instance types across Amazon DocumentDB engine versions and elastic clusters. Use these tables to determine which engine version best fits your workload requirements.

Engine status

Engine status by version

Feature / Capability	v3.6	v4.0	v5.0	v8.0	
Support status	EOL: Mar 2026	Active	Active (LTS available)	Active (GA Nov 2025)	
Extended Support available	Yes (paid, until Mar 2029)	N/A	N/A	N/A	
LTS release available	No	No	Yes (v5.0 LTS)	No	
Elastic clusters	No	No	Yes	No	

Cluster types

Cluster types by version

Cluster Type	v3.6	v4.0	v5.0	v8.0	Elastic clusters
Instance-based cluster	Yes	Yes	Yes	Yes	Yes
Serverless instances	No	No	Yes	Yes	No

Features

Features and configurations by engine version

Feature	v3.6	v4.0	v5.0	v8.0	Elastic clusters
Global clusters	No	Yes	Yes	Yes	No
Multi-AZ (3 AZs)	Yes	Yes	Yes	Yes	Yes (per shard)
Read replicas supported	15	15	15	15	15 per shard
ACID transactions	No	Yes	Yes	Yes	No
Retryable writes	No	No	No	Yes (8.0.2+)	No
Change streams	Yes	Yes	Yes	Yes	No
Vector search	No	No	Yes	Yes	No
Performance Insights	Yes	Yes	Yes	Yes	No
In-place major version upgrade (MVU)	Yes (to 5.0 or 8.0)	Yes (to 5.0 or 8.0)	Yes (to 8.0)	No (target only)	No
Client-side field level encryption (FLE)	No	No	Yes	Yes	No
I/O-Optimized storage	No	No	Yes	Yes	No
TLS certificate rotation (no reboot)	Yes (patch 1.0.208662+)	Yes (patch 2.0.10179+)	Yes (patch 3.0.4780+)	Yes	Yes
Cross-Region snapshot copy	Yes	Yes	Yes	Yes	Yes

Feature	v3.6	v4.0	v5.0	v8.0	Elastic clusters
Cross-account snapshot sharing	Yes	Yes	Yes	Yes	No
Audit and slow query logs	Yes	Yes	Yes	Yes	Yes
Dual-stack IPv4/IPv6	No	Yes (patch 2.0.11747 +)	Yes (patch 3.0.15902 +)	Yes	N/A
Collation	No	No	No	Yes	No
Views	No	No	No	Yes	No
Compression Support	No	No	LZ4	LZ4/Ztsd	No
Text Index	No	No	V1	V2	No
FedRAMP (GovCloud)	No	No	Yes	Yes	No

Instance types

Instance type support by version

Instance Type	v3.6	v4.0	v5.0	v8.0	
db.r4 (previous generation)	Yes	No	No	No	
db.r5	Yes	Yes	Yes	Yes	
db.r6g (Graviton2)	No	Yes	Yes	Yes	
db.r6gd (NVMe-backed Graviton2)	No	No	Yes	Yes	

Instance Type	v3.6	v4.0	v5.0	v8.0	
db.r8g (Graviton4)	No	No	Yes	Yes	
db.t3.medium	Yes	Yes	Yes	Yes	
db.t4g.medium (Graviton 2)	No	Yes	Yes	Yes	
serverless	No	No	Yes	Yes	

AWS integrations

AWS integration support by version

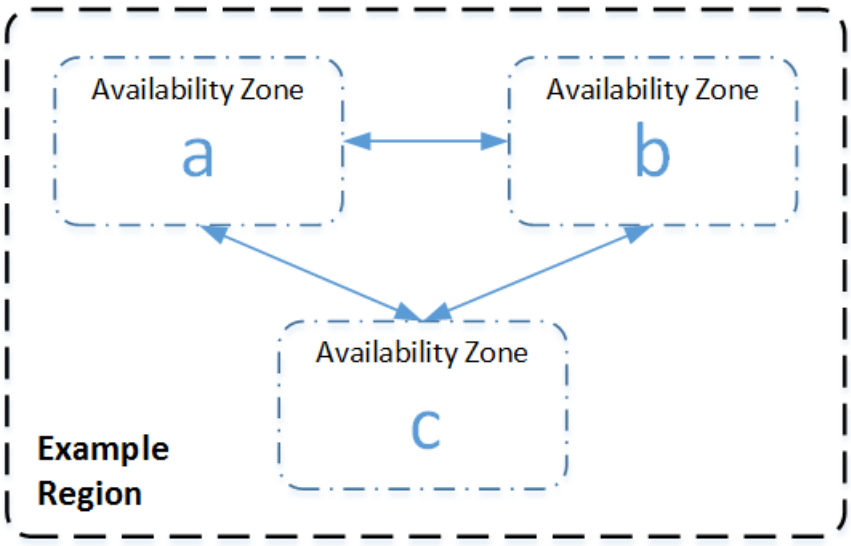
AWS Integration / Feature	v3.6	v4.0	v5.0	v8.0	Elastic clusters
AWS Backup	Yes	Yes	Yes	Yes	No
AWS Secrets Manager	Yes	Yes	Yes	Yes	No
AWS Lambda (native ESM)	No	Yes	Yes	Yes	No
Amazon OpenSearch Service Ingestion (Zero-ETL)	No	Yes	Yes	Yes	No
AWS DMS migration support	Yes	Yes	Yes	Yes (from 5.0 via AWS DMS)	Yes
AWS Identity and Access Management authentication	No	No	Yes (5.0 instance-based)	Yes	No
AWS CloudShell	No	No	Yes	Yes	No

AWS Integration / Feature	v3.6	v4.0	v5.0	v8.0	Elastic clusters
AWS Key Management Service	Yes	Yes	Yes	Yes	Yes
AWS CloudTrail	Yes	Yes	Yes	Yes	Yes

Choosing Regions and Availability Zones

Amazon cloud computing resources are hosted in multiple locations worldwide. These locations consist of AWS Regions and Availability Zones. Each *AWS Region* is a separate geographic area. Each Region has multiple, isolated locations known as *Availability Zones*. Amazon DocumentDB provides you the ability to place resources, such as instances, and data in multiple locations. Resources aren't replicated across AWS Regions unless you do so specifically.

Amazon operates advanced, highly available data centers. Although rare, failures can occur that affect the availability of instances that are in the same location. If you host all your instances in a single location that is affected by such a failure, none of your instances would be available. The following diagram shows an AWS Region with three Availability Zones.



It is important to remember that each Region is independent. Any Amazon DocumentDB activity that you initiate (for example, creating instances or listing available instances) runs only in your current default AWS Region. You can change the default Region on the console by setting the `EC2_REGION` environment variable. Or you can override it by using the `--region` parameter in the

AWS CLI. For more information, see Configuring the AWS Command Line Interface, specifically, the sections on environment variables and command line options.

When you create a cluster using the Amazon DocumentDB console, and you choose to create a replica in a different Availability Zone, Amazon DocumentDB creates two instances. It creates the primary instance in one Availability Zone and the replica instance in a different Availability Zone. The cluster volume is always replicated across three Availability Zones.

To create or work with an Amazon DocumentDB instance in a specific AWS Region, use the corresponding regional service endpoint.

Region availability

Amazon DocumentDB is available in the following AWS Regions.

Regions supported by Amazon DocumentDB

Region Name	Region	Availability Zones (compute)
US East (Ohio)	`us-east-2`	3
US East (N. Virginia)	`us-east-1`	6
US West (Oregon)	`us-west-2`	4
Africa (Cape Town)	`af-south-1`	3
South America (São Paulo)	`sa-east-1`	3
Asia Pacific (Hong Kong)	`ap-east-1`	3
Asia Pacific (Taipei)	`ap-east-2`	3
Asia Pacific (Hyderabad)	`ap-south-2`	3
Asia Pacific (Malaysia)	`ap-southeast-5`	3
Asia Pacific (Mumbai)	`ap-south-1`	3

Region Name	Region	Availability Zones (compute)
Asia Pacific (Osaka)	ap-northeast-3	3
Asia Pacific (Seoul)	ap-northeast-2	4
Asia Pacific (Singapore)	ap-southeast-1	3
Asia Pacific (Sydney)	ap-southeast-2	3
Asia Pacific (Jakarta)	ap-southeast-3	3
Asia Pacific (Melbourne)	ap-southeast-4	3
Asia Pacific (New Zealand)	ap-southeast-6	3
Asia Pacific (Thailand)	ap-southeast-7	3
Asia Pacific (Tokyo)	ap-northeast-1	3
Canada (Central)	ca-central-1	3
Canada West (Calgary)	ca-west-1	3
China (Beijing) Region	cn-north-1	3
China (Ningxia)	cn-northwest-1	3
Europe (Frankfurt)	eu-central-1	3
Europe (Zurich)	eu-central-2	3
Europe (Ireland)	eu-west-1	3
Europe (London)	eu-west-2	3
Europe (Milan)	eu-south-1	3

Region Name	Region	Availability Zones (compute)
Europe (Paris)	`eu-west-3`	3
Europe (Spain)	`eu-south-2`	3
Europe (Stockholm)	`eu-north-1`	3
Mexico (Central)	`mx-central-1`	3
Middle East (UAE)	`me-central-1`	3
Israel (Tel Aviv)	`il-central-1`	3
AWS GovCloud (US-West)	`us-gov-west-1`	3
AWS GovCloud (US-East)	`us-gov-east-1`	3

By default, the time zone for an Amazon DocumentDB cluster is Universal Time Coordinated (UTC).

For information on finding the connection endpoints for clusters and instances in a particular Region, see Understanding Amazon DocumentDB endpoints.

Zero-ETL integration with Amazon OpenSearch Service

Topics

- Amazon OpenSearch Service as a destination
- Limitations

Amazon OpenSearch Service as a destination

OpenSearch Service integration with Amazon DocumentDB enables you to stream full load and change data events to OpenSearch domains. The ingestion infrastructure is hosted as OpenSearch ingestion pipelines and provides a high-scale, low latency mechanism to continuously stream data from Amazon DocumentDB collections.

During full load, the zero-ETL integration first extracts historical full load data to OpenSearch using an ingestion pipeline. Once full load data is ingested, the OpenSearch ingestion pipelines will start reading data from Amazon DocumentDB change streams and eventually catch up to maintain near real time data consistency between Amazon DocumentDB and OpenSearch. OpenSearch stores documents in indexes. Incoming data from a Amazon DocumentDB collections can be sent to either one index or can be partitioned into different indices. Ingestion pipelines will sync all create, update and delete events in an Amazon DocumentDB collection as corresponding create, update, and delete of OpenSearch documents to keep both data systems in sync. Ingestion pipelines can be configured to read data from one collection and write to one index or read data from one collection and conditionally route to multiple indexes.

Ingestion pipelines can be configured to stream data from Amazon DocumentDB to Amazon OpenSearch Service using:

- Full load only
- Stream change stream events from Amazon DocumentDB without full load
- Full load followed by change streams from Amazon DocumentDB

To set up your ingestion pipeline, perform the following steps:

Step 1: Create an Amazon OpenSearch Service domain or OpenSearch serverless collection

An Amazon OpenSearch Service collection with appropriate permissions to read data is required. Refer to Getting started with Amazon OpenSearch Service or Getting started with Amazon OpenSearch Serverless in the *Amazon OpenSearch Service* Developer Guide to create a collection. Refer to Amazon OpenSearch Ingestion in the *Amazon OpenSearch Service* Developer Guide to create an AIM role with the correct permissions to access write data to the collection or domain.

Step 2: Enable change streams on the Amazon DocumentDB cluster

Ensure that change streams are enabled on the required collections in the Amazon DocumentDB cluster. Refer to Using change streams with Amazon DocumentDB for more information.

Step 3: Set up the pipeline role with permissions to write to the Amazon S3 bucket and destination domain or collection

After you have your Amazon DocumentDB collection created and change stream enabled, set up the pipeline role that you want to use in your pipeline configuration, and add the following permissions in the role:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "allowReadAndWriteToS3ForExport",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:AbortMultipartUpload",
                "s3:PutObject",
                "s3:PutObjectAcl"
            ],
            "Resource": [
                "arn:aws:s3:::my-bucket/export/*"
            ]
        }
    ]
}
```

In order for an OpenSearch pipeline to write data to an OpenSearch domain, the domain must have a domain-level access policy that allows the **sts_role_arn** pipeline role to access it. The following sample domain access policy allows the pipeline role named `pipeline-role`, which you created in the previous step, to write data to the domain named `ingestion-domain`:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::{your-account-id}:role/{pipeline-role}"
      },
```

```
      "Action": ["es:DescribeDomain", "es:ESHttp*"],
      "Resource": "arn:aws:es:{region}:{your-account-id}:domain/{domain-name}/*"
    }
  ]
}
```

Step 4: Add the permissions required on the pipeline role to create X-ENI

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:AttachNetworkInterface",
                "ec2:CreateNetworkInterface",
                "ec2:CreateNetworkInterfacePermission",
                "ec2:DeleteNetworkInterface",
                "ec2:DeleteNetworkInterfacePermission",
                "ec2:DetachNetworkInterface",
                "ec2:DescribeNetworkInterfaces"
            ],
            "Resource": [
                "arn:aws:ec2:*:420497401461:network-interface/*",
                "arn:aws:ec2:*:420497401461:subnet/*",
                "arn:aws:ec2:*:420497401461:security-group/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeDhcpOptions",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs",
                "ec2:Describe*"
            ],
            "Resource": "*"
        },
```

```
        {
            "Effect": "Allow",
            "Action": [ "ec2:CreateTags" ],
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
               "StringEquals": { "aws:RequestTag/OSISManaged": "true" }
            }
        }
    ]
}
```

Step 5: Create the pipeline

Configure an OpenSearch ingestion pipeline specifying Amazon DocumentDB as the source. This sample pipeline configuration assumes the use of a change stream fetching mechanism. Refer to Using an OpenSearch Ingestion pipeline with Amazon DocumentDB in the *Amazon OpenSearch Service Developer Guide* for more information.

Limitations

The following limitations apply to the Amazon DocumentDB OpenSearch integration:

- Only one Amazon DocumentDB collection as the source per pipeline is supported.
- Cross-region data ingestion is not supported. Your Amazon DocumentDB cluster and OpenSearch domain must be in the same AWS Region.
- Cross-account data ingestion is not supported. Your Amazon DocumentDB cluster and OpenSearch ingestion pipeline must be in the same AWS account.
- Amazon DocumentDB elastic clusters are not supported. Only Amazon DocumentDB instance-based clusters are supported.
- Ensure that the Amazon DocumentDB cluster has authentication enabled using AWS secrets. AWS secrets are the only supported authentication mechanism.
- The existing pipeline configuration can not be updated to ingest data from a different database and/or a different collection. To update the database and/or collection name of a pipeline, you must create a new pipeline.

Developing with Amazon DocumentDB

This section covers connecting to your Amazon DocumentDB cluster, querying and managing data, working with indexes, and using features like change streams, transactions, and vector search for generative AI applications.

Topics

- Connect to Amazon DocumentDB
- Program with Amazon DocumentDB
- Querying in Amazon DocumentDB
- Indexes
- Using change streams with Amazon DocumentDB
- Transactions in Amazon DocumentDB
- Retryable writes in Amazon DocumentDB
- Using collation in Amazon DocumentDB 8.0
- Using views in Amazon DocumentDB 8.0
- Amazon DocumentDB generative artificial intelligence
- Using AWS Lambda with change streams

Connect to Amazon DocumentDB

Amazon DocumentDB provides multiple connection options, including programmatic connections using MongoDB drivers, connecting from outside a VPC through SSH tunneling or AWS VPN, and popular tools like Studio 3T and DataGrip. The service supports replica set connections for high availability. For analytics and reporting, you can connect using JDBC or ODBC drivers. EC2 instances in the same VPC can connect directly to DocumentDB clusters. All connections require TLS and proper AWS credentials/authentication.

Topics

- Connecting programmatically to Amazon DocumentDB
- Connecting to an Amazon DocumentDB cluster from outside an Amazon VPC
- Connecting to Amazon DocumentDB as a replica set
- Connecting to an Amazon DocumentDB cluster from Studio 3T

- Connect to Amazon DocumentDB using DataGrip
- Connect using Amazon EC2
- Connect using the Amazon DocumentDB JDBC driver
- Connect using the Amazon DocumentDB ODBC driver

Connecting programmatically to Amazon DocumentDB

This section contains code examples that demonstrate how to connect to Amazon DocumentDB (with MongoDB compatibility) using several different languages. The examples are separated into two sections based on whether you are connecting to a cluster that has Transport Layer Security (TLS) enabled or disabled. By default, TLS is enabled on Amazon DocumentDB clusters. However, you can turn off TLS if you want. For more information, see Encrypting data in transit.

If you are attempting to connect to your Amazon DocumentDB from outside the VPC in which your cluster resides, see Connecting to an Amazon DocumentDB cluster from outside an Amazon VPC.

Before you connect to your cluster, you must know whether TLS is enabled on the cluster. The next section shows you how to determine the value of your cluster's `tls` parameter using either the AWS Management Console or the AWS CLI. Following that, you can continue by finding and applying the appropriate code example.

Topics

- Determining the value of your tls parameter
- Connecting with TLS enabled
- Connecting with TLS disabled
- Network compression in Amazon DocumentDB

Determining the value of your `tls` parameter

Determining whether your cluster has TLS enabled is a two-step process that you can perform using either the AWS Management Console or AWS CLI.

1. **Determine which parameter group is governing your cluster.**

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the left navigation pane, choose **Clusters**.
3. In the list of clusters, select the name of your cluster.
4. The resulting page shows the details of the cluster that you selected. Select the **Configuration** tab. In the **Configurations and status** section, locate the parameter group's name below **Cluster parameter group**.

Using the AWS CLI

The following AWS CLI code determines which parameter is governing your cluster. Make sure you replace *sample-cluster* with the name of your cluster.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].[DBClusterIdentifier,DBClusterParameterGroup]'
```

Output from this operation looks something like the following:

```
[
       [
           "sample-cluster",
           "sample-parameter-group"
       ]
]
```

2. **Determine the value of the `tls` parameter in your cluster's parameter group.**

Using the AWS Management Console

1. In the navigation pane, choose **Parameter groups**.
2. In the **Cluster parameter groups** window, select your cluster parameter group name from Step 1d.

3. The resulting page shows your cluster parameter group's parameters. You can see the value of the `tls` parameter here. For information on modifying this parameter, see Modifying Amazon DocumentDB cluster parameter groups.

Using the AWS CLI

You can use the `describe-db-cluster-parameters` AWS CLI command to view the details of the parameters in your cluster parameter group.

- **`--describe-db-cluster-parameters`** — To list all the parameters inside a parameter group and their values.
  - **`--db-cluster-parameter-group name`** — Required. The name of your cluster parameter group.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

```
aws docdb describe-db-cluster-parameters \
    --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following:

```
{
        "Parameters": [
            {
                "ParameterName": "profiler_threshold_ms",
                "ParameterValue": "100",
                "Description": "Operations longer than profiler_threshold_ms
 will be logged",
                "Source": "system",
                "ApplyType": "dynamic",
                "DataType": "integer",
                "AllowedValues": "50-2147483646",
                "IsModifiable": true,
                "ApplyMethod": "pending-reboot"
            },
            {
                "ParameterName": "tls",
                "ParameterValue": "disabled",
```

```
                "Description": "Config to enable/disable TLS",
                "Source": "user",
                "ApplyType": "static",
                "DataType": "string",
                "AllowedValues": "disabled,enabled,fips-140-3",
                "IsModifiable": true,
                "ApplyMethod": "pending-reboot"
            }
        ]
}
```

> **Note**
>
> Amazon DocumentDB supports FIPS 140-3 endpoints starting with Amazon DocumentDB 5.0 (engine version 3.0.3727) clusters in these Regions: ca-central-1, us-west-2, us-east-1, us-east-2, us-gov-east-1, us-gov-west-1.

After determining the value of your `tls` parameter, continue connecting to your cluster by using one of the code examples in the following sections.

- Connecting with TLS enabled
- Connecting with TLS disabled

Connecting with TLS enabled

To view a code example for programmatically connecting to a TLS-enabled Amazon DocumentDB cluster, choose the appropriate tab for the language that you want to use.

To encrypt data in transit, download the public key for Amazon DocumentDB named `global-bundle.pem` using the following operation.

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

If your application is on Microsoft Windows and requires a PKCS7 file, you can download the PKCS7 certificate bundle. This bundle contains both the intermediate and root certificates at https://truststore.pki.rds.amazonaws.com/global/global-bundle.p7b.

Python

The following code demonstrates how to connect to Amazon DocumentDB using Python when TLS is enabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
import pymongo
import sys

##Create a MongoDB client, open a connection to Amazon DocumentDB as a replica set
 and specify the read preference as secondary preferred
client = pymongo.MongoClient('mongodb://sample-user:password@sample-
cluster.node.us-east-1.docdb.amazonaws.com:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

##Specify the database to be used
db = client.sample_database

##Specify the collection to be used
col = db.sample_collection

##Insert a single document
col.insert_one({'hello':'Amazon DocumentDB'})

##Find the document that was previously written
x = col.find_one({'hello':'Amazon DocumentDB'})

##Print the result to the screen
print(x)

##Close the connection
client.close()
```

Node.js

The following code demonstrates how to connect to Amazon DocumentDB using Node.js when TLS is enabled.

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

In the following example, replace each *user input placeholder* with your cluster's information.

```
var MongoClient = require('mongodb').MongoClient

//Create a MongoDB client, open a connection to DocDB; as a replica set,
//  and specify the read preference as secondary preferred

var client = MongoClient.connect(
'mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/sample-database?
tls=true&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false',
{
  tlsCAFile: `global-bundle.pem` //Specify the DocDB; cert
},
function(err, client) {
    if(err)
        throw err;

    //Specify the database to be used
    db = client.db('sample-database');

    //Specify the collection to be used
    col = db.collection('sample-collection');

    //Insert a single document
    col.insertOne({'hello':'Amazon DocumentDB'}, function(err, result){
      //Find the document that was previously written
      col.findOne({'hello':'Amazon DocumentDB'}, function(err, result){
        //Print the result to the screen
        console.log(result);

        //Close the connection
        client.close()
```

```
      });
   });
});
```

PHP

The following code demonstrates how to connect to Amazon DocumentDB using PHP when TLS is enabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
<?php
//Include Composer's autoloader
require 'vendor/autoload.php';

$TLS_DIR = "/home/ubuntu/global-bundle.pem";

//Create a MongoDB client and open connection to Amazon DocumentDB
$client = new MongoDB\Client("mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/?retryWrites=false", ["tls" => "true", "tlsCAFile"
 => $TLS_DIR ]);

//Specify the database and collection to be used
$col = $client->sampledatabase->samplecollection;

//Insert a single document
$result = $col->insertOne( [ 'hello' => 'Amazon DocumentDB'] );

//Find the document that was previously written
$result = $col->findOne(array('hello' => 'Amazon DocumentDB'));

//Print the result to the screen
print_r($result);
?>
```

Go

The following code demonstrates how to connect to Amazon DocumentDB using Go when TLS is enabled.

> **Note**
>
> As of version 1.2.1, the MongoDB Go Driver will only use the first CA server certificate found in `sslcertificateauthorityfile`. The example code below addresses this limitation by manually appending all server certificates found in `sslcertificateauthorityfile` to a custom TLS configuration used during client creation.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

```
package main

import (
 "context"
 "fmt"
 "log"
 "time"

 "go.mongodb.org/mongo-driver/bson"
 "go.mongodb.org/mongo-driver/mongo"
 "go.mongodb.org/mongo-driver/mongo/options"

 "io/ioutil"
 "crypto/tls"
 "crypto/x509"
 "errors"
)

const (
 // Path to the AWS CA file
 caFilePath = "global-bundle.pem"

 // Timeout operations after N seconds
 connectTimeout  = 5
 queryTimeout    = 30
 username        = "sample-user"
 password        = "password"
 clusterEndpoint = "sample-cluster.node.us-east-1.docdb.amazonaws.com:27017"

 // Which instances to read from
```

```
 readPreference = "secondaryPreferred"

 connectionStringTemplate = "mongodb://%s:%s@%s/sample-database?
tls=true&replicaSet=rs0&readpreference=%s"
)

func main() {

 connectionURI := fmt.Sprintf(connectionStringTemplate, username, password,
 clusterEndpoint, readPreference)

 tlsConfig, err := getCustomTLSConfig(caFilePath)
 if err != nil {
  log.Fatalf("Failed getting TLS configuration: %v", err)
 }

 client, err :=
 mongo.NewClient(options.Client().ApplyURI(connectionURI).SetTLSConfig(tlsConfig))
 if err != nil {
  log.Fatalf("Failed to create client: %v", err)
 }

 ctx, cancel := context.WithTimeout(context.Background(),
 connectTimeout*time.Second)
 defer cancel()

 err = client.Connect(ctx)
 if err != nil {
  log.Fatalf("Failed to connect to cluster: %v", err)
 }

 // Force a connection to verify our connection string
 err = client.Ping(ctx, nil)
 if err != nil {
  log.Fatalf("Failed to ping cluster: %v", err)
 }

 fmt.Println("Connected to DocumentDB!")

 collection := client.Database("sample-database").Collection("sample-collection")

 ctx, cancel = context.WithTimeout(context.Background(), queryTimeout*time.Second)
 defer cancel()
```

```
 res, err := collection.InsertOne(ctx, bson.M{"name": "pi", "value": 3.14159})
 if err != nil {
  log.Fatalf("Failed to insert document: %v", err)
 }

 id := res.InsertedID
 log.Printf("Inserted document ID: %s", id)

 ctx, cancel = context.WithTimeout(context.Background(), queryTimeout*time.Second)
 defer cancel()

 cur, err := collection.Find(ctx, bson.D{})

 if err != nil {
  log.Fatalf("Failed to run find query: %v", err)
 }
 defer cur.Close(ctx)

 for cur.Next(ctx) {
  var result bson.M
  err := cur.Decode(&result)
  log.Printf("Returned: %v", result)

  if err != nil {
   log.Fatal(err)
  }
 }

 if err := cur.Err(); err != nil {
  log.Fatal(err)
 }

}

func getCustomTLSConfig(caFile string) (*tls.Config, error) {
 tlsConfig := new(tls.Config)
 certs, err := ioutil.ReadFile(caFile)

 if err != nil {
  return tlsConfig, err
 }

 tlsConfig.RootCAs = x509.NewCertPool()
 ok := tlsConfig.RootCAs.AppendCertsFromPEM(certs)
```

```
if !ok {
 return tlsConfig, errors.New("Failed parsing pem file")
}

return tlsConfig, nil
```

Java

When connecting to a TLS-enabled Amazon DocumentDB cluster from a Java application, your program must use the AWS-provided certificate authority (CA) file to validate the connection. To use the Amazon RDS CA certificate, do the following:

1. Download the Amazon RDS CA file from https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem .
2. Create a trust store with the CA certificate contained in the file by performing the following commands. Be sure to change the *truststore-password* to something else. If you are accessing a trust store that contains both the old CA certificate (`rds-ca-2015-root.pem`) and the new CA certificate (`rds-ca-2019-root.pem`), you can import the certificate bundle into the trust store.

   The following is a sample shell script that imports the certificate bundle into a trust store on a Linux operating system. In the following examples, replace each *user input placeholder* with your own information. Most notably, wherever the example directory "*mydir*" is located in the script, replace it with a directory you created for this task.

   ```
   mydir=/tmp/certs
   truststore=${mydir}/rds-truststore.jks
   storepassword=truststore-password

   curl -sS "https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem" >
    ${mydir}/global-bundle.pem
   awk 'split_after == 1 {n++;split_after=0} /-----END CERTIFICATE-----/
    {split_after=1}{print > "rds-ca-" n ".pem"}' < ${mydir}/global-bundle.pem

   for CERT in rds-ca-*; do
     alias=$(openssl x509 -noout -text -in $CERT | perl -ne 'next unless /
   Subject:/; s/.*(CN=|CN = )//; print')
     echo "Importing $alias"
     keytool -import -file ${CERT} -alias "${alias}" -storepass ${storepassword} -
   keystore ${truststore} -noprompt
   ```

```
  rm $CERT
done

rm ${mydir}/global-bundle.pem

echo "Trust store content is: "

keytool -list -v -keystore "$truststore" -storepass ${storepassword} | grep
 Alias | cut -d " " -f3- | while read alias
do
   expiry=`keytool -list -v -keystore "$truststore" -storepass ${storepassword}
 -alias "${alias}" | grep Valid | perl -ne 'if(/until: (.*?)\n/) { print
 "$1\n"; }'`
   echo " Certificate ${alias} expires in '$expiry'"
done
```

The following is a sample shell script that imports the certificate bundle into a trust store on macOS.

```
mydir=/tmp/certs
truststore=${mydir}/rds-truststore.jks
storepassword=truststore-password

curl -sS "https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem" >
 ${mydir}/global-bundle.pem
split -p "-----BEGIN CERTIFICATE-----" ${mydir}/global-bundle.pem rds-ca-

for CERT in rds-ca-*; do
  alias=$(openssl x509 -noout -text -in $CERT | perl -ne 'next unless /
Subject:/; s/.*(CN=|CN = )//; print')
  echo "Importing $alias"
  keytool -import -file ${CERT} -alias "${alias}" -storepass ${storepassword} -
keystore ${truststore} -noprompt
  rm $CERT
done

rm ${mydir}/global-bundle.pem

echo "Trust store content is: "

keytool -list -v -keystore "$truststore" -storepass ${storepassword} | grep
 Alias | cut -d " " -f3- | while read alias
```

```
do
   expiry=`keytool -list -v -keystore "$truststore" -storepass ${storepassword}
 -alias "${alias}" | grep Valid | perl -ne 'if(/until: (.*?)\n/) { print
 "$1\n"; }'`
   echo " Certificate ${alias} expires in '$expiry'"
done
```

3. Use the `keystore` in your program by setting the following system properties in your application before making a connection to the Amazon DocumentDB cluster.

```
javax.net.ssl.trustStore: truststore
javax.net.ssl.trustStorePassword: truststore-password;
```

4. The following code demonstrates how to connect to Amazon DocumentDB using Java when TLS is enabled.

   In the following example, replace each *user input placeholder* with your cluster's information.

```
package com.example.documentdb;

import com.mongodb.client.*;
import org.bson.Document;

public final class Test {
    private Test() {
    }
    public static void main(String[] args) {

        String template = "mongodb://%s:%s@%s/sample-database?
ssl=true&replicaSet=rs0&readpreference=%s";
        String username = "sample-user";
        String password = "password";
        String clusterEndpoint = "sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017";
        String readPreference = "secondaryPreferred";
        String connectionString = String.format(template, username, password,
 clusterEndpoint, readPreference);

        String truststore = "truststore";
        String truststorePassword = "truststore-password";

        System.setProperty("javax.net.ssl.trustStore", truststore);
```

```
        System.setProperty("javax.net.ssl.trustStorePassword",
 truststorePassword);

        MongoClient mongoClient = MongoClients.create(connectionString);

        MongoDatabase testDB = mongoClient.getDatabase("sample-database");
        MongoCollection<Document> numbersCollection =
 testDB.getCollection("sample-collection");

        Document doc = new Document("name", "pi").append("value", 3.14159);
        numbersCollection.insertOne(doc);

        MongoCursor<Document> cursor = numbersCollection.find().iterator();
        try {
            while (cursor.hasNext()) {
                System.out.println(cursor.next().toJson());
            }
        } finally {
            cursor.close();
        }

    }
}
```

C# / .NET

The following code demonstrates how to connect to Amazon DocumentDB using C# / .NET when TLS is enabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
using System;
using System.Text;
using System.Linq;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
using System.Net.Security;
using MongoDB.Driver;
using MongoDB.Bson;
```

```
namespace DocDB
{
    class Program
    {
        static void Main(string[] args)
        {
            string template = "mongodb://{0}:{1}@{2}/sampledatabase?
tls=true&replicaSet=rs0&readpreference={3}";
            string username = "sample-user";
            string password = "password";
            string readPreference = "secondaryPreferred";
            string clusterEndpoint="sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017";
            string connectionString = String.Format(template, username, password,
 clusterEndpoint, readPreference);

            string pathToCAFile = "<PATH/global-bundle.p7b_file>";

            // ADD CA certificate to local trust store
            // DO this once - Maybe when your service starts
            X509Store localTrustStore = new X509Store(StoreName.Root);
            X509Certificate2Collection certificateCollection = new
 X509Certificate2Collection();
            certificateCollection.Import(pathToCAFile);
            try
            {
                localTrustStore.Open(OpenFlags.ReadWrite);
                localTrustStore.AddRange(certificateCollection);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Root certificate import failed: " + ex.Message);
                throw;
            }
            finally
            {
                localTrustStore.Close();
            }

            var settings = MongoClientSettings.FromUrl(new
 MongoUrl(connectionString));
            var client = new MongoClient(settings);

            var database = client.GetDatabase("sampledatabase");
```

```
            var collection =
 database.GetCollection<BsonDocument>("samplecollection");
            var docToInsert = new BsonDocument { { "pi", 3.14159 } };
            collection.InsertOne(docToInsert);
        }
    }
}
```

MongoDB Shell

The following code demonstrates how to connect to and query Amazon DocumentDB using the newest version, mongosh, or the previous mongo shell version, when TLS is enabled.

Connect to Amazon DocumentDB with mongosh

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

```
mongosh --tls --host cluster-end-point:27017 --tlsCAFile  global-bundle.pem --
username sample-user --password password --retryWrites false
```

Connect to Amazon DocumentDB with the previous mongo shell version

If you use IAM, you must use a previous version of mongo shell. Enter one of the following command options:

```
mongo --ssl --host cluster-end-point:27017 --sslCAFile global-bundle.pem --
username sample-user --password password
```

If you are using a version equal to or greater than 4.2, use the following code to connect. Starting with engine version 8.0.2, Amazon DocumentDB supports retryable writes. On earlier

engine versions, retryable writes are not supported. If you are using legacy mongo shell (not mongosh), do not include the `retryWrites=false` command in any code string. For more information, see Retryable writes in Amazon DocumentDB.

```
mongo --tls --host cluster-end-point:27017 --tlsCAFile global-bundle.pem --
username sample-user --password password
```

Test the connection

1. Insert a single document.

   ```
   db.myTestCollection.insertOne({'hello':'Amazon DocumentDB'})
   ```

2. Find the document that was previously inserted.

   ```
   db.myTestCollection.find({'hello':'Amazon DocumentDB'})
   ```

R

The following code demonstrates how to connect to Amazon DocumentDB with R using mongolite (https://jeroen.github.io/mongolite/) when TLS is enabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
#Include the mongolite library.
library(mongolite)

mongourl <- paste("mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/test2?ssl=true&",
          "readPreference=secondaryPreferred&replicaSet=rs0", sep="")

#Create a MongoDB client, open a connection to Amazon DocumentDB as a replica
#   set and specify the read preference as secondary preferred
client <-  mongo(url = mongourl, options = ssl_options(weak_cert_validation = F, ca
 ="<PATH/global-bundle.pem>"))

#Insert a single document
str <- c('{"hello" : "Amazon DocumentDB"}')
client$insert(str)
```

```
#Find the document that was previously written
client$find()
```

Ruby

The following code demonstrates how to connect to Amazon DocumentDB with Ruby when TLS is enabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
require 'mongo'
require 'neatjson'
require 'json'
client_host = 'mongodb://sample-cluster.node.us-east-1.docdb.amazonaws.com:27017'
client_options = {
   database: 'test',
   replica_set: 'rs0',
   read: {:secondary_preferred => 1},
   user: 'sample-user',
   password: 'password',
   ssl: true,
   ssl_verify: true,
   ssl_ca_cert: 'PATH/global-bundle.pem',
   retry_writes: false
}


begin
   ##Create a MongoDB client, open a connection to Amazon DocumentDB as a
   ##   replica set and specify the read preference as secondary preferred
   client = Mongo::Client.new(client_host, client_options)

   ##Insert a single document
   x = client[:test].insert_one({"hello":"Amazon DocumentDB"})

   ##Find the document that was previously written
   result = client[:test].find()

   #Print the document
   result.each do |document|
      puts JSON.neat_generate(document)
```

```
   end
end

#Close the connection
client.close
```

Connecting with TLS disabled

To view a code example for programmatically connecting to a TLS-disabled Amazon DocumentDB cluster, choose the tab for language that you want to use.

Python

The following code demonstrates how to connect to Amazon DocumentDB using Python when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
## Create a MongoDB client, open a connection to Amazon DocumentDB as a replica set
 and specify the read preference as secondary preferred

import pymongo
import sys

client = pymongo.MongoClient('mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/?
replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

##Specify the database to be used
db = client.sample_database

##Specify the collection to be used
col = db.sample_collection

##Insert a single document
col.insert_one({'hello':'Amazon DocumentDB'})

##Find the document that was previously written
x = col.find_one({'hello':'Amazon DocumentDB'})

##Print the result to the screen
```

```
print(x)

##Close the connection
client.close()
```

Node.js

The following code demonstrates how to connect to Amazon DocumentDB using Node.js when TLS is disabled.

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

In the following example, replace each *user input placeholder* with your cluster's information.

```
var MongoClient = require('mongodb').MongoClient;

//Create a MongoDB client, open a connection to Amazon DocumentDB as a replica set,
//  and specify the read preference as secondary preferred
var client = MongoClient.connect(
'mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/sample-database?
replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false',
{
  useNewUrlParser: true
},

function(err, client) {
    if(err)
        throw err;
    //Specify the database to be used
    db = client.db('sample-database');

    //Specify the collection to be used
    col = db.collection('sample-collection');
```

```
    //Insert a single document
    col.insertOne({'hello':'Amazon DocumentDB'}, function(err, result){
      //Find the document that was previously written
      col.findOne({'hello':'Amazon DocumentDB'}, function(err, result){
        //Print the result to the screen
        console.log(result);

        //Close the connection
        client.close()
      });
   });
});
```

PHP

The following code demonstrates how to connect to Amazon DocumentDB using PHP when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
<?php
//Include Composer's autoloader
require 'vendor/autoload.php';

//Create a MongoDB client and open connection to Amazon DocumentDB
$client = new MongoDB\Client("mongodb://sample-user:password@sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017/?retryWrites=false");

//Specify the database and collection to be used
$col = $client->sampledatabase->samplecollection;

//Insert a single document
$result = $col->insertOne( [ 'hello' => 'Amazon DocumentDB'] );

//Find the document that was previously written
$result = $col->findOne(array('hello' => 'Amazon DocumentDB'));

//Print the result to the screen
print_r($result);
?>
```

Go

The following code demonstrates how to connect to Amazon DocumentDB using Go when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
package main

import (
 "context"
 "fmt"
 "log"
 "time"

 "go.mongodb.org/mongo-driver/bson"
 "go.mongodb.org/mongo-driver/mongo"
 "go.mongodb.org/mongo-driver/mongo/options"
)

const (
 // Timeout operations after N seconds
 connectTimeout  = 5
 queryTimeout    = 30
 username        = "sample-user"
 password        = "password"
 clusterEndpoint = "sample-cluster.node.us-east-1.docdb.amazonaws.com:27017"

 // Which instances to read from
 readPreference           = "secondaryPreferred"
 connectionStringTemplate = "mongodb://%s:%s@%s/sample-database?
replicaSet=rs0&readpreference=%s"
)

func main() {

 connectionURI := fmt.Sprintf(connectionStringTemplate, username, password,
 clusterEndpoint, readPreference)

 client, err := mongo.NewClient(options.Client().ApplyURI(connectionURI))
 if err != nil {
  log.Fatalf("Failed to create client: %v", err)
```

```
}

ctx, cancel := context.WithTimeout(context.Background(),
connectTimeout*time.Second)
defer cancel()

err = client.Connect(ctx)
if err != nil {
 log.Fatalf("Failed to connect to cluster: %v", err)
}

// Force a connection to verify our connection string
err = client.Ping(ctx, nil)
if err != nil {
 log.Fatalf("Failed to ping cluster: %v", err)
}

fmt.Println("Connected to DocumentDB!")

collection := client.Database("sample-database").Collection("sample-collection")

ctx, cancel = context.WithTimeout(context.Background(), queryTimeout*time.Second)
defer cancel()

res, err := collection.InsertOne(ctx, bson.M{"name": "pi", "value": 3.14159})
if err != nil {
 log.Fatalf("Failed to insert document: %v", err)
}

id := res.InsertedID
log.Printf("Inserted document ID: %s", id)

ctx, cancel = context.WithTimeout(context.Background(), queryTimeout*time.Second)
defer cancel()

cur, err := collection.Find(ctx, bson.D{})

if err != nil {
 log.Fatalf("Failed to run find query: %v", err)
}
defer cur.Close(ctx)

for cur.Next(ctx) {
 var result bson.M
```

```
  err := cur.Decode(&result)
  log.Printf("Returned: %v", result)

  if err != nil {
   log.Fatal(err)
  }
 }

 if err := cur.Err(); err != nil {
  log.Fatal(err)
 }

}
```

Java

The following code demonstrates how to connect to Amazon DocumentDB using Java when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
package com.example.documentdb;

import com.mongodb.MongoClient;
import com.mongodb.MongoClientURI;
import com.mongodb.ServerAddress;
import com.mongodb.MongoException;
import com.mongodb.client.MongoCursor;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.MongoCollection;
import org.bson.Document;


public final class Main {
    private Main() {
    }
    public static void main(String[] args) {

        String template = "mongodb://%s:%s@%s/sample-database?
replicaSet=rs0&readpreference=%s";
        String username = "sample-user";
        String password = "password";
```

```
        String clusterEndpoint = "sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017";
        String readPreference = "secondaryPreferred";
        String connectionString = String.format(template, username, password,
 clusterEndpoint, readPreference);

        MongoClientURI clientURI = new MongoClientURI(connectionString);
        MongoClient mongoClient = new MongoClient(clientURI);

        MongoDatabase testDB = mongoClient.getDatabase("sample-database");
        MongoCollection<Document> numbersCollection = testDB.getCollection("sample-
collection");

        Document doc = new Document("name", "pi").append("value", 3.14159);
        numbersCollection.insertOne(doc);

        MongoCursor<Document> cursor = numbersCollection.find().iterator();
        try {
            while (cursor.hasNext()) {
                System.out.println(cursor.next().toJson());
            }
        } finally {
            cursor.close();
        }

    }
}
```

C# / .NET

The following code demonstrates how to connect to Amazon DocumentDB using C# / .NET when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
using System;
using System.Text;
using System.Linq;
using System.Collections.Generic;
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
using System.Net.Security;
```

```
using MongoDB.Driver;
using MongoDB.Bson;

namespace CSharpSample
{
   class Program
    {
       static void Main(string[] args)
        {
           string template = "mongodb://{0}:{1}@{2}/sampledatabase?
replicaSet=rs0&readpreference={3}";
           string username = "sample-user";
           string password = "password";
           string clusterEndpoint = "sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017";
           string readPreference = "secondaryPreferred";
           string connectionString = String.Format(template, username, password,
 clusterEndpoint, readPreference);

           var settings = MongoClientSettings.FromUrl(new
 MongoUrl(connectionString));
           var client = new MongoClient(settings);

           var database = client.GetDatabase("sampledatabase");
           var collection =
 database.GetCollection<BsonDocument>("samplecollection");
           var docToInsert = new BsonDocument { { "pi", 3.14159 } };
            collection.InsertOne(docToInsert);
        }
    }
}
```

MongoDB Shell

The following code demonstrates how to connect to and query Amazon DocumentDB using the newest version, mongosh, or the previous mongo shell version, when TLS is disabled.

Connect to Amazon DocumentDB with mongosh

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB.

Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

In the following examples, replace each *user input placeholder* with your cluster's information.

```
mongosh --host cluster-end-point:27017 --username sample-user --password password --
retryWrites false
```

Connect to Amazon DocumentDB with the previous mongo shell version

If you use IAM, you must use a previous version of mongo shell. Enter one of the following command options:

```
mongo --host cluster-end-point:27017 --username sample-user --password password
```

If you are using a version equal to or greater than 4.2, use the following code to connect. Retryable writes are not supported in Amazon DocumentDB. If you are using legacy mongo shell (not mongosh), do not include the `retryWrites=false` command in any code string. By default, retryable writes are disabled. Including `retryWrites=false` might cause a failure in normal read commands.

```
mongo --host cluster-end-point:27017 --username sample-user --password password
```

Test the connection

1. Insert a single document.

   ```
   db.myTestCollection.insertOne({'hello':'Amazon DocumentDB'})
   ```

2. Find the document that was previously inserted.

   ```
   db.myTestCollection.find({'hello':'Amazon DocumentDB'})
   ```

R

The following code demonstrates how to connect to Amazon DocumentDB with R using mongolite (https://jeroen.github.io/mongolite/) when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
#Include the mongolite library.
library(mongolite)

#Create a MongoDB client, open a connection to Amazon DocumentDB as a replica
#   set and specify the read preference as secondary preferred
client <- mongo(url = "mongodb://sample-user:password@sample-
cluster.node.us-east-1.docdb.amazonaws.com:27017/sample-database?
readPreference=secondaryPreferred&replicaSet=rs0")

##Insert a single document
str <- c('{"hello" : "Amazon DocumentDB"}')
client$insert(str)

##Find the document that was previously written
client$find()
```

Ruby

The following code demonstrates how to connect to Amazon DocumentDB with Ruby when TLS is disabled.

In the following example, replace each *user input placeholder* with your cluster's information.

```
require 'mongo'
require 'neatjson'
require 'json'
client_host = 'mongodb://sample-cluster.node.us-east-1.docdb.amazonaws.com:27017'
client_options = {
   database: 'test',
   replica_set: 'rs0',
   read: {:secondary_preferred => 1},
   user: 'sample-user',
   password: 'password',
   retry_writes: false
}

begin
   ##Create a MongoDB client, open a connection to Amazon DocumentDB as a
```

```
   ##   replica set and specify the read preference as secondary preferred
   client = Mongo::Client.new(client_host, client_options)

   ##Insert a single document
   x = client[:test].insert_one({"hello":"Amazon DocumentDB"})

   ##Find the document that was previously written
   result = client[:test].find()

   #Print the document
   result.each do |document|
      puts JSON.neat_generate(document)
   end
end

#Close the connection
client.close
```

Network compression in Amazon DocumentDB

With wire-protocol compression, you can reduce the data that your application sends over the network. Amazon DocumentDB 8.0.1 and later support compression. You get improved throughput for workloads that transfer large documents or large result sets. After you enable compression, the driver and server exchange compressed messages.

When you enable network compression, you trade CPU resources for reduced network bandwidth. Enable compression when your workload is network-bound and transfers large payloads. For small requests and responses, you might not see a net benefit, because the compression overhead can exceed the savings.

Supported compressors

Amazon DocumentDB supports the `snappy` compressor. Amazon DocumentDB doesn't support other compressors that MongoDB drivers offer, such as `zlib` and `zstd`.

If a driver requests a list of compressors, Amazon DocumentDB negotiates only the supported ones. For example, if a driver offers `snappy` and `zstd`, Amazon DocumentDB selects `snappy` and ignores `zstd`. If a driver offers only unsupported compressors, Amazon DocumentDB doesn't compress the connection. In this case, the driver and the server exchange messages without compression.

Enable compression

To enable compression, add the `compressors` option to your connection string and set it to `snappy`. In the following connection string, replace *`username`*, *`password`*, and *`docdb-cluster-endpoint`* with your own values:

```
mongodb://username:password@docdb-cluster-endpoint:27017/?
tls=true&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false&compressors=snappy
```

When you use a MongoDB driver, set the same option through the driver's connection options. Most drivers require an additional client-side dependency to perform snappy compression:

- The Node.js driver requires the `snappy` package.
- The Python driver (PyMongo) requires the `python-snappy` package. On Linux, `python-snappy` also requires the system snappy library (for example, `snappy-devel` through `dnf` or `yum`).

For the exact option name and any required packages, see your driver's documentation.

> **Note**
>
> If the client-side compression dependency is missing, the driver connects successfully. However, it exchanges messages without compression and doesn't return an error. To confirm that compression is active, verify the negotiated compressor as described in Verify compression.

The following Python code example connects to your cluster with snappy compression enabled and uses a `try` and `except` block to handle connection errors:

```
import pymongo

try:
    client = pymongo.MongoClient(
        'mongodb://username:password@docdb-cluster-endpoint:27017/'
        '?tls=true&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false'
        '&compressors=snappy',
        tlsCAFile='global-bundle.pem')

    db = client['sample-database']
```

```
    col = db['sample-collection']
    col.insert_one({'item': 'DocumentDB'})
    print(col.find_one())
except Exception as error:
    print(f'Connection failed: {error}')
```

Verify compression

To confirm that a driver and Amazon DocumentDB negotiated compression, inspect the `compression` field that the `hello` MongoDB wire protocol command returns. When the driver and Amazon DocumentDB negotiate compression, this field lists the agreed compressors.

Connecting to an Amazon DocumentDB cluster from outside an Amazon VPC

Amazon DocumentDB (with MongoDB compatibility) clusters are deployed within an Amazon Virtual Private Cloud (Amazon VPC). They can be accessed directly by Amazon EC2 instances or other AWS services that are deployed in the same Amazon VPC. Additionally, Amazon DocumentDB can be accessed by EC2 instances or other AWS services in different VPCs in the same AWS Region or other Regions via VPC peering.

However, suppose that your use case requires that you (or your application) access your Amazon DocumentDB resources from outside the cluster's VPC. In that case, you can use SSH tunneling (also known as *port forwarding*) to access your Amazon DocumentDB resources.

It is beyond the scope of this topic to discuss SSH tunneling in depth. For more information about SSH tunneling, see the following:

- SSH Tunnel
- SSH Port Forwarding Example, specifically the Local Forwarding section

To create an SSH tunnel, you need an Amazon EC2 instance running in the same Amazon VPC as your Amazon DocumentDB cluster. You can either use an existing EC2 instance in the same VPC as your cluster or create one. For more information, see the topic that is appropriate for your operating system:

- Getting Started with Amazon EC2 Linux Instances
- Getting Started with Amazon EC2 Windows Instances

You might typically connect to an EC2 instance using the following command.

```
ssh -i "ec2Access.pem" ubuntu@ec2-34-229-221-164.compute-1.amazonaws.com
```

If so, you can set up an SSH tunnel to the Amazon DocumentDB cluster `sample-cluster.node.us-east-1.docdb.amazonaws.com` by running the following command on your local computer. The `-L` flag is used for forwarding a local port. When using an SSH tunnel, connect to your cluster using the cluster endpoint and do not attempt to connect in replica set mode (i.e., specifying `replicaSet=rs0` in your connection string) as it will result in an error.

```
ssh -i "ec2Access.pem" -L 27017:sample-cluster.node.us-east-1.docdb.amazonaws.com:27017
 ubuntu@ec2-34-229-221-164.compute-1.amazonaws.com -N
```

After the SSH tunnel is created, any commands that you issue to `localhost:27017` are forwarded to the Amazon DocumentDB cluster `sample-cluster` running in the Amazon VPC. If Transport Layer Security (TLS) is enabled on your Amazon DocumentDB cluster, you need to download the public key for Amazon DocumentDB from https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem . The following operation downloads this file:

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

> **Note**
>
> TLS is enabled by default for new Amazon DocumentDB clusters. However, you can disable it. For more information, see Managing Amazon DocumentDB cluster TLS settings.

To connect to your Amazon DocumentDB cluster from outside the Amazon VPC, use the following command.

```
mongo --sslAllowInvalidHostnames --ssl --sslCAFile global-bundle.pem --username
 <yourUsername> --password <yourPassword>
```

Connecting to Amazon DocumentDB as a replica set

When you're developing against Amazon DocumentDB, connect to your cluster as a replica set and distribute reads to replica instances using the built-in read preference capabilities of your driver.

This section goes deeper into what that means and describes how you can connect to your Amazon DocumentDB cluster as a replica set using Python as an example.

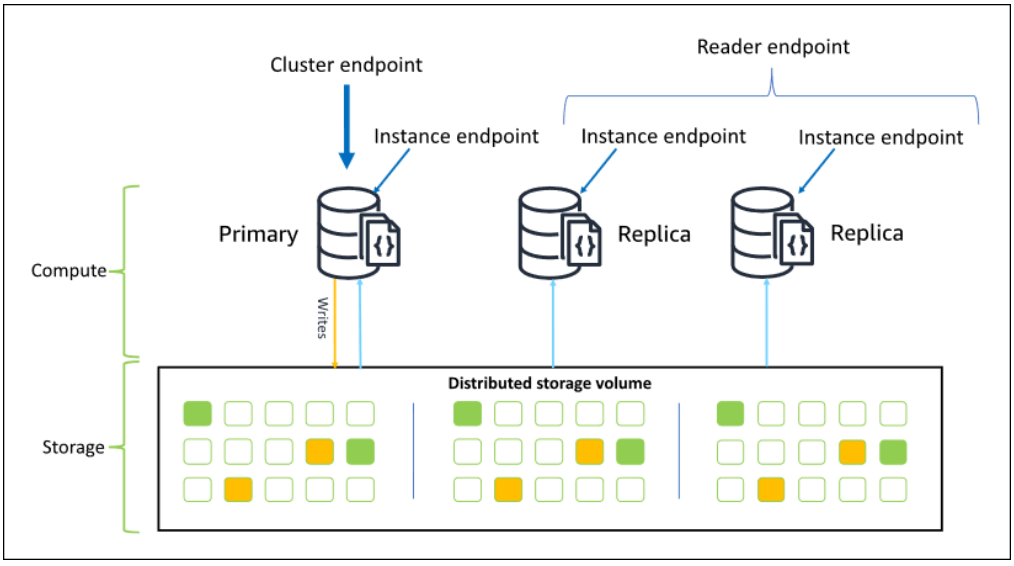
Amazon DocumentDB has three endpoints that you can use to connect to your cluster:

- Cluster endpoint
- Reader endpoint
- Instance endpoints

When connecting to Amazon DocumentDB, use the cluster endpoint. This is a CNAME that points to the primary instance in your cluster, as shown in the following diagram.

When using an SSH tunnel, connect to your cluster using the cluster endpoint. Do not connect in replica set mode (specifying `replicaSet=rs0` in your connection string) as it will result in an error.

> **Note**
>
> For more information about Amazon DocumentDB endpoints, see Amazon DocumentDB endpoints.



Using the cluster endpoint, you can connect to your cluster in replica set mode. You can then use the built-in read preference driver capabilities. In the following example, specifying /?

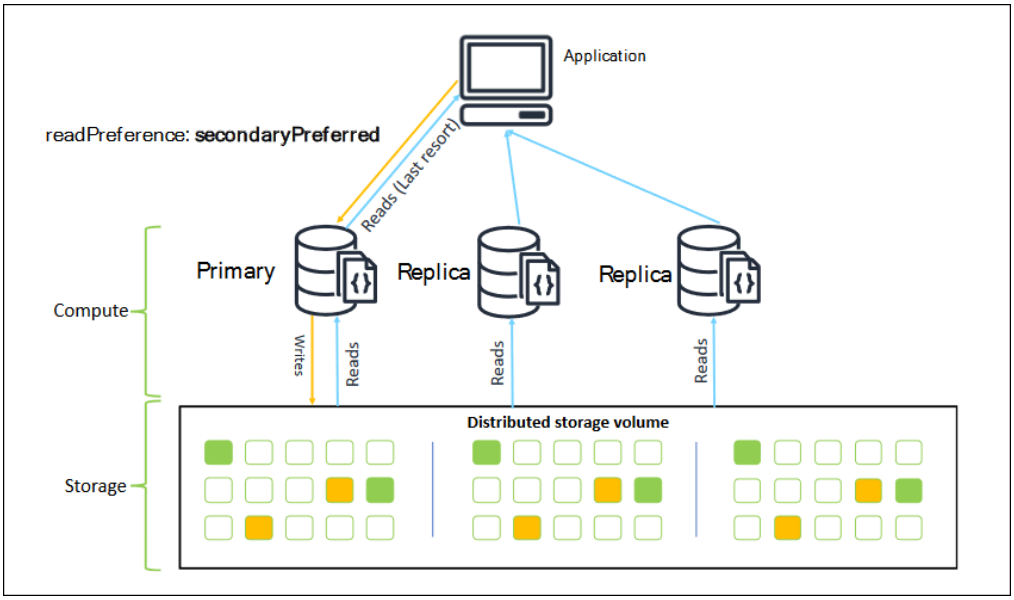
`replicaSet=rs0` signifies to the SDK that you want to connect as a replica set. If you omit `/?replicaSet=rs0'`, the client routes all requests to the cluster endpoint, that is, your primary instance.

```
## Create a MongoDB client, open a connection to Amazon DocumentDB as a
   ## replica set and specify the read preference as secondary preferred
   client = pymongo.MongoClient('mongodb://<user-name>:<password>@mycluster.node.us-
east-1.docdb.amazonaws.com:27017/?replicaSet=rs0')
```

The advantage of connecting as a replica set is that it enables your SDK to discover the cluster topography automatically, including when instances are added or removed from the cluster. You can then use your cluster more efficiently by routing read requests to your replica instances.

When you connect as a replica set, you can specify the `readPreference` for the connection. If you specify a read preference of `secondaryPreferred`, the client routes read queries to your replicas and write queries to your primary instance (as in the following diagram). This is a better use of your cluster resources. For more information, see Read preference options.

```
## Create a MongoDB client, open a connection to Amazon DocumentDB as a
   ##   replica set and specify the read preference as secondary preferred
client = pymongo.MongoClient('mongodb://<user-name>:<password>@mycluster.node.us-
east-1.docdb.amazonaws.com:27017/?replicaSet=rs0&readPreference=secondaryPreferred')
```



Reads from Amazon DocumentDB replicas are eventually consistent. They return the data in the same order as it was written on the primary, and there is often less than a 50 ms replication

lag. You can monitor the replica lag for your cluster using the Amazon CloudWatch metrics `DBInstanceReplicaLag` and `DBClusterReplicaLagMaximum`. For more information, see Monitoring Amazon DocumentDB with CloudWatch.

Unlike traditional monolithic database architecture, Amazon DocumentDB separates storage and compute. Given this modern architecture, we encourage you to read scale on replica instances. Reads on replica instances don't block writes being replicated from the primary instance. You can add up to 15 read replica instances in a cluster and scale out to millions of reads per second.

The key benefit of connecting as a replica set and distributing reads to replicas is that it increases the overall resources in your cluster that are available to do work for your application. As a best practice, connect as a replica set. This is most beneficial in the following scenarios:

- You're using nearly 100 percent CPU on your primary.
- The buffer cache hit ratio is near zero.
- You reach the connection or cursor limits for an individual instance.

Scaling up a cluster instance size is an option, and in some cases, that can be the best way to scale the cluster. But you should also consider how to better use the replicas that you already have in your cluster. This lets you increase scale without the increased cost of using a larger instance type. We also recommend that you monitor and alert on these limits (that is `CPUUtilization`, `DatabaseConnections`, and `BufferCacheHitRatio`) using CloudWatch alarms so that you know when a resource is being heavily used.

For more information, see the following topics:

- Best practices for Amazon DocumentDB
- Amazon DocumentDB quotas

Using cluster connections

Consider the scenario of using all the connections in your cluster. For example, an `r5.2xlarge` instance has a limit of 4,500 connections (and 450 open cursors). If you create a three-instance Amazon DocumentDB cluster and connect only to the primary instance using the cluster endpoint, your cluster limits for open connections and cursors are 4,500 and 450 respectively. You might reach these limits if you're building applications that use many workers that get spun up in containers. The containers open up a number of connections all at once and saturate the cluster.

Instead, you could connect to the Amazon DocumentDB cluster as a replica set and distribute your reads to the replica instances. You could then effectively triple the number of available connections and cursors available in the cluster to 13,500 and 1,350 respectively. Adding more instances to the cluster only increases the number of connections and cursors for read workloads. If you need to increase the number of connections for writes to your cluster, we recommend increasing the instance size.

> **Note**
>
> The number of connections for `large`, `xlarge`, and `2xlarge` instances increases with the instance size up to 4,500. The maximum number of connections per instance for `4xlarge` instances or greater is 4,500. For more information on limits by instance types, see Instance quotas.

Typically we don't recommend that you connect to your cluster using the read preference of `secondary`. This is because if there are no replica instances in your cluster, the reads fail. For example, suppose that you have a two-instance Amazon DocumentDB cluster with one primary and one replica. If the replica has an issue, read requests from a connection pool that is set as `secondary` fail. The advantage of `secondaryPreferred` is that if the client can't find a suitable replica instance to connect to, it falls back to the primary for reads.

Multiple connection pools

In some scenarios, reads in an application need to have read-after-write consistency, which can be served only from the primary instance in Amazon DocumentDB. In these scenarios, you might create two client connection pools: one for writes and one for reads that need read-after-write consistency. To do that, your code would look something like the following.

```
## Create a MongoDB client,
##    open a connection to Amazon DocumentDB as a replica set and specify the
 readPreference as primary
clientPrimary = pymongo.MongoClient('mongodb://<user-
name>:<password>@mycluster.node.us-east-1.docdb.amazonaws.com:27017/?
replicaSet=rs0&readPreference=primary')

## Create a MongoDB client,
##    open a connection to Amazon DocumentDB as a replica set and specify the
 readPreference as secondaryPreferred
```

```
secondaryPreferred = pymongo.MongoClient('mongodb://<user-
name>:<password>@mycluster.node.us-east-1.docdb.amazonaws.com:27017/?
replicaSet=rs0&readPreference=secondaryPreferred')
```

Another option is to create a single connection pool and overwrite the read preference for a given collection.

```
##Specify the collection and set the read preference level for that collection
col = db.review.with_options(read_preference=ReadPreference.SECONDARY_PREFERRED)
```

Summary

To better use the resources in your cluster, connect using replica set mode. If it's suitable for your application, you can read scale your application by distributing your reads to the replica instances.

Connecting to an Amazon DocumentDB cluster from Studio 3T

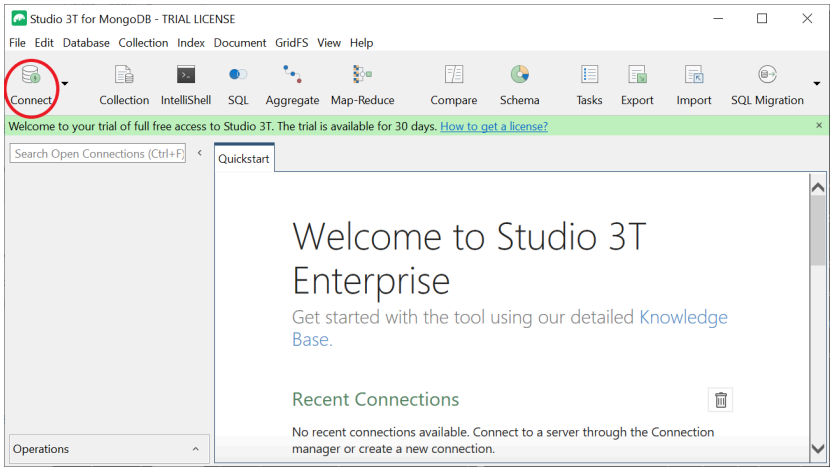
Studio 3T is a popular GUI and IDE for developers and data engineers who work with MongoDB. It offers several powerful capabilities Tree, Table and JSON views of your data, easy import/export in CSV, JSON, SQL and BSON/mongodump, flexible querying option, a visual drag-and-drop UI, a built-in mongo shell with auto-completion, an aggregation pipeline editor, and SQL query support.

Prerequisites

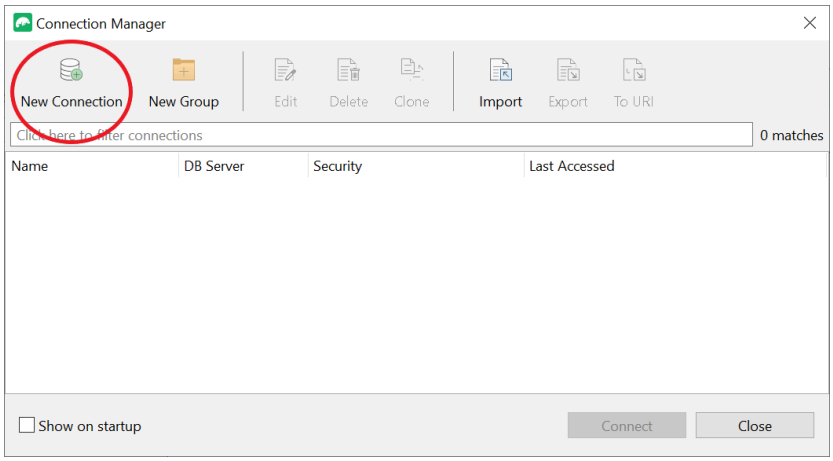
- If you don't already have an Amazon DocumentDB cluster using Amazon EC2 as a bastion/jump host, see Connect using Amazon EC2.
- If you don't have Studio 3T, download and install it.

Connect with Studio 3T

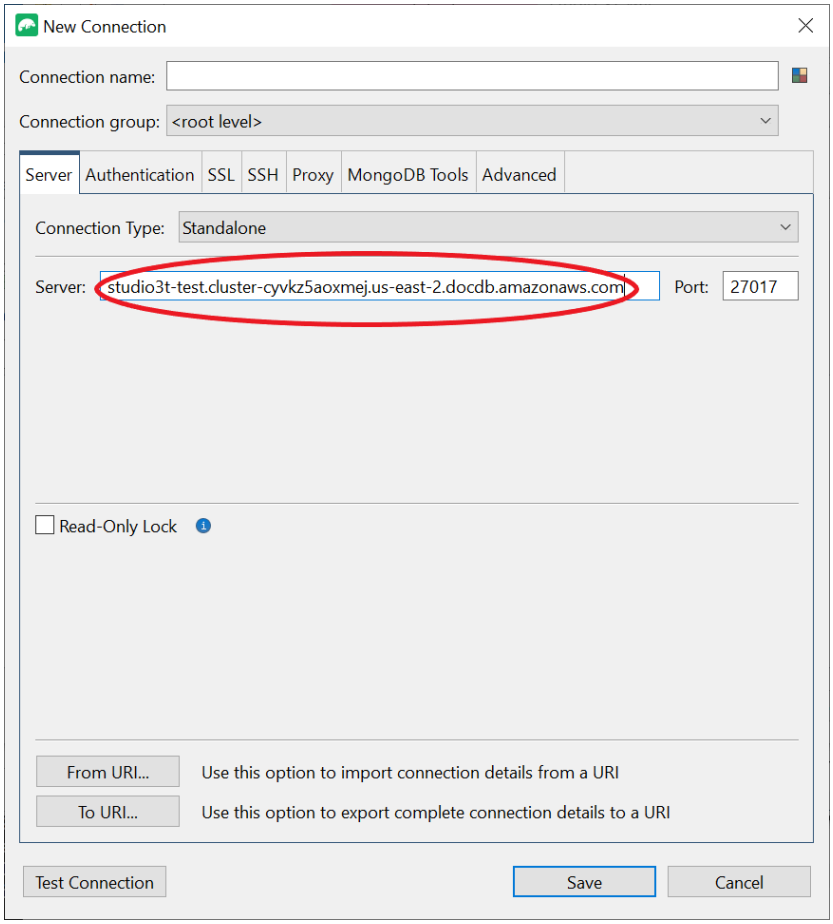
1. Choose **Connect** in the top left corner of the toolbar.



2. Choose **New Connection** in the top left corner of the toolbar.

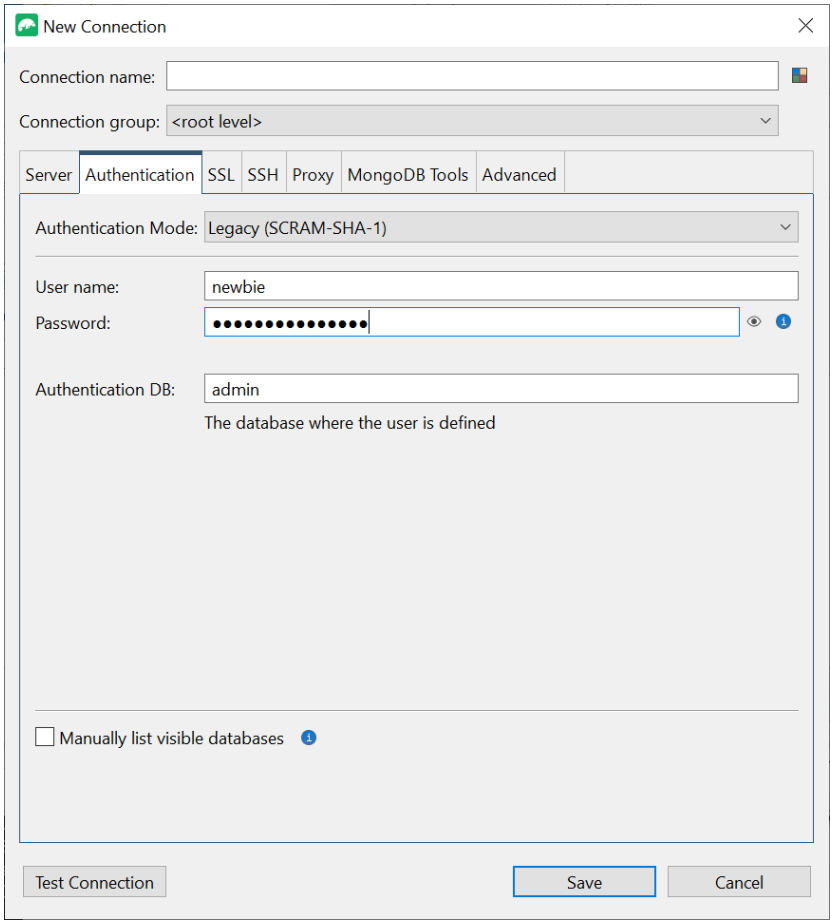


3. On the **Server** tab, in the **Server** field, enter the cluster endpoint information.

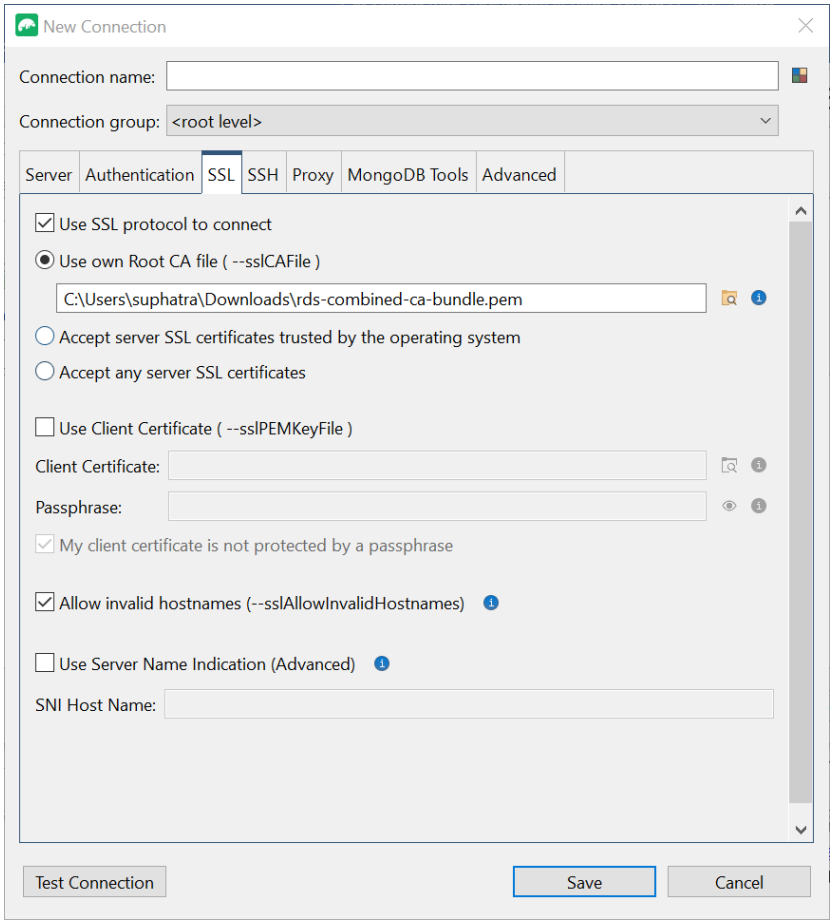


> **Note**
>
> To find your cluster endpoint, see Finding an instance's endpoint.

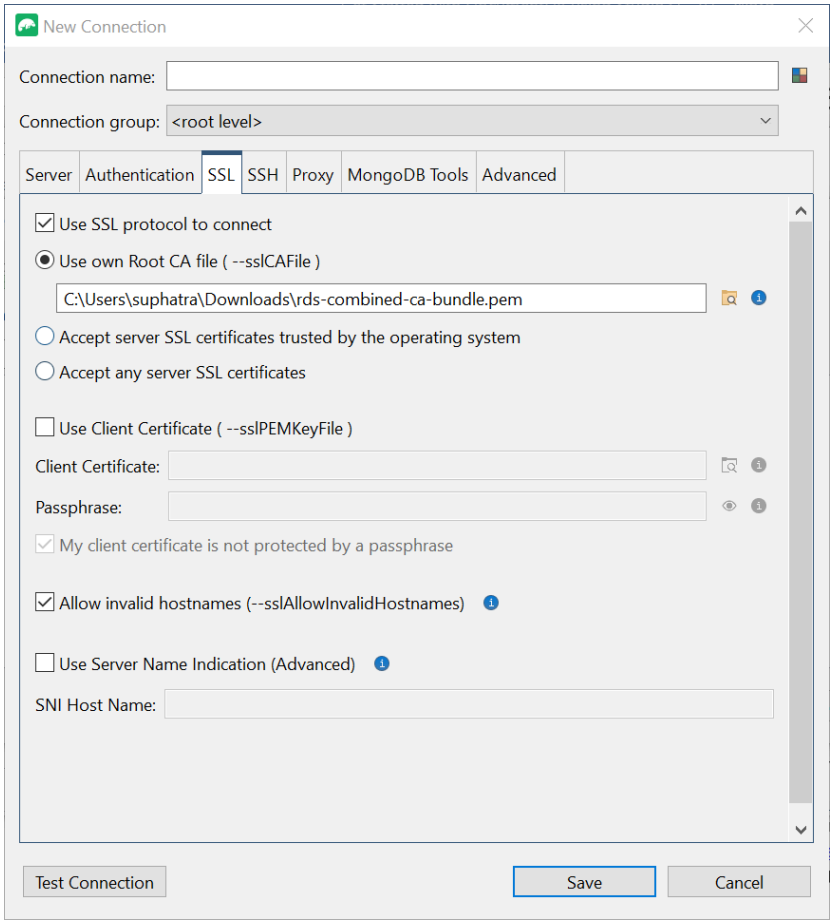
4. Choose the **Authentication** tab and select **Legacy** in the drop down menu for **Authentication Mode**.



5. Input your username and credentials in the **User name** and **Password** fields.
6. Choose the **SSL** tab and check the box **Use SSL protocol to connect**.



7. Choose **Use own Root CA file**. Then add the Amazon DocumentDB certificate (you can skip this step if SSL is disabled on your DocumentDB cluster). Check the box to allow **invalid hostnames**.



> **Note**
>
> Don't have the certificate? You can download it with the following command: `wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem`

8. If you are connecting from a client machine outside the Amazon VPC, you need to create a SSH tunnel. You will do this in the **SSH** tab.

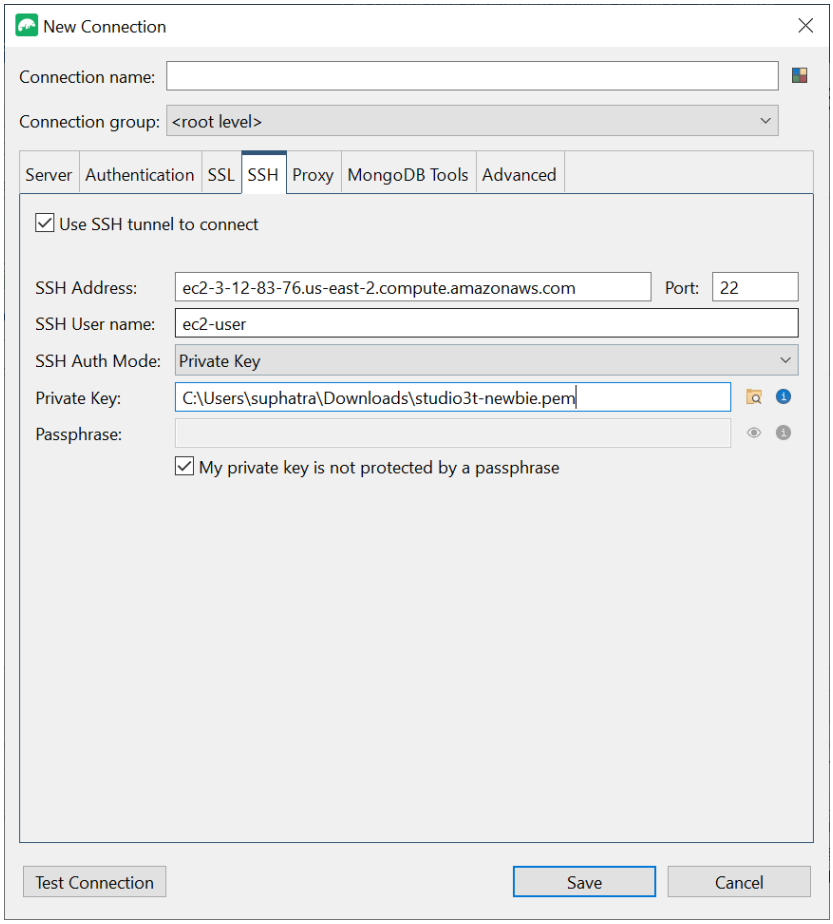
   a. Check the box for **Use SSH tunnel** and input the SSH address in the **SSH Address** field. This is your instance Public DNS (IPV4). You can get this URL from your Amazon EC2 Management Console.

   b. Enter your username. This is the username of your Amazon EC2 instance

   c. For **SSH Auth Mode**, select **Private Key**. In the **Private Key** field, choose the file finder icon to locate and choose the Private key of your Amazon EC2 instance. This is the .pem file (key pair) that you saved while creating your instance in Amazon EC2 Console.

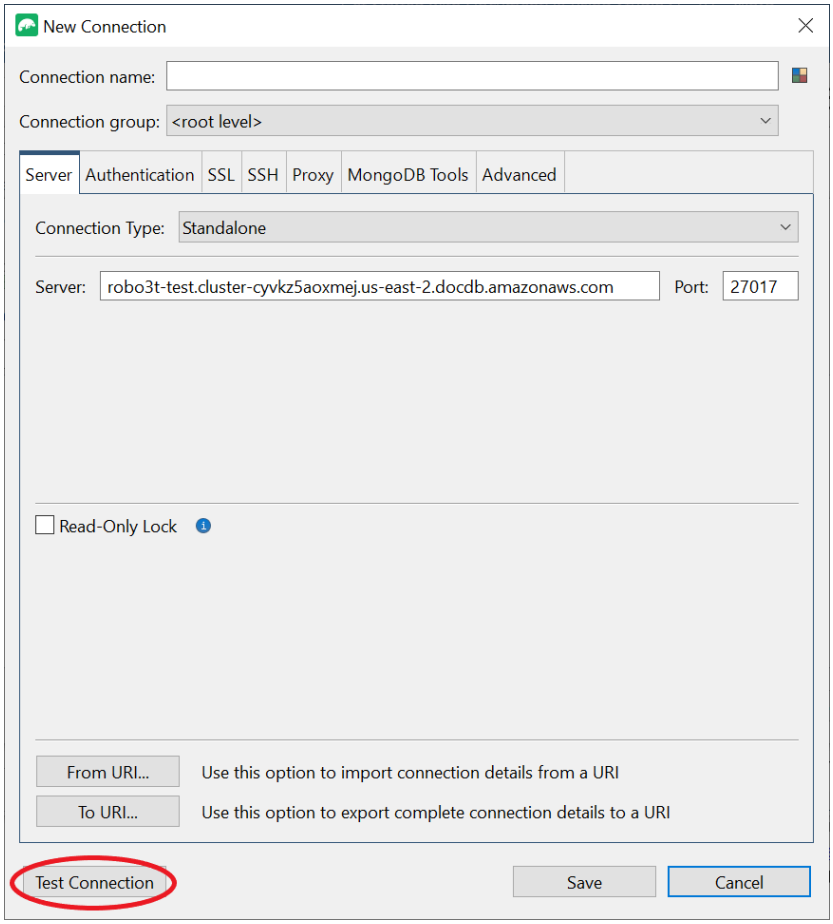
d. If you are on Linux/macOS client machine, you might have to change the permissions of your private key using the following command:

```
chmod 400 /fullPathToYourPemFile/<yourKey>.pem
```

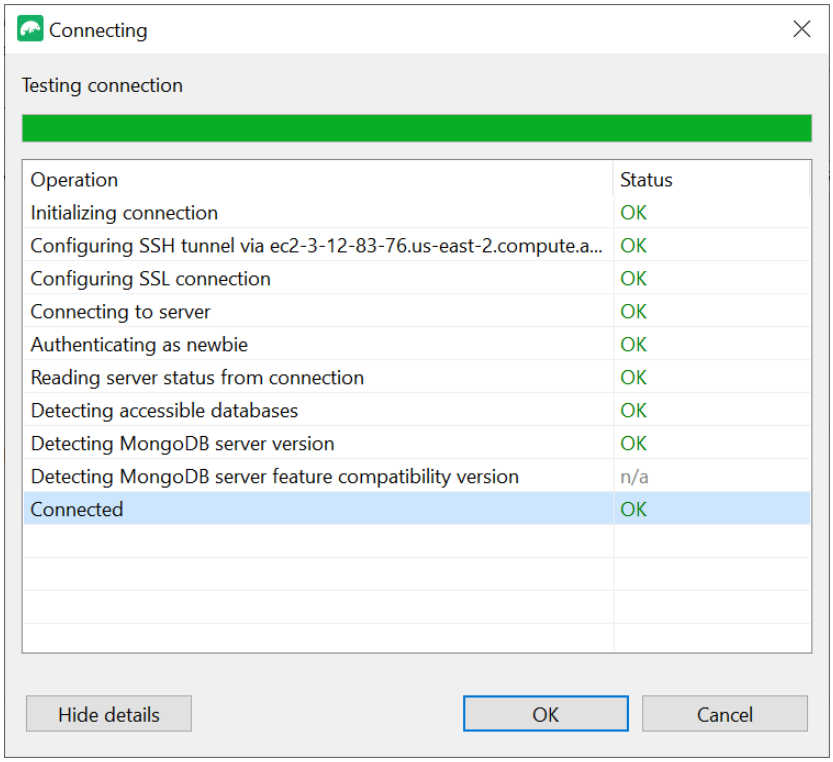


> **Note**
>
> This Amazon EC2 instance should be in the same Amazon VPC and security group as your DocumentDB cluster. You can get the SSH address, username and private key from your Amazon EC2 Management Console.

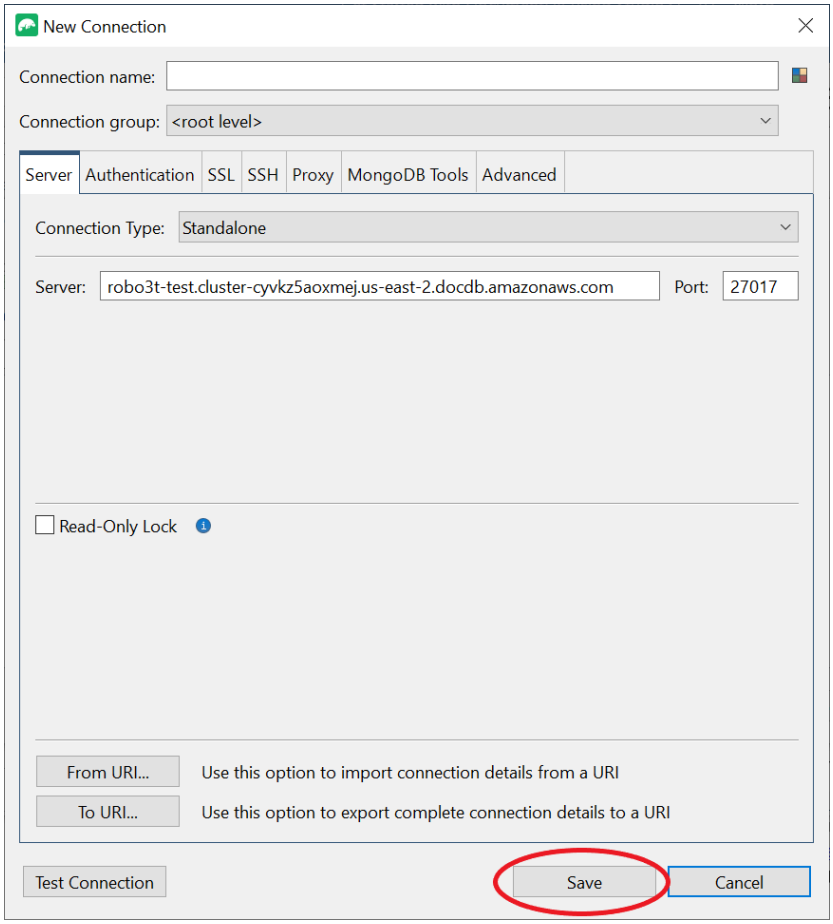
9. Now test your configuration by choosing the **Test connection** button.



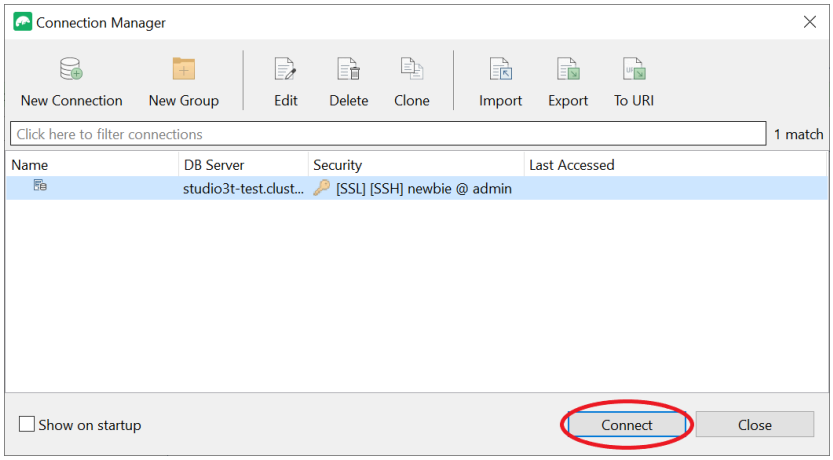
10. A diagnostic window should load a green bar to indicate the test was successful. Now choose **OK** to close out the diagnostic window.



11. Choose **Save** to save your connection for future use.



12. Now select your cluster and choose **Connect**.



Congratulations! You are now successfully connected to your Amazon DocumentDB cluster through Studio 3T.

Connect to Amazon DocumentDB using DataGrip

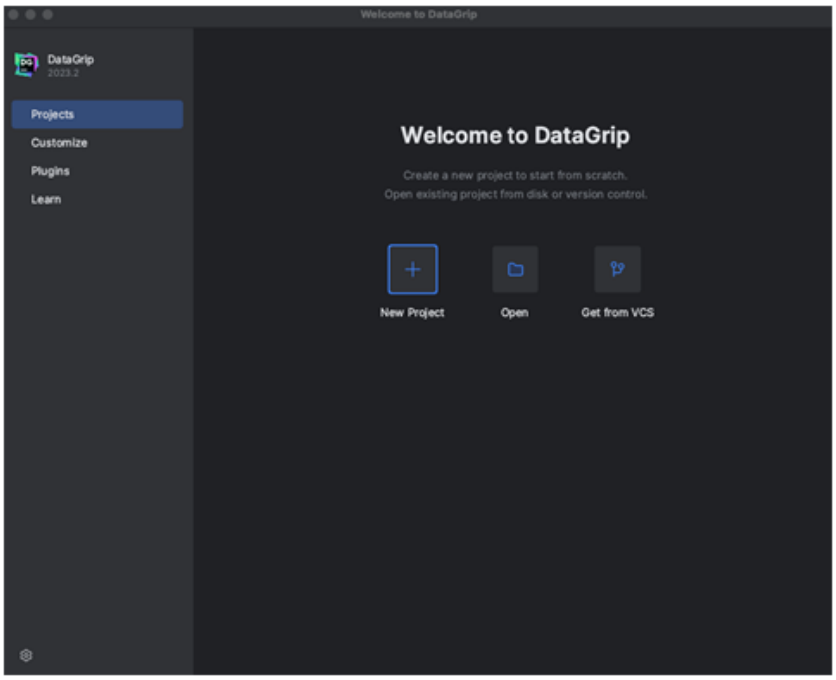
DataGrip is a powerful integrated development environment (IDE) that supports various database systems, including Amazon DocumentDB. This section walks you through the steps to connect to your Amazon DocumentDB cluster using DataGrip, allowing you to easily manage and query your data using a graphical interface.

Prerequisites

- DataGrip IDE installed on your machine. You can download it from JetBrains.
- An Amazon EC2 instance running in the same VPC as your Amazon DocumentDB cluster. You'll use this instance to establish a secure tunnel from your local machine to the Amazon DocumentDBcluster. Follow the instructions on how to Connect using Amazon EC2.
- Alternative to an Amazon EC2 instance, a VPN connection, or if you are already accessing your AWS infrastructure using a secure VPN. If you prefer this option, follow the instructions to securely access Amazon DocumentDB using AWS Client VPN.

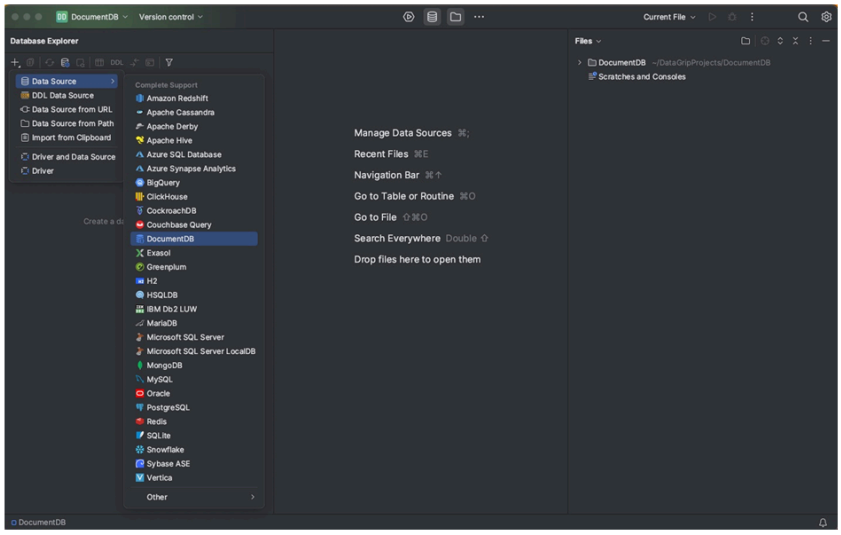
Connect using DataGrip

1. Launch DataGrip on your computer and create a **New Project**.

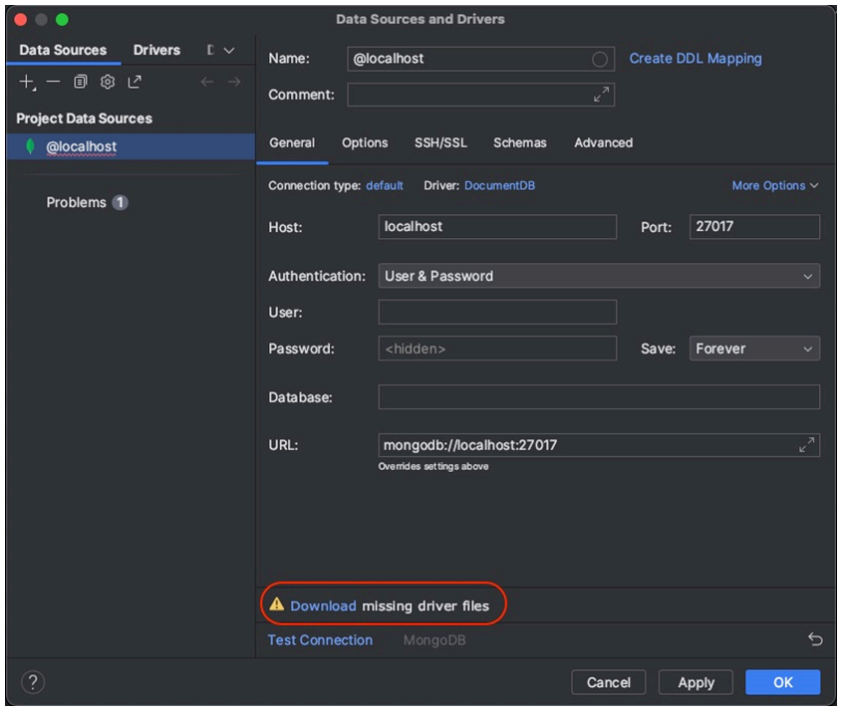


2. Add a new data source using one of the following ways:
   a. From the main menu, navigate to **File – New – Data Source** and select **DocumentDB**

b. In the **Database Explorer**, choose the new icon (**+**) in the toolbar. Navigate to **Data Source** and select **DocumentDB**.



3. On the **Data Sources** page in the **General** tab, check if there is a **Download missing driver files** link at the bottom of the connection settings area. Choose this link to download drivers that are required to interact with a database. For a direct download link, refer to JetBrains JDBC drivers.



4. In the **General** tab, specify the connection details:

   a. In the **Host** field, specify the Amazon DocumentDB cluster endpoint.

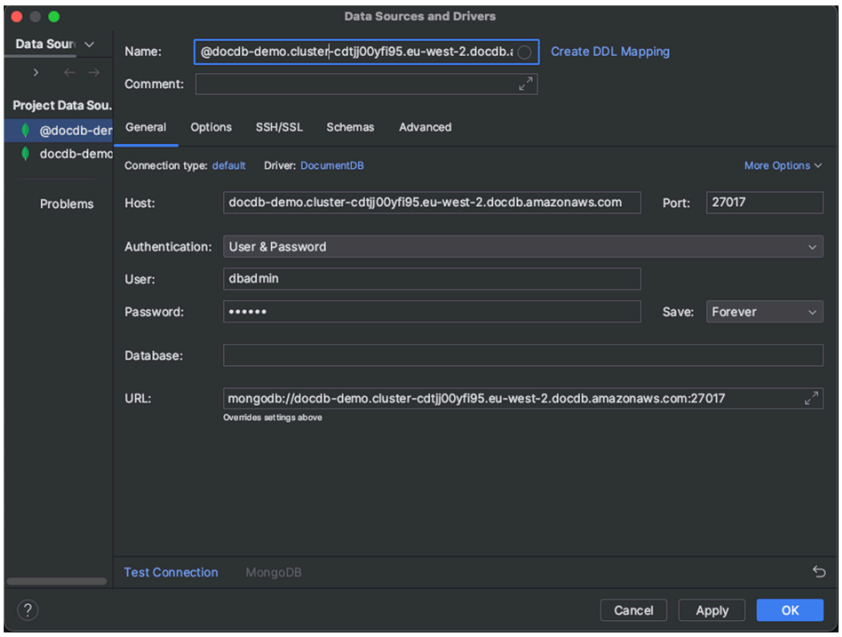
   b. **Port** is already set to 27017. Change it if your cluster was deployed on a different port.

c. For **Authentication**, choose **User & Password**.

d. Enter your user name and password information.

e. The **Database** field is optional. You can specify the database to which you want to connect.

f. The **URL** field auto-completes as you add the preceding details.



5. In the **SSH/SSL** tab, enable **Use SSH tunnel**, then choose the icon to open the **SSH Configuration** dialog. Enter the following information:

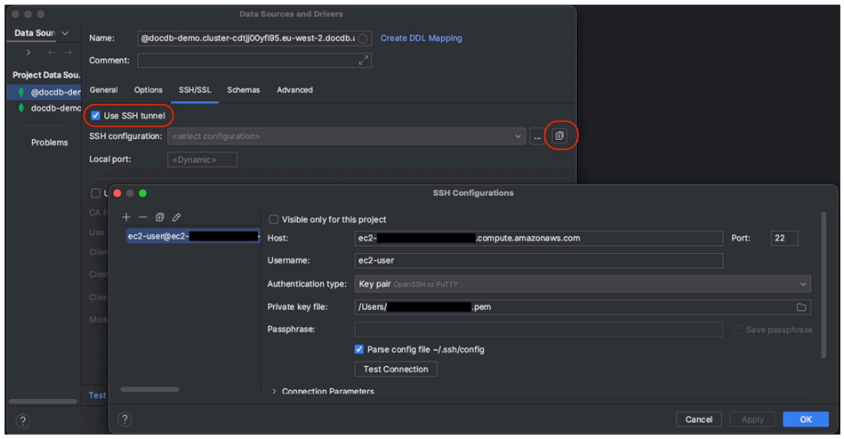
   a. in the **Host** field, enter the hostname of your Amazon EC2 instance.

   b. Enter the username and password for your Amazon EC2 instance.

   c. For **Authentication type**, choose **Key pair**.

   d. Enter your **Private key file**.

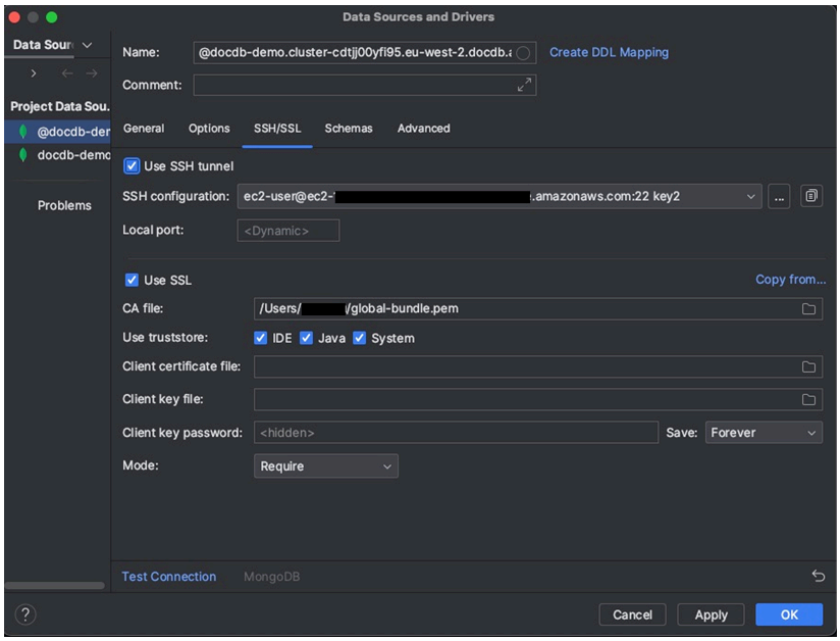
> **Note**
>
> If you're using the VPN option, there is no need to configure the SSH tunnel.



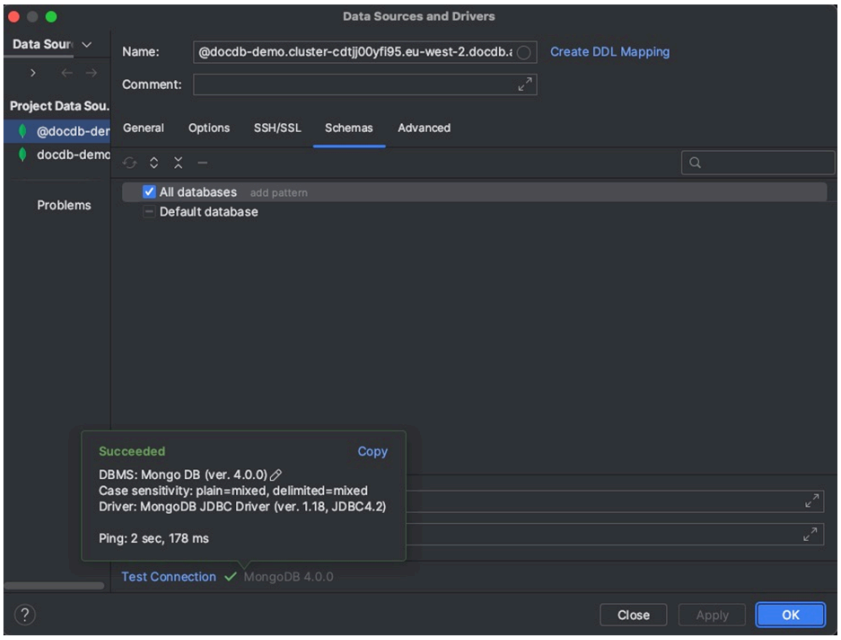
6. In the **SSH/SSL** tab, enable **Use SSL**. In the **CA file** field, enter the location to the `global-bundle.pem` file on your computer. For **Mode**, leave the option **Require**.

> **Note**
>
> You can download the certificate from this location or with this command: wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem

> **Note**
>
> If you are connecting to Amazon DocumentDB elastic cluster, you don't have to specify the CA file. Leave the **Use SSL** option checked and all the other options at their default values.



7. In the **Schemas** tab, choose **All databases** or enter the filter "*:*" in the **Schema pattern** field. Choose **Test Connection** to test the connection.



8. Once the connection is successfully tested, choose **OK** to save the data source configuration.

DataGrip features

DataGrip provides various features to help you work with Amazon DocumentDB efficiently:

- **SQL Editor** — Write and execute SQL-like queries on your DocumentDB collections using the SQL editor in DataGrip.
- **Visual Query Builder** — Use the visual query builder to create queries graphically without writing SQL code.
- **Schema Management** — Easily manage your database schema, including creating, altering, and dropping collections.
- **Data Visualization** — View and analyze your data using various visualization tools available in DataGrip.
- **Export and Import Data** — Transfer data between Amazon DocumentDB and other databases using DataGrip's export and import features.

Refer to the official DataGrip documentation for more advanced features and tips on working with Amazon DocumentDB and other database systems.

Connect using Amazon EC2

This section describes how to set up connectivity between an Amazon DocumentDB cluster and Amazon EC2 and access the Amazon DocumentDB cluster from the Amazon EC2 instance.

There are two options for configuring the EC2 connection:

- Automatically connect your EC2 instance to an Amazon DocumentDB database — Use the automatic connection feature in the EC2 console to automatically configure the connection between your EC2 instance and a new or existing Amazon DocumentDB database. This connection allows traffic to travel between the EC2 instance and the Amazon DocumentDB database. This option is typically used for testing and creating new security groups.
- Manually connect your EC2 instance to your Amazon DocumentDB database — Configure the connection between your EC2 instance to your Amazon DocumentDB database by manually configuring and assigning the security groups to reproduce the configuration that is created by the automatic connection feature. This option is typically used for changing more advanced settings and using exisitng security groups.

Prerequisites

Regardless of the option, and before you create your first Amazon DocumentDB cluster, you must do the following:

Create an Amazon Web Services (AWS) account

Before you can begin using Amazon DocumentDB, you must have an Amazon Web Services (AWS) account. The AWS account is free. You pay only for the services and resources that you use.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://signin.aws.amazon.com/signup?request_type=register.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Optionally, set up the needed AWS Identity and Access Management (IAM) permissions.

Access to manage Amazon DocumentDB resources such as clusters, instances, and cluster parameter groups requires credentials that AWS can use to authenticate your requests. For more information, see Identity and Access Management for Amazon DocumentDB.

1. In the search bar of the AWS Management Console, enter **`IAM`** and select **IAM** in the drop down menu that appears.
2. Once you're in the IAM console, select **Users** from the navigation pane.
3. Select your username.
4. Choose **Add permissions**.
5. Select **Attach existing policies directly**.
6. Type `AmazonDocDBFullAccess` in the search bar and select it once it appears in the search results.
7. Choose **Next: Review**.
8. Choose **Add permissions**.

Create an Amazon Virtual Private Cloud (Amazon VPC)

Depending on which AWS Region you are in, you may or may not have a default VPC already created. If you don't have a default VPC, complete step 1 of the Getting Started with Amazon VPC in the *Amazon VPC User Guide*. This will take less than five minutes.

Connect Amazon EC2 automatically

Topics

- Automatically connect an EC2 instance to a new Amazon DocumentDB database
- Automatically connect an EC2 instance to an existing Amazon DocumentDB database
- Overview of automatic connectivity with an EC2 instance
- Viewing connected compute resources

Before setting up a connection between an EC2 instance and a new Amazon DocumentDB database, make sure you meet the requirements described in Overview of automatic connectivity with an EC2 instance. If you make changes to security groups after you configure connectivity, the changes might affect the connection between the EC2 instance and the Amazon DocumentDB database.

> **Note**
>
> You can only set up a connection between an EC2 instance and an Amazon DocumentDB database automatically by using the AWS Management Console. You can't set up a connection automatically with the AWS CLI or Amazon DocumentDB API.

Automatically connect an EC2 instance to a new Amazon DocumentDB database

The following process assume you have completed the steps in the Prerequisites topic.

Steps

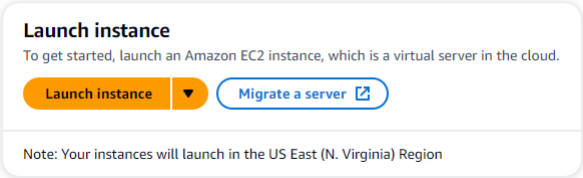
- Step 1: Create an Amazon EC2 instance
- Step 2: Create an Amazon DocumentDB cluster
- Step 3: Connect to your Amazon EC2 instance
- Step 4: Install the MongoDB Shell

- Step 5: Manage Amazon DocumentDB TLS
- Step 6: Connect to your Amazon DocumentDB cluster
- Step 7: Insert and query data
- Step 8: Explore

Step 1: Create an Amazon EC2 instance

In this step, you will create an Amazon EC2 instance in the same Region and Amazon VPC that you will later use to provision your Amazon DocumentDB cluster.

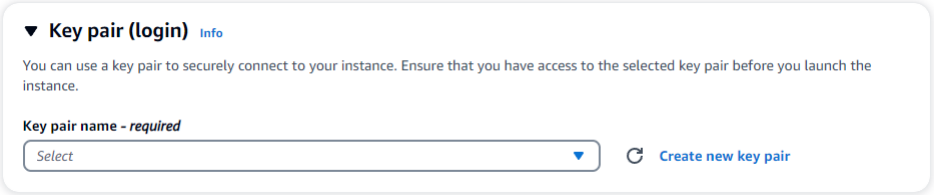
1. On the Amazon EC2 console, choose **Launch instance**.



2. Enter a name or identifier in the **Name** field located in the **Name and tags** section.
3. In the **Amazon Machine Image (AMI)** drop-down list, locate **Amazon Linux 2 AMI** and choose it.



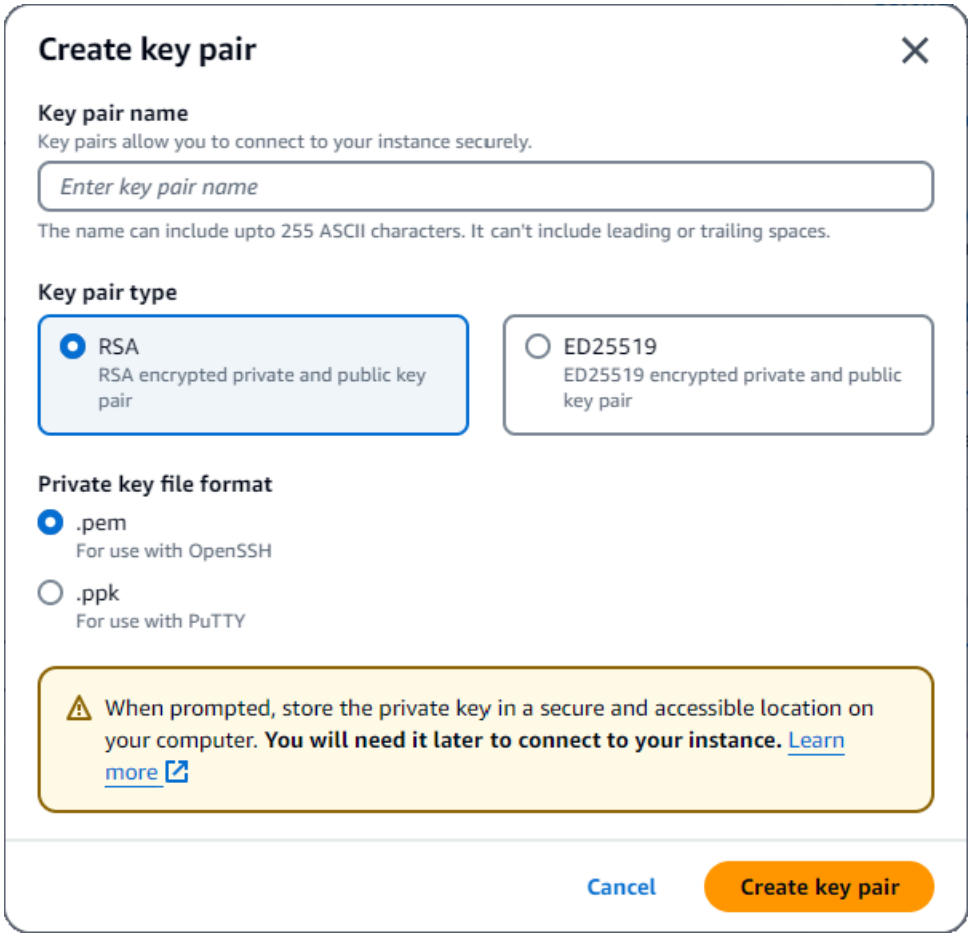
4. Locate and choose **t3.micro** in the **Instance type** drop-down list.
5. In the **Key pair (login)** section, enter the identifier of an existing key-pair, or choose **Create new key pair**.



   You must provide an Amazon EC2 key pair.

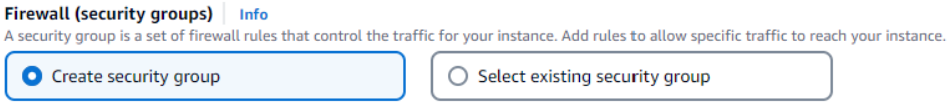
   - If you do have an Amazon EC2 key pair:
     1. Select a key pair, choose your key pair from the list.
     2. You must already have the private key file (.pem or .ppk file) available to log in to your Amazon EC2 instance.

- If you do not have an Amazon EC2 key pair:
  1. Choose **Create new key pair**, the **Create key pair** dialog box appears.
  2. Enter a name in the **Key pair name** field.
  3. Choose the **Key pair type** and **Private key file format**.
  4. Choose **Create key pair**.



> **Note**
>
> For security purposes, we highly recommend using a key-pair for both SSH and internet connectivity to your EC2 instance.

6. **Optional:** In the **Network settings section**, under **Firewall (security groups)**, choose **Create security group**.



Choose **Create security group** (check all the traffic allow rules that apply to your EC2 connectivity).

> **Note**
>
> If you want to use an existing security group, follow the instructions in Connect Amazon EC2 manually.

7. In the **Summary** section, review your EC2 configuration and choose **Launch instance** if correct.

Step 2: Create an Amazon DocumentDB cluster

While the Amazon EC2 instance is being provisioned, create your Amazon DocumentDB cluster.

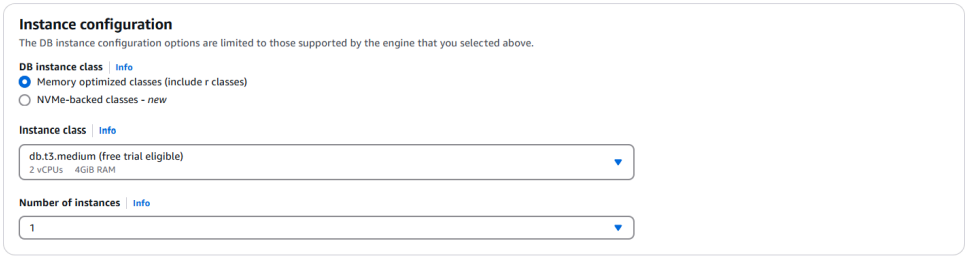
1. Navigate to the Amazon DocumentDB console and choose **Clusters** from the navigation pane.
2. Choose **Create**.
3. Leave the **Cluster type** setting at it's default of **Instance Based Cluster**.
4. In **Cluster configuration**, for **Cluster identifier**, enter a unique name. Note that the console will change all cluster names into lower-case regardless of how they are entered.

   For **Engine version**, keep the default version, or choose another available version. For more information, see Amazon DocumentDB cluster settings.
5. For **Cluster storage configuration**, leave the default setting of **Amazon DocumentDB Standard**.
6. In **Instance configuration**:

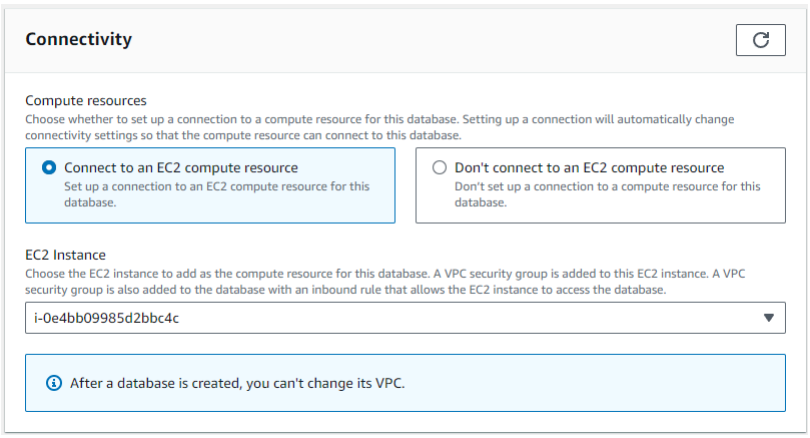
   - For **DB instance class**, choose **Memory optimized classes (include r classes)** (this is default).

     The other instance option is **NVMe-backed classes**. To learn more, see NVMe-backed instances.
   - For **Instance class**, choose the instance type that suits your needs. For a more detailed explanation of instance classes, see Instance class specifications.

- For **number of instances**, choose a number that best reflects your needs. Remember, the lower the number, the lower the cost, and the lower the read/write volume that can be managed by the cluster.



7. For **Connectivity**, choose **Connect to an EC2 compute resource**. This is the EC2 instance you created in Step 1.



> **Note**
>
> Connecting to an EC2 compute resource automatically creates a security group for your EC2 compute resource connection to your Amazon DocumentDB cluster. When you have completed creating your cluster and you want to see the newly created security group, navigate to the cluster list and choose your cluster's identifier. In the **Connectivity & security** tab, go to **Security Groups** and find your group under **Security group name (ID)**. It will contain the name of your cluster and look similar to this: `docdb-ec2-docdb-2023-12-11-21-33-41:i-0e4bb09985d2bbc4c (sg-0238e0b0bf0f73877)`.

8. In the **Authentication** section, enter a username for the primary user, and then choose **Self managed**. Enter a password, then confirm it.

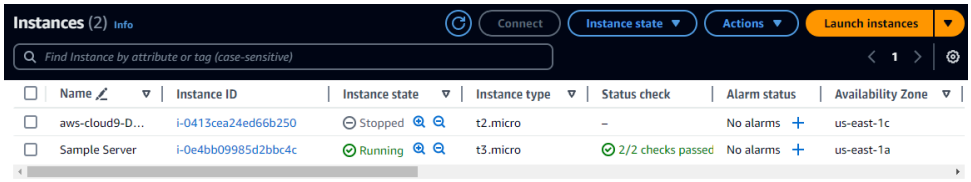
If you instead chose **Managed in AWS Secrets Manager**, see [Password management with Amazon DocumentDB and AWS Secrets Manager] for more information.

9. Choose **Create cluster**.

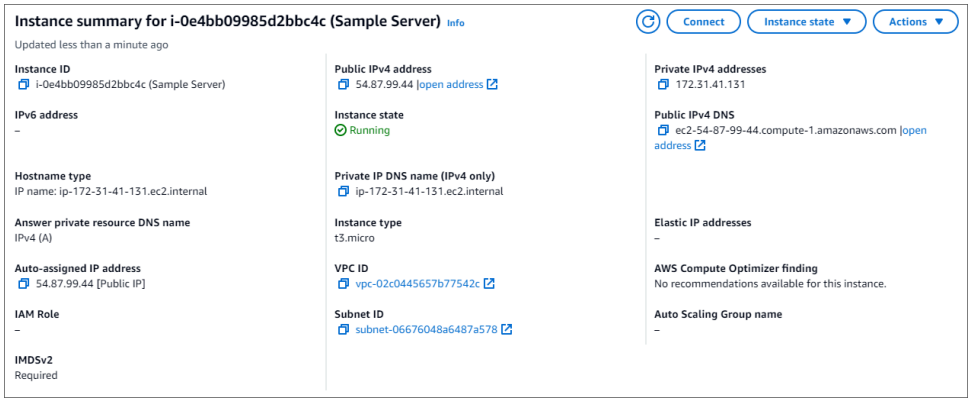
Step 3: Connect to your Amazon EC2 instance

To install the mongo shell, you must first connect to your Amazon EC2 instance. Installing the mongo shell enables you to connect to and query your Amazon DocumentDB cluster. Complete the following steps:

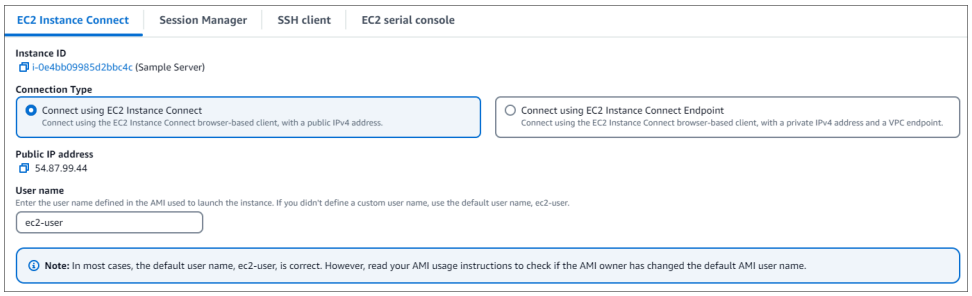
1. On the Amazon EC2 console, navigate to your instances and see if the instance you just created is running. If it is, select the instance by clicking the instance ID.



2. Choose **Connect**.



3. There are four tabbed options for your connection method: Amazon EC2 Instance Connect, Session Manager, SSH client, or EC2 serial console. You must choose one and follow its instructions. When complete, choose **Connect**.



> **Note**
>
> If your IP address changed after you started this walk-through, or you are coming back to your environment at a later time, you must update your demoEC2 security group inbound rule to enable inbound traffic from your new API address.

Step 4: Install the MongoDB Shell

You can now install the MongoDB shell, which is a command-line utility that you use to connect and query your Amazon DocumentDB cluster. There are currently two versions of MongoDB shell: the newest version, mongosh, and the previous version, mongo shell.

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

Follow the instructions below to install the MongoDB shell for your operating system.

On Amazon Linux

To install the MongoDB shell on Amazon Linux

If you are not using IAM authentication and want to use the latest MongoDB shell (mongosh) to connect to your Amazon DocumentDB cluster, follow these steps:

1. Create the repository file. At the command line of your EC2 instance you created, execute the follow command:

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2023/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://pgp.mongodb.com/server-5.0.asc" | sudo tee /etc/yum.repos.d/
mongodb-org-5.0.repo
```

2. When it is complete, install mongosh with one of the two following command options at the command prompt:

Option 1 — If you chose the default Amazon Linux 2023 during the Amazon EC2 configuration, enter this command:

```
sudo yum install -y mongodb-mongosh-shared-openssl3
```

Option 2 — If you chose Amazon Linux 2 during the Amazon EC2 configuration, enter this command:

```
sudo yum install -y mongodb-mongosh
```

If you are using IAM authentication, you must use the previous version of the MongoDB shell (5.0) to connect to your Amazon DocumentDB cluster, follow these steps:

1. Create the repository file. At the command line of your EC2 instance you created, execute the follow command:

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2023/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://pgp.mongodb.com/server-5.0.asc" | sudo tee /etc/yum.repos.d/
mongodb-org-5.0.repo
```

2. When it is complete, install the mongodb 5.0 shell with the following command option at the command prompt:

```
sudo yum install -y mongodb-org-shell
```

On Ubuntu

To install mongosh on Ubuntu

1. Import the public key that will be used by the package management system.

```
curl -fsSL https://pgp.mongodb.com/server-5.0.asc | sudo gpg --dearmor -o /usr/
share/keyrings/mongodb-server-5.0.gpg
```

2. Create the list file `mongodb-org-5.0.list` for MongoDB using the command appropriate for your version of Ubuntu.

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-
server-5.0.gpg ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0
 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list
```

3. Import and update the local package database using the following command:

```
sudo apt-get update
```

4. Install mongosh.

```
sudo apt-get install -y mongodb-mongosh
```

For information about installing earlier versions of MongoDB on your Ubuntu system, see Install MongoDB Community Edition on Ubuntu.

On other operating systems

To install the mongo shell on other operating systems, see Install MongoDB Community Edition in the MongoDB documentation.

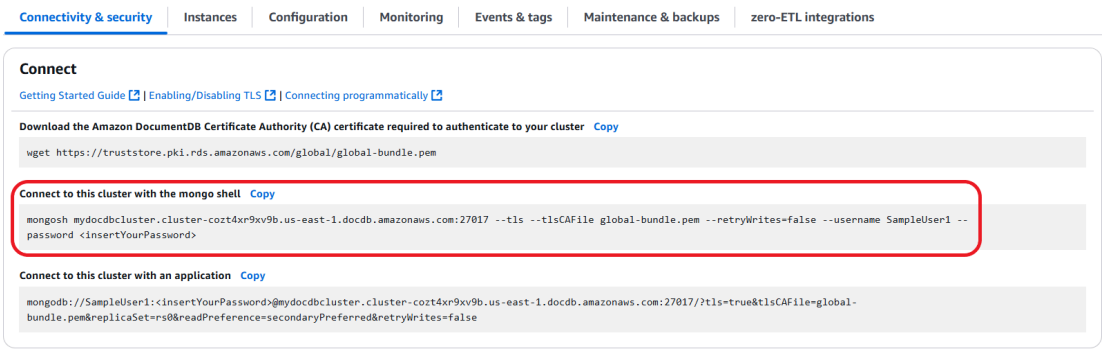
Step 5: Manage Amazon DocumentDB TLS

Download the CA certificate for Amazon DocumentDB with the following code: `wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem`

> **Note**
>
> Transport Layer Security (TLS) is enabled by default for any new Amazon DocumentDB clusters. For more information, see Encrypting data in transit.

Step 6: Connect to your Amazon DocumentDB cluster

1. On the Amazon DocumentDB console, under **Clusters**, locate your cluster. Choose the cluster you created by clicking the **Cluster identifier** for that cluster.
2. In the **Connectivity and security** tab, locate **Connect to this cluster with the mongo shell** in the **Connect** box:



Copy the connection string provided and paste it into your terminal.

Make the following changes to it:

a. Make sure you have the correct username in the string.

b. Omit `<insertYourPassword>` so that you are prompted for the password by the mongo shell when you connect.

c. Optional: If you are using IAM authentication, or are using the previous version of the MongoDB shell, modify your connection string as follows:

```
mongo --ssl --host mydocdbcluster.cluster-cozt4xr9xv9b.us-east-1.docdb.amazonaws.com:27017 --sslCAFile global-bundle.pem --username SampleUser1 --password
```

Replace `mydocdbcluster.cluster-cozt4xr9xv9b.us-east-1` with the same information from your cluster.

3. Press enter in your terminal. You are now be prompted for your password. Enter your password.
4. When you enter your password and can see the `rs0 [direct: primary] <env-name>>` prompt, you are successfully connected to your Amazon DocumentDB cluster.

To solve connection problems, see Troubleshooting Amazon DocumentDB.

Step 7: Insert and query data

Now that you are connected to your cluster, you can run a few queries to get familiar with using a document database.

1. To insert a single document, enter the following:

```
db.collection.insertOne({"hello":"DocumentDB"})
```

You get the following output:

```
{
  acknowledged: true,
  insertedId: ObjectId('673657216bdf6258466b128c')
}
```

2. You can read the document that you wrote with the `findOne()` command (because it only returns a single document). Input the following:

```
db.collection.findOne()
```

You get the following output:

```
{ "_id" : ObjectId("5e401fe56056fda7321fbd67"), "hello" : "DocumentDB" }
```

3. To perform a few more queries, consider a gaming profiles use case. First, insert a few entries into a collection titled `profiles`. Input the following:

```
db.profiles.insertMany([{ _id: 1, name: 'Matt', status: 'active', level: 12, score:
 202 },
      { _id: 2, name: 'Frank', status: 'inactive', level: 2, score: 9 },
      { _id: 3, name: 'Karen', status: 'active', level: 7, score: 87 },
      { _id: 4, name: 'Katie', status: 'active', level: 3, score: 27 }
])
```

You get the following output:

```
{ acknowledged: true, insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4 } }
```

4. Use the `find()` command to return all the documents in the profiles collection. Input the following:

```
db.profiles.find()
```

You will get an output that will match the data you typed in Step 3.

5. Use a query for a single document using a filter. Input the following:

```
db.profiles.find({name: "Katie"})
```

   You get the following output:

```
{ "_id" : 4, "name" : "Katie", "status": "active", "level": 3, "score":27}
```

6. Now let's try to find a profile and modify it using the `findAndModify` command. We'll give the user Matt an extra 10 points with the following code:

```
db.profiles.findAndModify({
        query: { name: "Matt", status: "active"},
        update: { $inc: { score: 10 } }
    })
```

   You get the following output (note that his score hasn't increased yet):

```
{
    [{_id : 1, name : 'Matt', status: 'active', level: 12, score: 202}]
```

7. You can verify that his score has changed with the following query:

   ```
   db.profiles.find({name: "Matt"})
   ```

   You get the following output:

```
{ "_id" : 1, "name" : "Matt", "status" : "active", "level" : 12, "score" : 212 }
```

Step 8: Explore

Congratulations! You have successfully completed the Quick Start Guide to Amazon DocumentDB.

What's next? Learn how to fully leverage this powerful database with some of its popular features:

- Managing Amazon DocumentDB clusters and instances
- Scaling
- Backing up and restoring

> **Note**
>
> To save on cost, you can either stop your Amazon DocumentDB cluster to reduce costs or delete the cluster. By default, after 30 minutes of inactivity, your AWS Cloud9 environment will stop the underlying Amazon EC2 instance.

Automatically connect an EC2 instance to an existing Amazon DocumentDB database

The following procedure assumes you have an existing Amazon DocumentDB cluster and an existing Amazon EC2 instance.

Access your Amazon DocumentDB cluster and set up the Amazon EC2 connection

1. Access your Amazon DocumentDB cluster.

   a. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

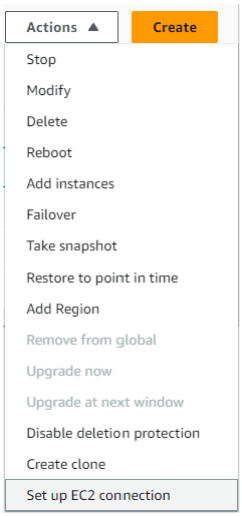
   b. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

   c. Specify the cluster that you want by choosing the button to the left of the cluster's name.

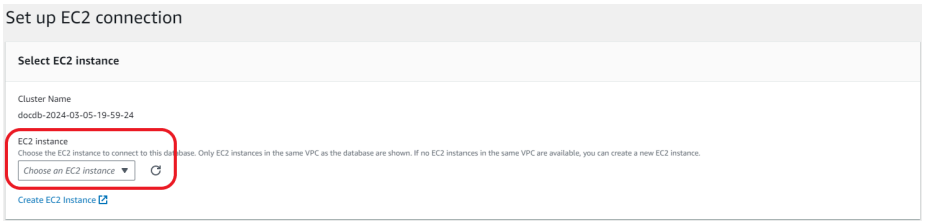
2. Set up the Amazon EC2 connection.

   a. Choose **Actions**, and then choose **Set up EC2 connection**.



The **Set up EC2 connection** dialog appears.

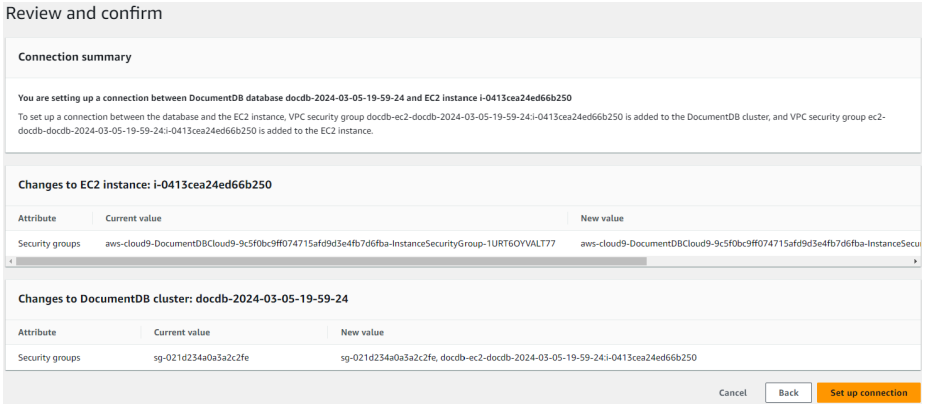
b. In the **EC2 instance** field, choose the EC2 instance you want connected to your cluster.



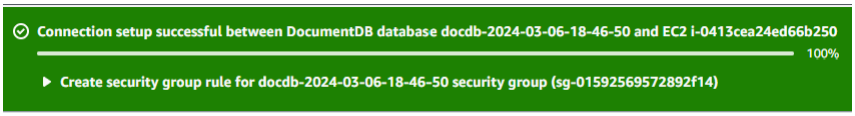
c. Choose **Continue**.

The **Review and confirm** dialog appears.

d. Make sure the changes are correct. Then choose **Set up connection**.



If successful, the following verification appears:



Overview of automatic connectivity with an EC2 instance

When you set up a connection between an EC2 instance and an Amazon DocumentDB database, Amazon DocumentDB automatically configures the VPC security group for your EC2 instance and for your Amazon DocumentDB database.

The following are requirements for connecting an EC2 instance with an Amazon DocumentDB database:

- The EC2 instance must exist in the same VPC as the Amazon DocumentDB database.

  If no EC2 instances exist in the same VPC, then the console provides a link to create one.
- The user who sets up connectivity must have permissions to perform the following Amazon EC2 operations:
  - `ec2:AuthorizeSecurityGroupEgress`
  - `ec2:AuthorizeSecurityGroupIngress`
  - `ec2:CreateSecurityGroup`
  - `ec2:DescribeInstances`
  - `ec2:DescribeNetworkInterfaces`
  - `ec2:DescribeSecurityGroups`
  - `ec2:ModifyNetworkInterfaceAttribute`
  - `ec2:RevokeSecurityGroupEgress`

If the DB instance and EC2 instance are in different Availability Zones, your account may incur cross-Availability Zone costs.

When you set up a connection to an EC2 instance, Amazon DocumentDB acts according to the current configuration of the security groups associated with the Amazon DocumentDB database and EC2 instance, as described in the following table:

Current Amazon DocumentDB security group configuration	Current EC2 security group configuration	Amazon DocumentDB action
There are one or more security groups associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n` . A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	There are one or more security groups associated with the EC2 instance with a name that matches the pattern `DocumentDB-ec2-n` (where n is a number). A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with the VPC security group of the Amazon DocumentDB database as the source.	Amazon DocumentDB takes no action. A connection was already configured automatically between the EC2 instance and the Amazon DocumentDB database. Because a connection already exists between the EC2 instance and the Amazon DocumentDB database, the security groups aren't modified.
Either of the following conditions apply: • There is no security group associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n`. • There are one or more security groups associated with the Amazon DocumentDB with a name that matches the pattern `DocumentDB-ec2-n`. However, Amazon DocumentDB can't use any of these security groups for the connection with	Either of the following conditions apply: • There is no security group associated with the EC2 instance with a name that matches the pattern `ec2-DocumentDB-n` . • There are one or more security groups associated with the EC2 instance with a name that matches the pattern `ec2-DocumentDB-n` . However, Amazon DocumentDB can't use any of these security groups for the connection with the Amazon DocumentDB database.	Amazon DocumentDB action: create new security groups

Current Amazon DocumentDB security group configuration	Current EC2 security group configuration	Amazon DocumentDB action
the EC2 instance. Amazon DocumentDB can't use a security group that doesn't have one inbound rule with the VPC security group of the EC2 instance as the source. Amazon DocumentDB also can't use a security group that has been modified. Examples of modifications include adding a rule or changing the port of an existing rule.	Amazon DocumentDB can't use a security group that doesn't have one outbound rule with the VPC security group of the Amazon DocumentDB database as the source. Amazon DocumentDB also can't use a security group that has been modified.	
There are one or more security groups associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n` . A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	There are one or more security groups associated with the EC2 instance with a name that matches the pattern `ec2-DocumentDB-n` . However, Amazon DocumentDB can't use any of these security groups for the connection with the Amazon DocumentDB database. Amazon DocumentDB can't use a security group that doesn't have one outbound rule with the VPC security group of the Amazon DocumentDB database as the source. Amazon DocumentDB also can't use a security group that has been modified.	Amazon DocumentDB action: create new security groups

Current Amazon DocumentDB security group configuration	Current EC2 security group configuration	Amazon DocumentDB action
There are one or more security groups associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n` . A security group that matches the pattern hasn't been modified. This security group has only one inbound rule with the VPC security group of the EC2 instance as the source.	A valid EC2 security group for the connection exists, but it is not associated with the EC2 instance. This security group has a name that matches the pattern `DocumentDB-ec2-n`. It hasn't been modified. It has only one outbound rule with the VPC security group of the Amazon DocumentDB database as the source.	Amazon DocumentDB action: associate EC2 security group

Current Amazon DocumentDB security group configuration	Current EC2 security group configuration	Amazon DocumentDB action
Either of the following conditions apply: • There is no security group associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n`. • There are one or more security groups associated with the Amazon DocumentDB database with a name that matches the pattern `DocumentDB-ec2-n`. However, Amazon DocumentDB can't use any of these security groups for the connection with the EC2 instance. Amazon DocumentDB can't use a security group that doesn't have one inbound rule with the VPC security group of the EC2 instance as the source. Amazon DocumentDB also can't use security group that has been modified.	There are one or more security groups associated with the EC2 instance with a name that matches the pattern `DocumentDB-ec2-n`. A security group that matches the pattern hasn't been modified. This security group has only one outbound rule with the VPC security group of the Amazon DocumentDB database as the source.	Amazon DocumentDB action: create new security groups

Amazon DocumentDB action: create new security groups

Amazon DocumentDB takes the following actions:

- Creates a new security group that matches the pattern `DocumentDB-ec2-n`. This security group has an inbound rule with the VPC security group of the EC2 instance as the source. This security group is associated with the Amazon DocumentDB database and allows the EC2 instance to access the Amazon DocumentDB database.
- Creates a new security group that matches the pattern `ec2-DocumentDB-n`. This security group has an outbound rule with the VPC security group of the Amazon DocumentDB database as the source. This security group is associated with the EC2 instance and allows the EC2 instance to send traffic to the Amazon DocumentDB database.

Amazon DocumentDB action: associate EC2 security group

Amazon DocumentDB associates the valid, existing EC2 security group with the EC2 instance. This security group allows the EC2 instance to send traffic to the Amazon DocumentDB database.

Viewing connected compute resources

You can use the AWS Management Console to view the compute resources that are connected to an Amazon DocumentDB database. The resources shown include compute resource connections that were set up automatically. You can set up connectivity with compute resources automatically in the following ways:

- You can select the compute resource when you create the database. For more information, see Creating an Amazon DocumentDB cluster and Creating a Multi-AZ DB cluster.
- You can set up connectivity between an existing database and a compute resource. For more information, see Connect Amazon EC2 automatically.

The listed compute resources don't include ones that were connected to the database manually. For example, you can allow a compute resource to access a database manually by adding a rule to the VPC security group associated with the database.

For a compute resource to be listed, the following conditions must apply:

- The name of the security group associated with the compute resource matches the pattern `ec2-DocumentDB-n` (where n is a number).
- The security group associated with the compute resource has an outbound rule with the port range set to the port that the Amazon DocumentDB database uses.

- The security group associated with the compute resource has an outbound rule with the source set to a security group associated with the Amazon DocumentDB database.
- The name of the security group associated with the Amazon DocumentDB database matches the pattern `DocumentDB-ec2-n` (where n is a number).
- The security group associated with the Amazon DocumentDB database has an inbound rule with the port range set to the port that the Amazon DocumentDB database uses.
- The security group associated with the Amazon DocumentDB database has an inbound rule with the source set to a security group associated with the compute resource.

To view compute resources connected to an Amazon DocumentDB database

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Databases**, and then choose the name of the Amazon DocumentDB database.
3. On the **Connectivity & security** tab, view the compute resources in the **Connected compute resources** section.

Connect Amazon EC2 manually

Topics

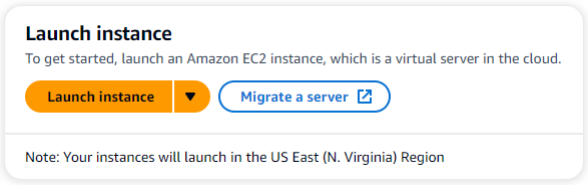
- Step 1: Create an Amazon EC2 instance
- Step 2: Create a security group
- Step 3: Create an Amazon DocumentDB cluster
- Step 4: Connect to your Amazon EC2 instance
- Step 5: Install the MongoDB Shell
- Step 6: Manage Amazon DocumentDB TLS
- Step 7: Connect to your Amazon DocumentDB cluster
- Step 8: Insert and query data
- Step 9: Explore

The following steps assume you have completed the steps in the Prerequisites topic.

Step 1: Create an Amazon EC2 instance

In this step, you will create an Amazon EC2 instance in the same Region and Amazon VPC that you will later use to provision your Amazon DocumentDB cluster.

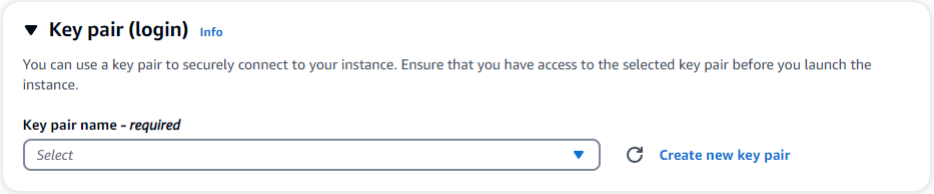
1. On the Amazon EC2 console, choose **Launch instance**.



2. Enter a name or identifier in the **Name** field located in the **Name and tags** section.
3. In the **Amazon Machine Image (AMI)** drop-down list, locate **Amazon Linux 2 AMI** and choose it.

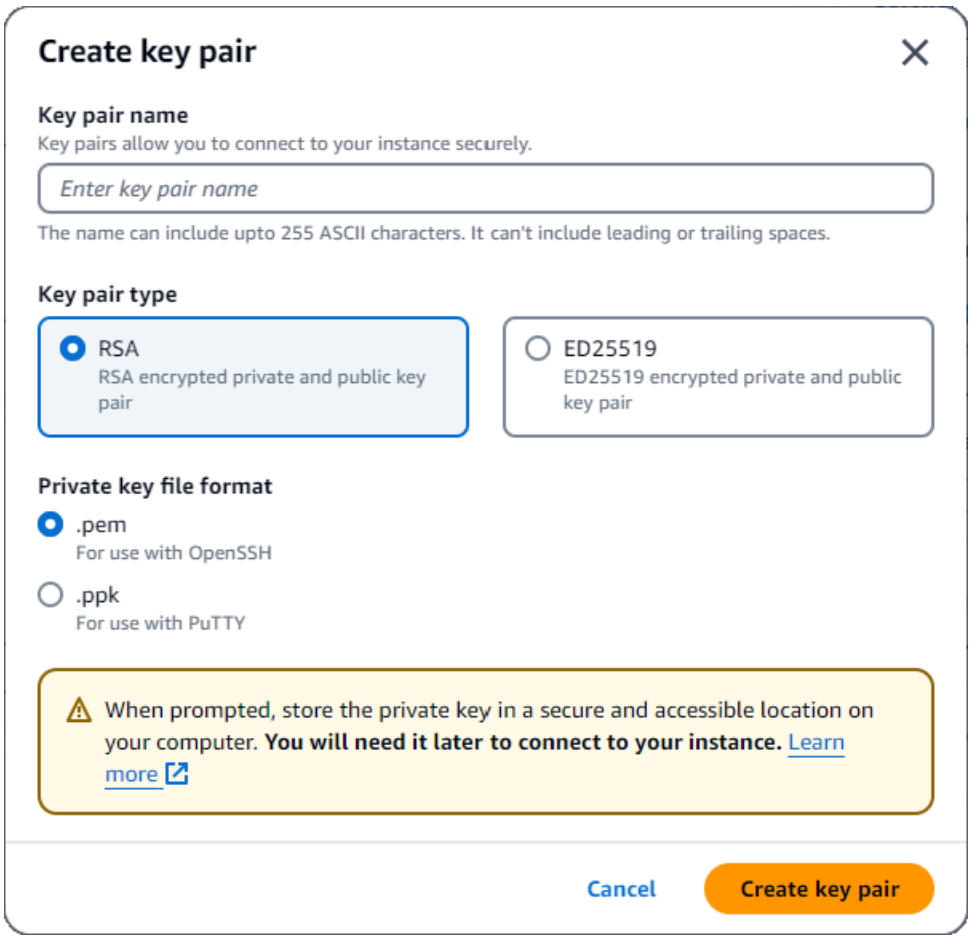


4. Locate and choose **t3.micro** in the **Instance type** drop-down list.
5. In the **Key pair (login)** section, enter the identifier of an existing key-pair, or choose **Create new key pair**.



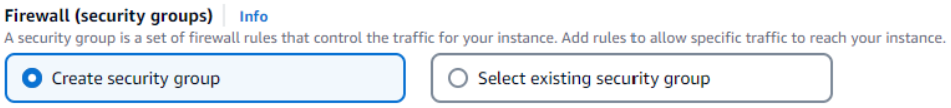
   You must provide an Amazon EC2 key pair.

   - If you do have an Amazon EC2 key pair:
     1. Select a key pair, choose your key pair from the list.
     2. You must already have the private key file (.pem or .ppk file) available to log in to your Amazon EC2 instance.
   - If you do not have an Amazon EC2 key pair:
     1. Choose **Create new key pair**, the **Create key pair** dialog box appears.
     2. Enter a name in the **Key pair name** field.
     3. Choose the **Key pair type** and **Private key file format**.
     4. Choose **Create key pair**.



> **Note**
>
> For security purposes, we highly recommend using a key-pair for both SSH and internet connectivity to your EC2 instance.

6. In the **Network settings section**, under **Firewall (security groups)**, choose either **Create security group** or **Select existing security group**.



   If you chose to select an existing security group, select one from the **Common security groups** drop-down list.

   If you chose to create a new security group, perform the following:

a. Check all the traffic allow rules that apply to your EC2 connectivity.

b. In the IP field, choose **My IP** or select **Custom** to choose from a list of CIDR blocks, prefix lists, or security groups. We do not recommend **Anywhere** as a choice, unless your EC2 instance is on an isolated network, because it allows any IP address access to your EC2 instance.

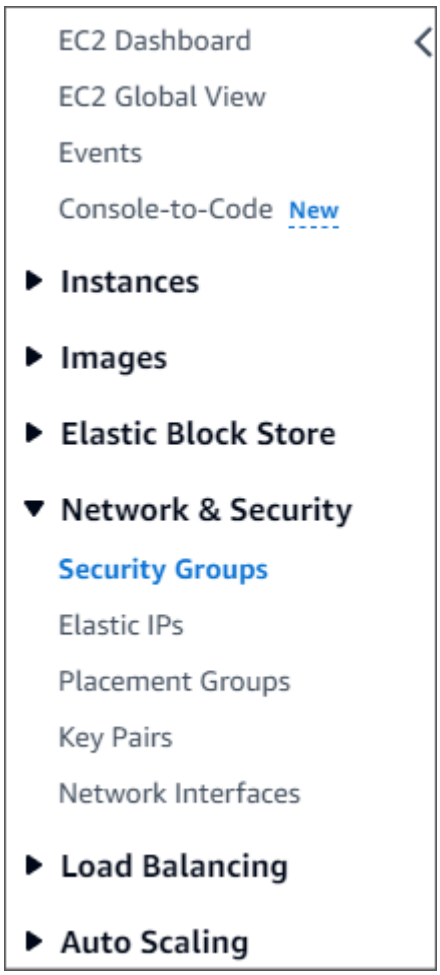


7. In the **Summary** section, review your EC2 configuration and choose **Launch instance** if correct.

Step 2: Create a security group

You will now create a new security group in your default Amazon VPC. The security group demoDocDB enables you to connect to your Amazon DocumentDB cluster on port 27017 (the default port for Amazon DocumentDB) from your Amazon EC2 instance.

1. On the Amazon EC2 Management Console, under **Network and Security**, choose **Security groups**.



2. Choose **Create security group**.



3. In the **Basic details** section:

   a. For **Security group name**, enter `demoDocDB`.

   b. For **Description**, enter a description.

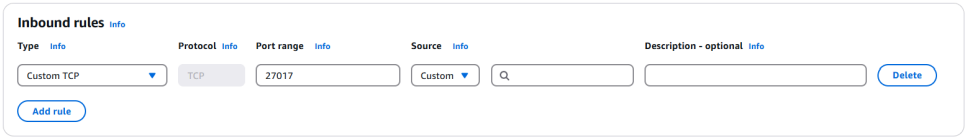
   c. For **VPC**, accept the usage of your default VPC.

4. In the **Inbound rules** section, choose **Add rule**.

   a. For **Type**, choose **Custom TCP Rule** (default).

   b. For **Port range**, enter `27017`.

   c. For **Source**, choose **Custom**. In the field next to it, search for the security group you just created in step 1. You may need to refresh your browser for the Amazon EC2 console to auto-populate the source name.



5. Accept all other defaults and choose **Create security group**.



Step 3: Create an Amazon DocumentDB cluster

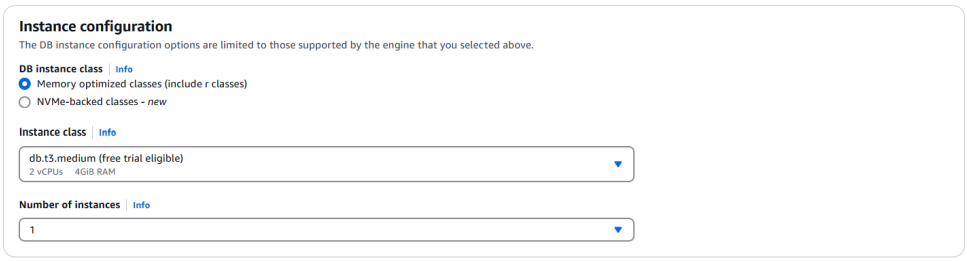
While the Amazon EC2 instance is being provisioned, you will create your Amazon DocumentDB cluster.

1. Navigate to the Amazon DocumentDB console and choose **Clusters** from the navigation pane.
2. Choose **Create**.
3. Leave the **Cluster type** setting at it's default of **Instance Based Cluster**.
4. In **Cluster configuration**, for **Cluster identifier**, enter a unique name. Note that the console will change all cluster names into lower-case regardless of how they are entered.

   For **Engine version**, keep the default version, or choose another available version. For more information, see Amazon DocumentDB cluster settings.
5. For **Cluster storage configuration**, leave the default setting of **Amazon DocumentDB Standard**.
6. In **Instance configuration**:

   - For **DB instance class**, choose **Memory optimized classes (include r classes)** (this is default).

     The other instance option is **NVMe-backed classes**. To learn more, see NVMe-backed instances.
   - For **Instance class**, choose the instance type that suits your needs. For a more detailed explanation of instance classes, see Instance class specifications.
   - For **number of instances**, choose a number that best reflects your needs. Remember, the lower the number, the lower the cost, and the lower the read/write volume that can be managed by the cluster.



7. For **Connectivity**, leave the default setting of **Don't connect to an EC2 compute resource**.

> **Note**
>
> Connecting to an EC2 compute resource automatically creates security groups for your connection to your cluster. Since you manually created these security groups in the previous step, you should select **Don't connect to an EC2 compute resource** so as not to create a second set of security groups.

8. In the **Authentication** section, enter a username for the primary user, and then choose **Self managed**. Enter a password, then confirm it.

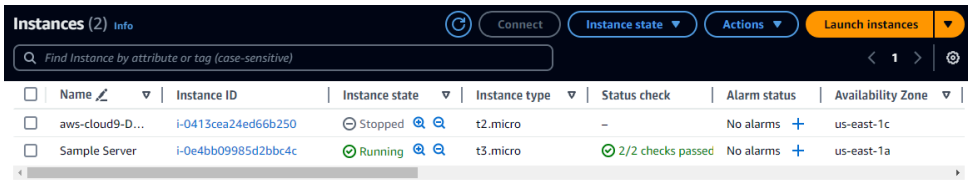
   If you instead chose **Managed in AWS Secrets Manager**, see Password management with Amazon DocumentDB and AWS Secrets Manager for more information.

9. Choose **Create cluster**.

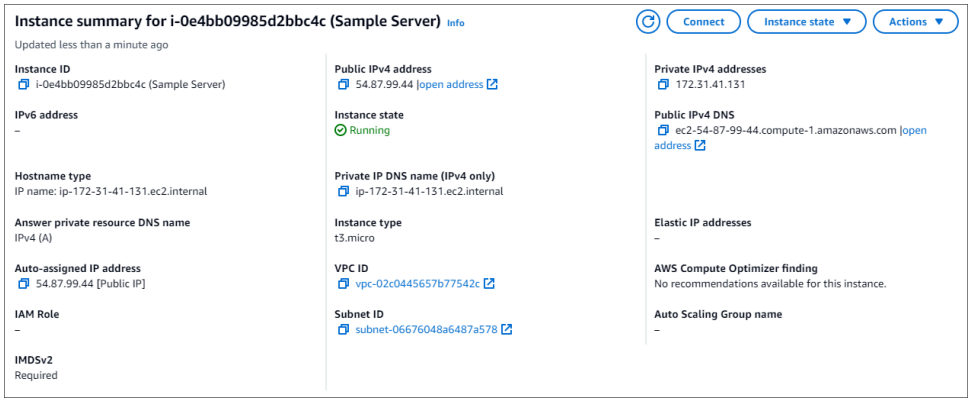
Step 4: Connect to your Amazon EC2 instance

Connecting to your Amazon EC2 instance will allow you to install the MongoDB shell. Installing the mongo shell enables you to connect to and query your Amazon DocumentDB cluster. Complete the following steps:

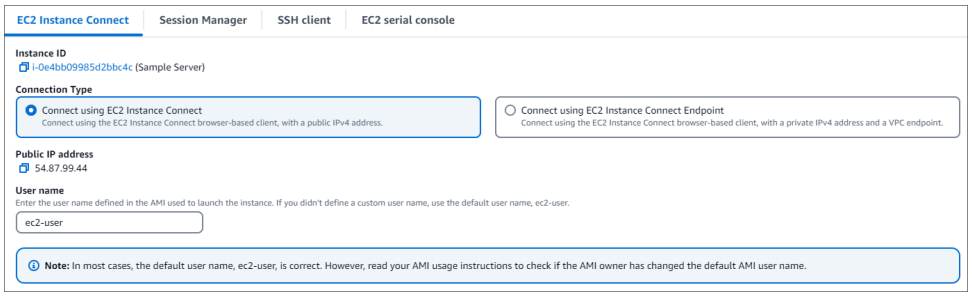
1. On the Amazon EC2 console, navigate to your instances and see if the instance you just created is running. If it is, select the instance by clicking the instance ID.



2. Choose **Connect**.



3. There are four tabbed options for your connection method: Amazon EC2 Instance Connect, Session Manager, SSH client, or EC2 serial console. You must choose one and follow its instructions. When complete, choose **Connect**.



> **Note**
>
> If your IP address changed after you started this walk-through, or you are coming back to your environment at a later time, you must update your demoEC2 security group inbound rule to enable inbound traffic from your new API address.

Step 5: Install the MongoDB Shell

You can now install the MongoDB shell, which is a command-line utility that you use to connect and query your Amazon DocumentDB cluster. There are currently two versions of MongoDB shell: the newest version, mongosh, and the previous version, mongo shell.

> **Important**
>
> There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js

drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

Follow the instructions below to install the MongoDB shell for your operating system.

On Amazon Linux

To install the MongoDB shell on Amazon Linux

If you are not using IAM and want to use the latest MongoDB shell (mongosh) to connect to your Amazon DocumentDB cluster, follow these steps:

1. Create the repository file. At the command line of your EC2 instance you created, execute the follow command:

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2023/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://pgp.mongodb.com/server-5.0.asc" | sudo tee /etc/yum.repos.d/
mongodb-org-5.0.repo
```

2. When it is complete, install mongosh with one of the two following command options at the command prompt:

   Option 1 — If you chose the default Amazon Linux 2023 during the Amazon EC2 configuration, enter this command:

```
sudo yum install -y mongodb-mongosh-shared-openssl3
```

   Option 2 — If you chose Amazon Linux 2 during the Amazon EC2 configuration, enter this command:

```
sudo yum install -y mongodb-mongosh
```

If you are using IAM, you must use the previous version of the MongoDB shell (5.0) to connect to your Amazon DocumentDB cluster, follow these steps:

1. Create the repository file. At the command line of your EC2 instance you created, execute the follow command:

```
echo -e "[mongodb-org-5.0] \nname=MongoDB Repository\nbaseurl=https://
repo.mongodb.org/yum/amazon/2023/mongodb-org/5.0/x86_64/\ngpgcheck=1 \nenabled=1
 \ngpgkey=https://pgp.mongodb.com/server-5.0.asc" | sudo tee /etc/yum.repos.d/
mongodb-org-5.0.repo
```

2. When it is complete, install the mongodb 5.0 shell with the following command option at the command prompt:

```
sudo yum install -y mongodb-org-shell
```

On Ubuntu

To install mongosh on Ubuntu

1. Import the public key that will be used by the package management system.

```
curl -fsSL https://pgp.mongodb.com/server-5.0.asc | sudo gpg --dearmor -o /usr/
share/keyrings/mongodb-server-5.0.gpg
```

2. Create the list file `mongodb-org-5.0.list` for MongoDB using the command appropriate for your version of Ubuntu.

```
echo "deb [ arch=amd64,arm64 signed-by=/usr/share/keyrings/mongodb-
server-5.0.gpg ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/5.0
 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-5.0.list
```

3. Import and update the local package database using the following command:

```
sudo apt-get update
```

4. Install mongosh.

```
sudo apt-get install -y mongodb-mongosh
```

For information about installing earlier versions of MongoDB on your Ubuntu system, see Install MongoDB Community Edition on Ubuntu.

On other operating systems

To install the mongo shell on other operating systems, see Install MongoDB Community Edition in the MongoDB documentation.

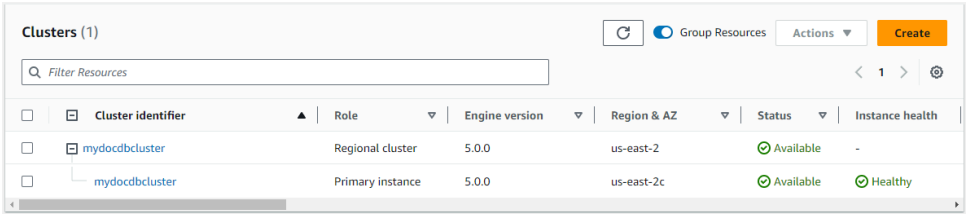
Step 6: Manage Amazon DocumentDB TLS

Download the CA certificate for Amazon DocumentDB with the following code: `wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem`

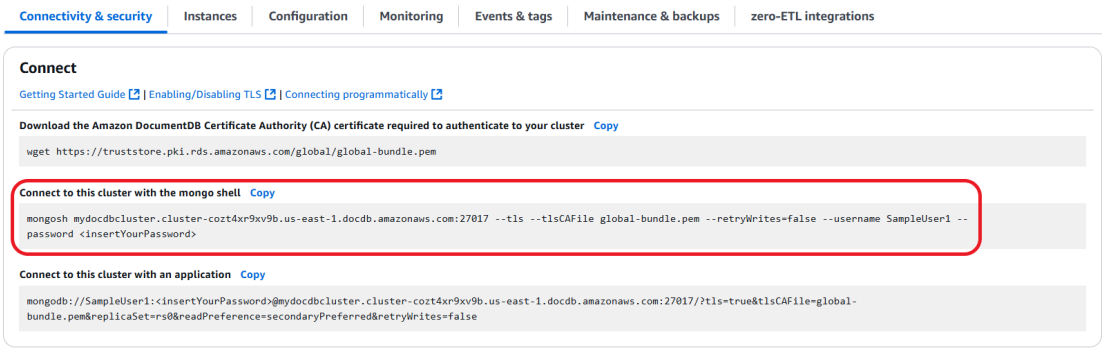
> **Note**
>
> Transport Layer Security (TLS) is enabled by default for any new Amazon DocumentDB clusters. For more information, see Encrypting data in transit.

Step 7: Connect to your Amazon DocumentDB cluster

1. On the Amazon DocumentDB console, under **Clusters**, locate your cluster. Choose the cluster you created by clicking the **Cluster identifier** for that cluster.



2. In the **Connectivity and security** tab, locate **Connect to this cluster with the mongo shell** in the **Connect** box:



Copy the connection string provided and paste it into your terminal.

Make the following changes to it:

a. Make sure you have the correct username in the string.

b. Omit `<insertYourPassword>` so that you are prompted for the password by the mongo shell when you connect.

c. Optional: If you are using IAM authentication, or are using the previous version of the MongoDB shell, modify your connection string as follows:

```
mongo --ssl --host mydocdbcluster.cluster-cozt4xr9xv9b.us-
east-1.docdb.amazonaws.com:27017 --sslCAFile global-bundle.pem --
username SampleUser1 --password
```

Replace `mydocdbcluster.cluster-cozt4xr9xv9b.us-east-1` with the same information from your cluster.

3. Press enter in your terminal. You are now be prompted for your password. Enter your password.

4. When you enter your password and can see the `rs0 [direct: primary] <env-name>>` prompt, you are successfully connected to your Amazon DocumentDB cluster.

To solve connection problems, see Troubleshooting Amazon DocumentDB.

Step 8: Insert and query data

Now that you are connected to your cluster, you can run a few queries to get familiar with using a document database.

1. To insert a single document, enter the following:

```
db.collection.insertOne({"hello":"DocumentDB"})
```

You get the following output:

```
{
  acknowledged: true,
  insertedId: ObjectId('673657216bdf6258466b128c')
}
```

2. You can read the document that you wrote with the `findOne()` command (because it only returns a single document). Input the following:

```
db.collection.findOne()
```

You get the following output:

```
{ "_id" : ObjectId("5e401fe56056fda7321fbd67"), "hello" : "DocumentDB" }
```

3. To perform a few more queries, consider a gaming profiles use case. First, insert a few entries into a collection titled `profiles`. Input the following:

```
db.profiles.insertMany([{ _id: 1, name: 'Matt', status: 'active', level: 12, score:
 202 },
     { _id: 2, name: 'Frank', status: 'inactive', level: 2, score: 9 },
     { _id: 3, name: 'Karen', status: 'active', level: 7, score: 87 },
     { _id: 4, name: 'Katie', status: 'active', level: 3, score: 27 }
])
```

You get the following output:

```
{ acknowledged: true, insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4 } }
```

4. Use the `find()` command to return all the documents in the profiles collection. Input the following:

```
db.profiles.find()
```

You will get an output that will match the data you typed in Step 3.

5. Use a query for a single document using a filter. Input the following:

```
db.profiles.find({name: "Katie"})
```

You get the following output:

```
{ "_id" : 4, "name" : "Katie", "status": "active", "level": 3, "score":27}
```

6. Now let's try to find a profile and modify it using the `findAndModify` command. We'll give the user Matt an extra 10 points with the following code:

```
db.profiles.findAndModify({
```

```
        query: { name: "Matt", status: "active"},
        update: { $inc: { score: 10 } }
    })
```

You get the following output (note that his score hasn't increased yet):

```
{
    [{_id : 1, name : 'Matt', status: 'active', level: 12, score: 202}]
```

7. You can verify that his score has changed with the following query:

   ```
   db.profiles.find({name: "Matt"})
   ```

   You get the following output:

   ```
   { "_id" : 1, "name" : "Matt", "status" : "active", "level" : 12, "score" : 212 }
   ```

Step 9: Explore

Congratulations! You have successfully completed the Quick Start Guide to Amazon DocumentDB.

What's next? Learn how to fully leverage this powerful database with some of its popular features:

- Managing Amazon DocumentDB clusters and instances
- Scaling
- Backing up and restoring

> **Note**
>
> To save on cost, you can either stop your Amazon DocumentDB cluster to reduce costs or delete the cluster. By default, after 30 minutes of inactivity, your AWS Cloud9 environment will stop the underlying Amazon EC2 instance.

Connect using the Amazon DocumentDB JDBC driver

The JDBC driver for Amazon DocumentDB provides an SQL-relational interface for developers and enables connectivity from BI tools such as Tableau and DbVisualizer.

For more detailed information, refer to the Amazon DocumentDB JDBC Driver documentation on GitHub.

Topics

- Getting started
- Connect to Amazon DocumentDB from Tableau Desktop
- Connect to Amazon DocumentDB from DbVisualizer
- JDBC automatic schema generation
- SQL support and limitations
- Troubleshooting

Getting started

Step 1. Create Amazon DocumentDB Cluster

If you do not have an Amazon DocumentDB cluster created, then create one using the instructions in the Get started with Amazon DocumentDB.

> **Note**
>
> Amazon DocumentDB is a Virtual Private Cloud (VPC) only service. If you are connecting from a local machine, outside the cluster's VPC, you will need to create an SSH connection to an Amazon EC2 instance. In this case, launch your cluster using the instructions in Connect using Amazon EC2. See Using an SSH Tunnel to Connect to Amazon DocumentDB for more information on SSH tunneling and when you might need it.

Step 2. JRE or JDK Installation

Depending on your BI application, you may need to ensure a 64-bit JRE or JDK installation version 8 or later is installed on your computer. You can download the Java SE Runtime Environment 8 here.

Step 3. Download the DocumentDB JDBC Driver

Download the DocumentDB JDBC driver from here. The driver is packaged as a single JAR file (e.g. documentdb-jdbc-1.0.0-all.jar).

Step 4. Using an SSH Tunnel to Connect to Amazon DocumentDB

Amazon DocumentDB (with MongoDB compatibility) clusters are deployed within an Amazon Virtual Private Cloud (Amazon VPC). They can be accessed directly by Amazon EC2 instances or other AWS services that are deployed in the same Amazon VPC. Additionally, Amazon DocumentDB can be accessed by EC2a instances or other AWS services in different VPCs in the same AWS Region or other Regions via VPC peering.

You can use SSH tunneling (also known as port forwarding) to access your Amazon DocumentDB resources, from outside the cluster's VPC. This will be the case for most users not running their application on a VM in the same VPC as the DocumentDB cluster.

To create an SSH tunnel, you need an Amazon EC2 instance running in the same Amazon VPC as your Amazon DocumentDB cluster. You can either use an existing EC2 instance in the same VPC as your cluster or create one. You can set up an SSH tunnel to the Amazon DocumentDB cluster `sample-cluster.node.us-east-1.docdb.amazonaws.com` by running the following command on your local computer.

```
ssh -i "ec2Access.pem" -L 27017:sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017 ubuntu@ec2-34-229-221-164.compute-1.amazonaws.com -
N
```

The -L flag is used for forwarding a local port. This is a prerequisite for connecting to any BI tool running on a client outside your VPC. Once you run the step above you can move on to the next steps for the BI tool of your choice.

For further information on SSH tunneling , refer to the documentation on Using an SSH tunnel to connect to Amazon DocumentDB.

Connect to Amazon DocumentDB from Tableau Desktop

Topics

- Adding the Amazon DocumentDB JDBC Driver
- Connecting to Amazon DocumentDB using Tableau - SSH Tunnel

Adding the Amazon DocumentDB JDBC Driver

To connect to Amazon DocumentDB from Tableau Desktop you must download and install the Amazon DocumentDB JDBC driver and the DocumentDB Tableau connector.

1. Download the Amazon DocumentDB JDBC driver JAR file from the Amazon DocumentDB JDBC Driver repository and copy it to one of these directories according to your operating system:
   - *Windows* - `C:\Program Files\Tableau\Drivers`
   - *MacOS* - `~/Library/Tableau/Drivers`
2. Download the DocumentDB Tableau connector (a TACO file) from the Tableau Exchange website and copy it to your *My Tableau Repository/Connectors directory*.
   - *Windows* - `C:\Users\[user]\Documents\My Tableau Repository\Connectors`
   - *MacOS* - `/Users/[user]/Documents/My Tableau Repository/Connectors`

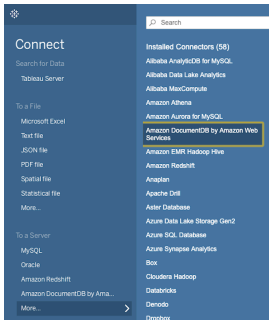
For additional information, refer to the Tableau documentation.

> **Note**
>
> If you are using newer CA certificates, make sure to upgrade your JDBC driver to v1.4.5 (available in this AWS GitHub repository.).

Connecting to Amazon DocumentDB using Tableau - SSH Tunnel

To connect to Tableau from a client machine outside of the VPC of your DocumentDB cluster, you must setup an SSH tunnel before following the steps below:

1. Launch the Tableau Desktop application.
2. Navigate to **Connect** > **To A Server** > **More**.
3. Choose **Amazon DocumentDB by Amazon Web Services** under **Installed Connectors**.

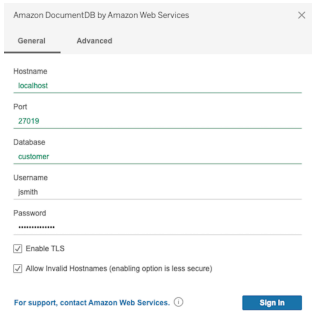


Connecting to Amazon DocumentDB using Tableau - external SSH tunnel

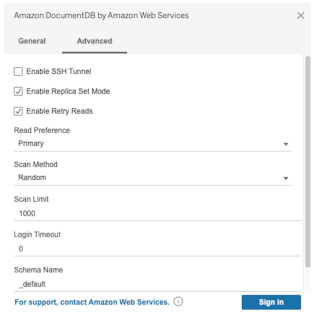
1. Enter the required connection parameters **Hostname**, **Port**, **Database**, **Username** and **Password**. The connection parameters in the following example are equivalent to the JDBC connection string :

   ```
   jdbc:documentdb://localhost:27019/test?
   tls=true&tlsAllowInvalidHostnames=true&scanMethod=random&scanLimit=1000&logir
   ```

   with the username and password parameters passed separately in a properties collection. For more information on connection string parameters, refer to the [Amazon DocumentDB JDBC Driver github documentation].



2. (Optional) More advanced options can be found on the **Advanced** tab.

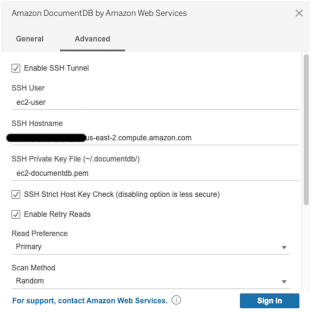


3. Choose **Sign in.**

Connecting to Amazon DocumentDB using Tableau - internal SSH tunnel

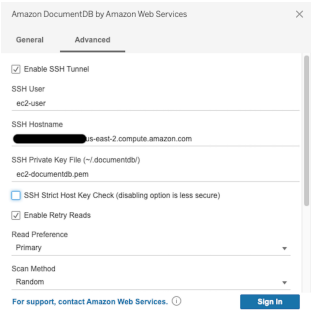
> **Note**
>
> If you prefer to not setup the SSH tunnel using a terminal, you can use the Tableau GUI to specify your EC2 instance details which the JDBC driver will inherently use to create a SSH tunnel.

1. On the **Advanced** tab, choose the **Enable SSH Tunnel option** to review further properties.



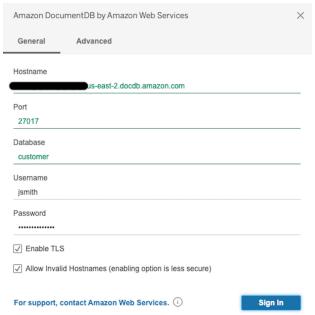
2. Enter the **SSH User**, **SSH Hostname**, and **SSH Private Key File**.
3. (Optional) You can disable the **SSH Strict Host Key Check** option which bypasses the host key check against a known hosts file.

> **Note**
>
> Disabling this option is less secure as it can lead to a man-in-the-middle attack.



4. Enter the required parameters; **Hostname**, **Port**, **Database**, **Username** and **Password**.

> **Note**
>
> Make sure you use the DocumentDB cluster endpoint and not localhost when using the internal SSH tunnel option.



5. Choose **Sign In**.

Connect to Amazon DocumentDB from DbVisualizer

Topics

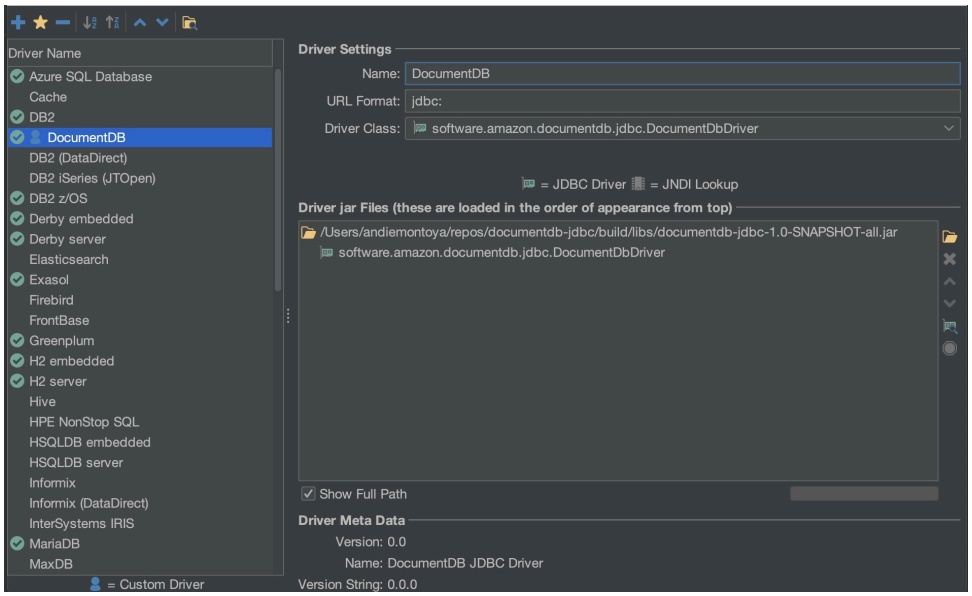
- Adding the Amazon DocumentDB JDBC driver
- Connecting to Amazon DocumentDB using DbVisualizer

Adding the Amazon DocumentDB JDBC driver

To connect to Amazon DocumentDB from DbVisualizer you must first import the Amazon DocumentDB JDBC Driver

1. Start the DbVisualizer application and navigate to the menu path: **Tools > Driver Manager...**
2. Choose **+** (or in the menu, select **Driver > Create Driver**).
3. Set **Name** to `DocumentDB`.
4. Set **URL Format** to `jdbc:documentdb://<host>[:port]/<database>[?option=value[&option=value[...]]]`
5. Choose the **folder** button and then select the Amazon DocumentDB JDBC driver JAR file and choose the **Open** button.

6. Verify that the **Driver Class** field is set to `software.amazon.documentdb.jdbc.DocumentDbDriver`. Your Driver Manager settings for **DocumentDB** should look like the following example.



7. Close the dialog. The Amazon DocumentDB JDBC driver will be setup and ready to use.

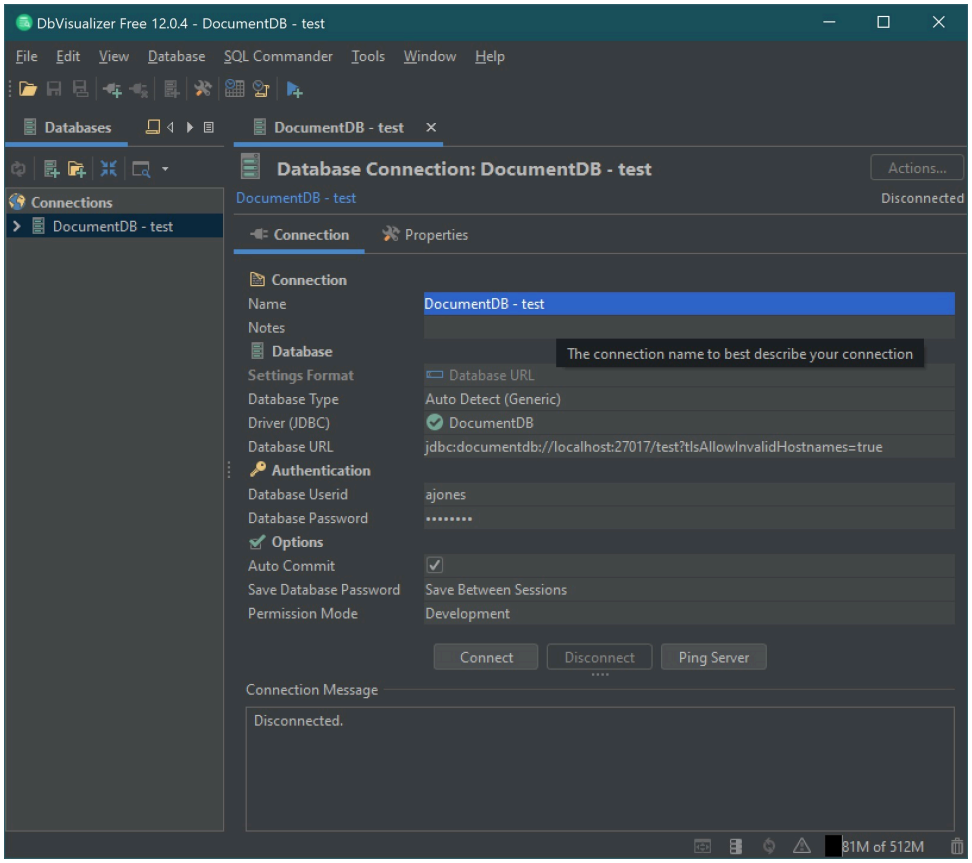
Connecting to Amazon DocumentDB using DbVisualizer

Connect to Amazon DocumentDB Using DbVisualizer

1. If you are connecting from outside the Amazon DocumentDB cluster's VPC, ensure you have setup an SSH tunnel.
2. Choose **Database > Create Database Connection** from the top level menu.
3. Enter a descriptive name for the **Name** field.
4. Set **Driver (JDBC)** to the DocumentDB driver you created in the previous section.
5. Set **Database URL** to your JDBC connection string.

   For example: `jdbc:documentdb://localhost:27017/database?tlsAllowInvalidHostnames=true`
6. Set **Database Userid** to your Amazon DocumentDB user ID.
7. Set **Database Password** to the corresponding password for the user ID.

   Your Database Connection dialog should look like the following dialog:



8. Choose **Connect**.

JDBC automatic schema generation

Amazon DocumentDB is a document database and therefore does not have the concept of tables and schema. However, BI tools such as Tableau will expect the database it connects to present a schema. Specifically, when the JDBC driver connection needs to get the schema for the collection in the database, it will poll for all the collections in the database. The driver will determine if a cached version of the schema for that collection already exists. If a cached version does not exist, it will sample the collection for documents and create a schema based on the following behavior.

Topics

- Schema generation limitations
- Scanning method options
- Amazon DocumentDB data types
- Mapping scalar document fields
- Object and array data type handling

Schema generation limitations

The DocumentDB JDBC driver imposes a limit on the length of identifiers at 128 characters. The schema generator may truncate the length of generated identifiers (table names and column names) to ensure they fit that limit.

Scanning method options

The sampling behavior can be modified using connection string or data source options.

- *scanMethod*=<option>
  - *random - (default) - The sample documents are returned in random order.*
  - *idForward* - The sample documents are returned in order of id.
  - *idReverse* - The sample documents are returned in reverse order of id.
  - *all* - Sample all the documents in the collection.
- *scanLimit*=<n> - The number of documents to sample. The value must be a positive integer. The default value is *1000*. If *scanMethod* is set to *all*, this option is ignored.

Amazon DocumentDB data types

The Amazon DocumentDB server supports a number of MongoDB data types. Listed below are the supported data types, and their associated JDBC data types.

MongoDB Data Type	Supported in DocumentDB	JDBC Data Type
Binary Data	Yes	VARBINARY
Boolean	Yes	BOOLEAN
Double	Yes	DOUBLE
32-bit Integer	Yes	INTEGER
64-bit Integer	Yes	BIGINT
String	Yes	VARCHAR
ObjectId	Yes	VARCHAR

MongoDB Data Type	Supported in DocumentDB	JDBC Data Type
Date	Yes	TIMESTAMP
Null	Yes	VARCHAR
Regular Expression	Yes	VARCHAR
Timestamp	Yes	VARCHAR
MinKey	Yes	VARCHAR
MaxKey	Yes	VARCHAR
Object	Yes	virtual table
Array	Yes	virtual table
Decimal128	No	DECIMAL
JavaScript	No	VARCHAR
JavaScript (with scope)	No	VARCHAR
Undefined	No	VARCHAR
Symbol	No	VARCHAR
DBPointer (4.0+)	No	VARCHAR

Mapping scalar document fields

When scanning a sample of documents from a collection, the JDBC driver will create one or more schema to represent the samples in the collection. In general, a scalar field in the document maps to a column in the table schema. For example, in a collection named team, and a single document `{ "_id" : "112233", "name" : "Alastair", "age": 25 }`, this would map to schema:

Table Name	Column Name	Data Type	Key
team	*team id*	VARCHAR	PK

Table Name	Column Name	Data Type	Key
team	name	VARCHAR	
team	age	INTEGER	

Data type conflict promotion

When scanning the sampled documents, it is possible that the data types for a field are not consistent from document to document. In this case, the JDBC driver will promote the JDBC data type to a common data type that will suit all data types from the sampled documents.

For Example:

```
{
"_id" : "112233",
"name" : "Alastair", "age" : 25
}

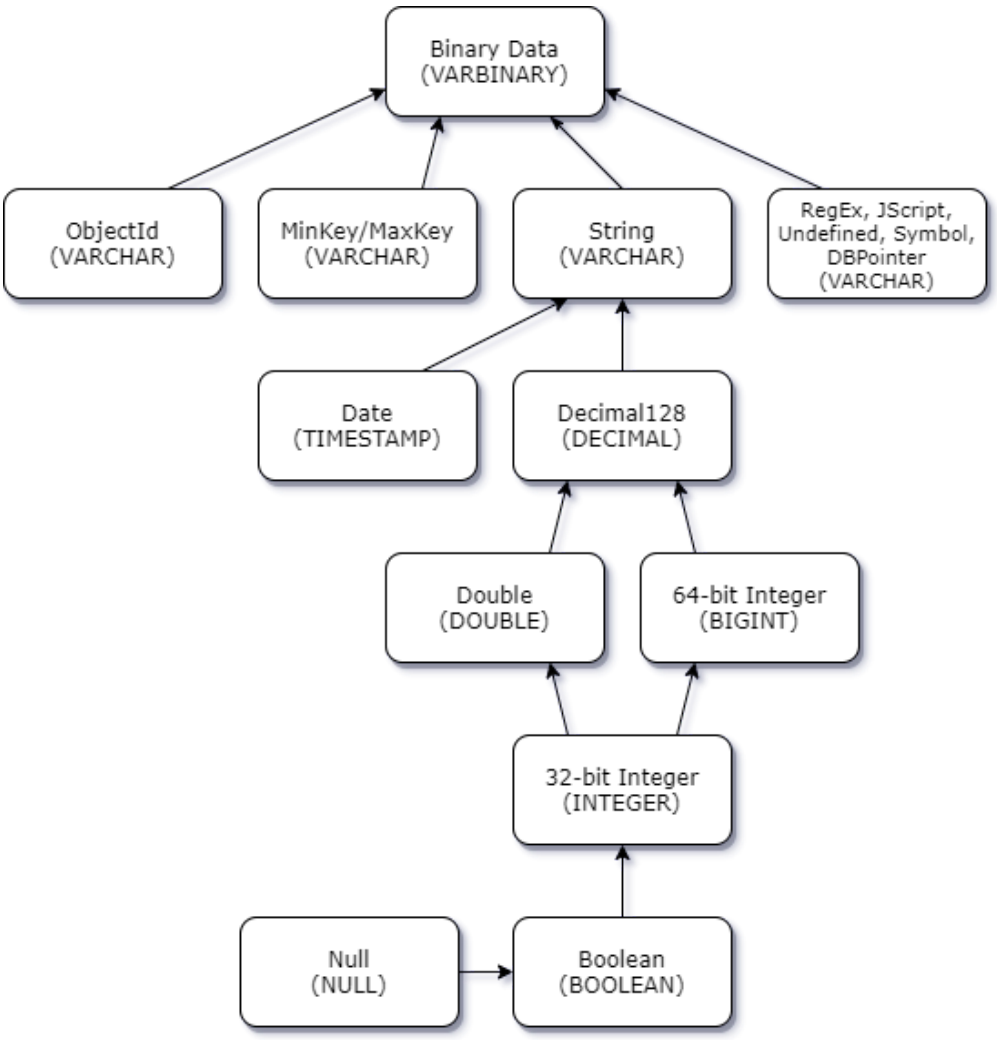
{
"_id" : "112244",
"name" : "Benjamin",
"age" : "32"
}
```

The *age* field is of type 32-bit integer in the first document but string in the second document. Here the JDBC driver will promote the JDBC data type to VARCHAR to handle either data type when encountered.

Table Name	Column Name	Data Type	Key
team	*team id*	VARCHAR	PK
team	name	VARCHAR	
team	age	VARCHAR	

Scalar-scalar conflict promotion

The following diagram shows the way in which scalar-scalar data type conflicts are resolved.



Scalar-complex type conflict promotion

Like the scalar-scalar type conflicts, the same field in different documents can have conflicting data types between complex (array and object) and scalar (integer, boolean, etc.). All of these conflicts are resolved (promoted) to VARCHAR for those fields. In this case, array and object data is returned as the JSON representation.

Embedded Array - String Field Conflict Example:

```
{
   "_id":"112233",
   "name":"George Jackson",
```

```
   "subscriptions":[
      "Vogue",
      "People",
      "USA Today"
   ]
}
{
   "_id":"112244",
   "name":"Joan Starr",
   "subscriptions":1
}
```

The preceding example maps to schema for the customer2 table:

Table Name	Column Name	Data Type	Key
customer2	*customer2 id*	VARCHAR	PK
customer2	name	VARCHAR	
customer2	subscription	VARCHAR	

and the customer1_subscriptions virtual table:

Table Name	Column Name	Data Type	Key
customer1_subscriptions	*customer1 id*	VARCHAR	PK/FK
customer1_subscriptions	subscriptions_index_lvl0	BIGINT	PK
customer1_subscriptions	value	VARCHAR	
customer_address	city	VARCHAR	
customer_address	region	VARCHAR	
customer_address	country	VARCHAR	

Table Name	Column Name	Data Type	Key
customer_address	code	VARCHAR	

Object and array data type handling

So far, we've only described how scalar data types are mapped. Object and Array data types are (currently) mapped to virtual tables. The JDBC driver will create a virtual table to represent either object or array fields in a document. The name of the mapped virtual table will concatenate the original collection's name followed by the field's name separated by an underscore character ("_").

The base table's primary key ("_id") takes on a new name in the new virtual table and is provided as a foreign key to the associated base table.

For embedded array type fields, index columns are generated to represent the index into the array at each level of the array.

Embedded object field example

For object fields in a document, a mapping to a virtual table is created by the JDBC driver.

```
{
   "Collection: customer",
   "_id":"112233",
   "name":"George Jackson",
   "address":{
      "address1":"123 Avenue Way",
      "address2":"Apt. 5",
      "city":"Hollywood",
      "region":"California",
      "country":"USA",
      "code":"90210"
   }
}
```

The preceding example maps to schema for customer table:

Table Name	Column Name	Data Type	Key
customer	*customer id*	VARCHAR	PK

Table Name	Column Name	Data Type	Key
customer	name	VARCHAR	

and the customer_address virtual table:

Table Name	Column Name	Data Type	Key
customer_address	*customer id*	VARCHAR	PK/FK
customer_address	address1	VARCHAR	
customer_address	address2	VARCHAR	
customer_address	city	VARCHAR	
customer_address	region	VARCHAR	
customer_address	country	VARCHAR	
customer_address	code	VARCHAR	

Embedded array field example

For array fields in a document, a mapping to a virtual table is also created by the JDBC driver.

```
{
   "Collection: customer1",
   "_id":"112233",
   "name":"George Jackson",
   "subscriptions":[
      "Vogue",
      "People",
      "USA Today"
   ]
}
```

The preceding example maps to schema for customer1 table:

Table Name	Column Name	Data Type	Key
customer1	*customer1 id*	VARCHAR	PK
customer1	name	VARCHAR	

and the customer1_subscriptions virtual table:

Table Name	Column Name	Data Type	Key
customer1_subscriptions	*customer1 id*	VARCHAR	PK/FK
customer1_subscriptions	subscriptions_index_lvl0	BIGINT	PK
customer1_subscriptions	value	VARCHAR	
customer_address	city	VARCHAR	
customer_address	region	VARCHAR	
customer_address	country	VARCHAR	
customer_address	code	VARCHAR	

SQL support and limitations

The Amazon DocumentDB JDBC driver is a read-only driver that supports a subset of SQL-92 and some common extensions. Refer to the SQL limitations documentation and JDBC limitations documentation for more information.

Troubleshooting

If you are having problems using the Amazon DocumentDB JDBC driver, refer to the Troubleshooting Guide.

Connect using the Amazon DocumentDB ODBC driver

The ODBC driver for Amazon DocumentDB provides an SQL-relational interface for developers and enables connectivity from BI tools such as Power BI Desktop and Microsoft Excel.

For more detailed information, refer to the Amazon DocumentDB ODBC Driver documentation on GitHub.

Topics

- Getting started
- Setting up the Amazon DocumentDB ODBC driver in Windows
- Connect to Amazon DocumentDB from Microsoft Excel
- Connect to Amazon DocumentDB from Microsoft Power BI Desktop
- Automatic schema generation
- SQL support and limitations
- Troubleshooting

Getting started

Step 1. Create Amazon DocumentDB Clusters

If you don't already have an Amazon DocumentDB cluster, there are a number of ways to get started.

> **Note**
>
> Amazon DocumentDB is a Virtual Private Cloud (VPC)-only service. If you are connecting from a local machine outside the cluster's VPC, you need to create an SSH connection to an Amazon EC2 instance. For more information, see Connect using Amazon EC2. For more information on SSH tunneling, see Using an SSH Tunnel to Connect to Amazon DocumentDB.

Step 2. JRE or JDK Installation

Depending on your BI application, you may need to ensure a 64-bit JRE or JDK installation version 8 or later installed on your computer. You can download the Java SE Runtime Environment 8 here.

Step 3. Download the Amazon DocumentDB ODBC Driver

Download the Amazon DocumentDB ODBC driver here. Choose the proper installer (for example, documentdb-odbc-1.0.0.msi). Follow the installation guide.

Step 4. Using an SSH Tunnel to Connect to Amazon DocumentDB

Amazon DocumentDB clusters are deployed within an Amazon Virtual Private Cloud (Amazon VPC). They can be accessed directly by Amazon EC2 instances or other AWS services that are deployed in the same Amazon VPC. Additionally, Amazon DocumentDB can be accessed by Amazon EC2 instances or other AWS services in different VPCs in the same AWS Region or other Regions via VPC peering.

However, suppose that your use case requires that you (or your application) access your Amazon DocumentDB resources from outside the cluster's VPC. This will be the case for most users not running their application on a VM in the same VPC as the Amazon DocumentDB cluster. When connecting from outside the VPC, you can use SSH tunneling (also known as port forwarding) to access your Amazon DocumentDB resources.

To create an SSH tunnel, you need an Amazon EC2 instance running in the same Amazon VPC as your Amazon DocumentDB cluster. You can either use an existing EC2 instance in the same VPC as your cluster or create one. You can set up an SSH tunnel to the Amazon DocumentDB cluster `sample-cluster.node.us-east-1.docdb.amazonaws.com` by running the following command on your local computer:

```
ssh -i "ec2Access.pem" -L 27017:sample-cluster.node.us-
east-1.docdb.amazonaws.com:27017 ubuntu@ec2-34-229-221-164.compute-1.amazonaws.com -
N
```

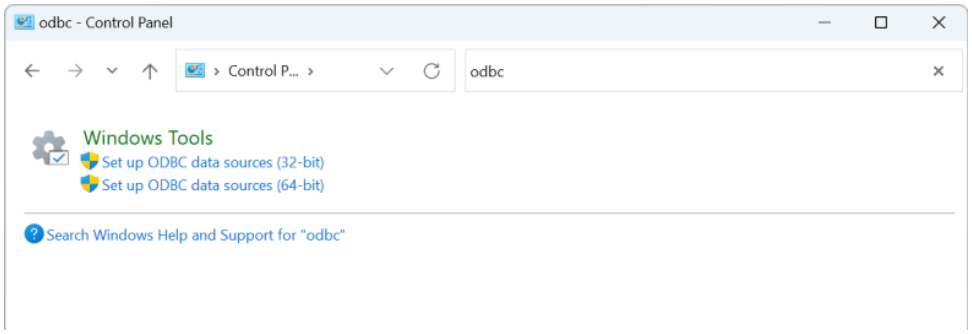
The `-L` flag is used for forwarding a local port. This is a prerequisite for connecting to any BI tool running on a client outside your VPC. Once you run the step above you can move on to the next steps for the BI tool of your choice.

For further information on SSH tunneling, refer to the documentation on Using an SSH Tunnel to Connect to Amazon DocumentDB.

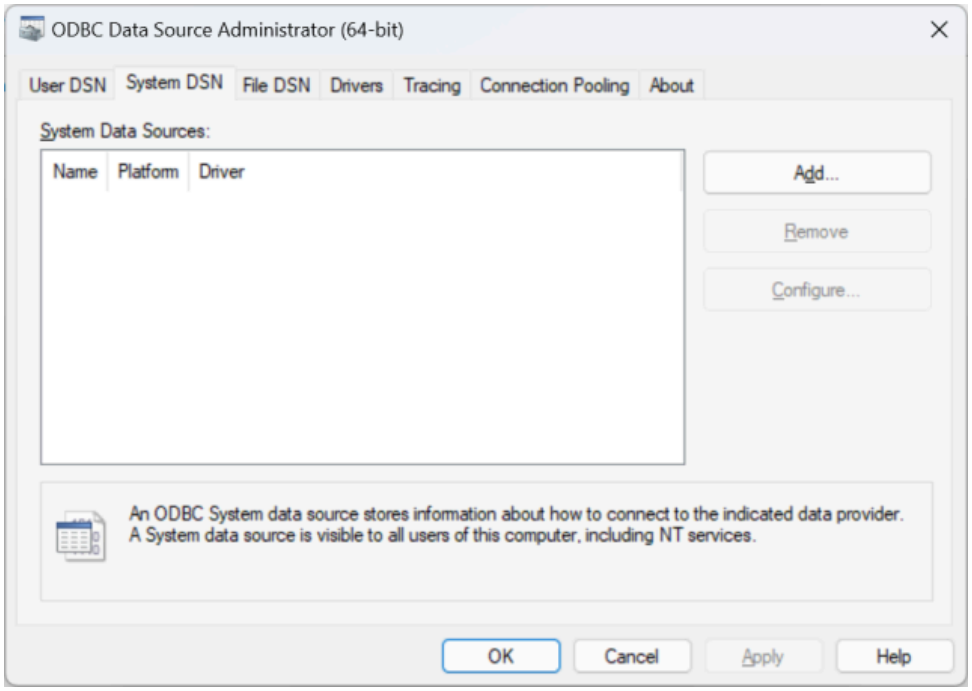
Setting up the Amazon DocumentDB ODBC driver in Windows

Use the following procedure to set up the Amazon DocumentDB ODBC driver in Windows:

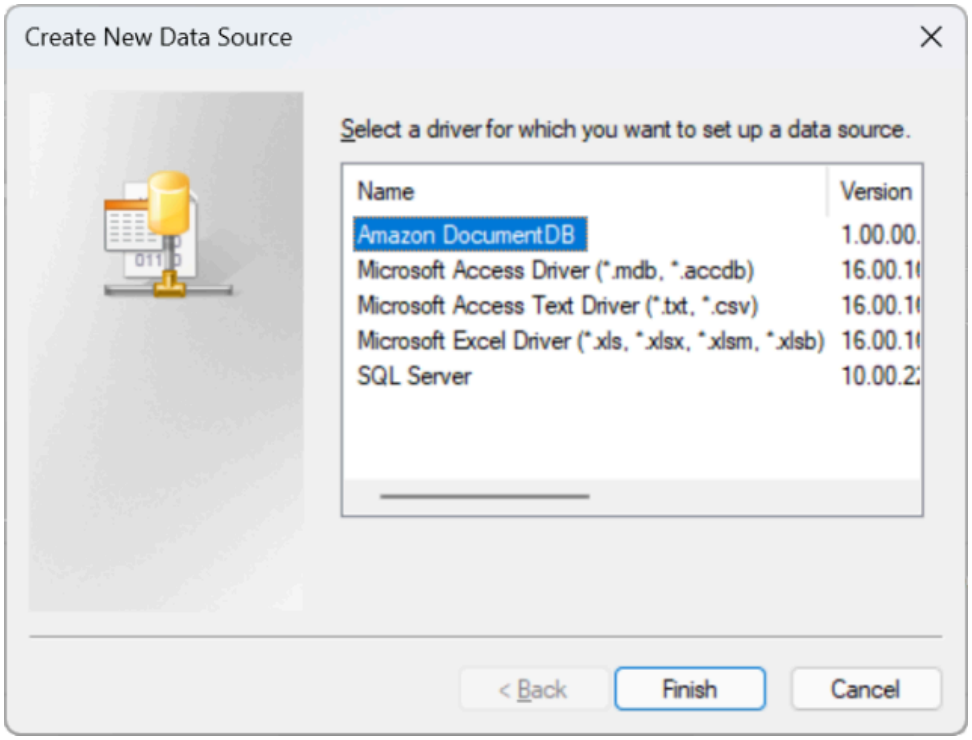
1. Open **Control Panel** in Windows and search for ODBC (or in the menu, select **Windows Tools** > **ODBC Data Sources (32-bit)** or **ODBC Data Sources (64-bit)**):



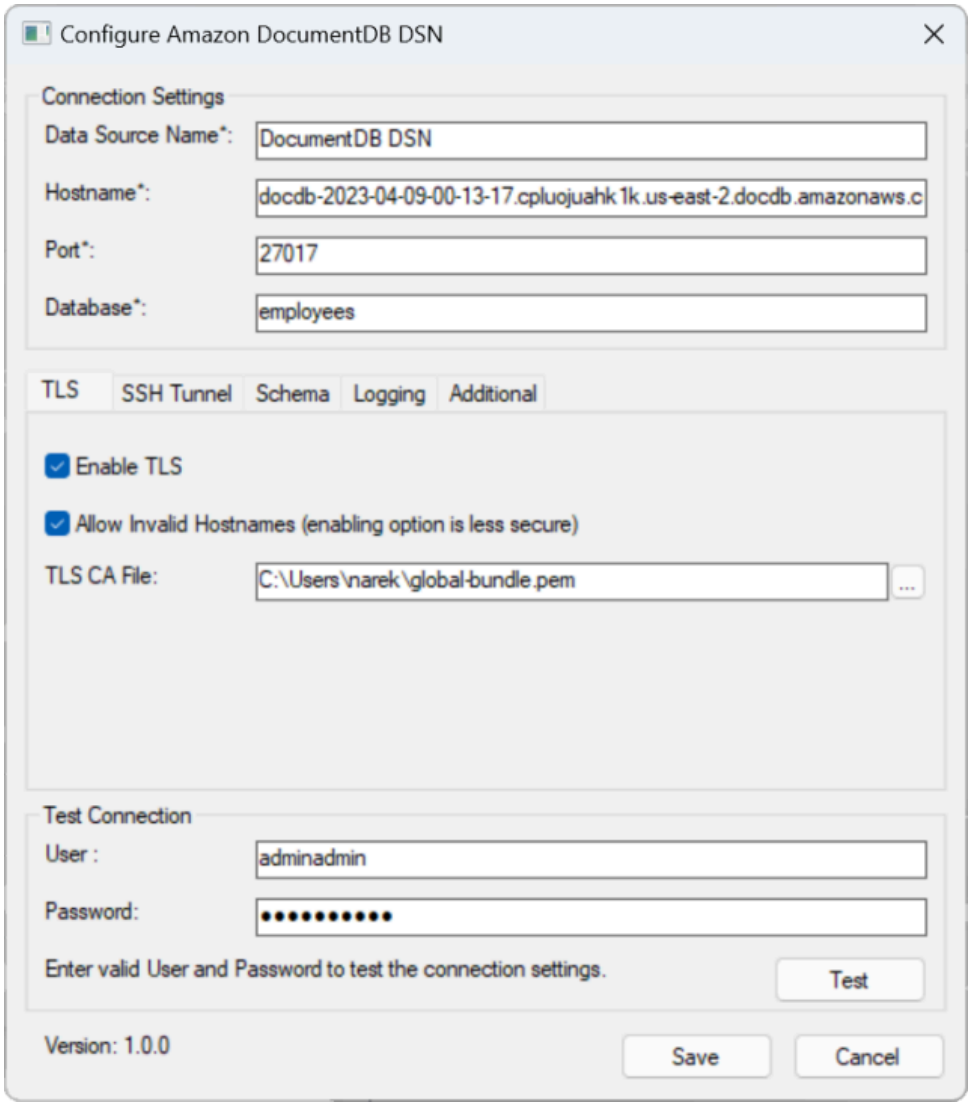
2. Select the appropriate ODBC Driver Data Source Administrator: opt for the 32-bit version if it is installed, otherwise, choose the 64-bit version.
3. Select the Sytem DSN tab and then choose **Add...** to add a new DSN:



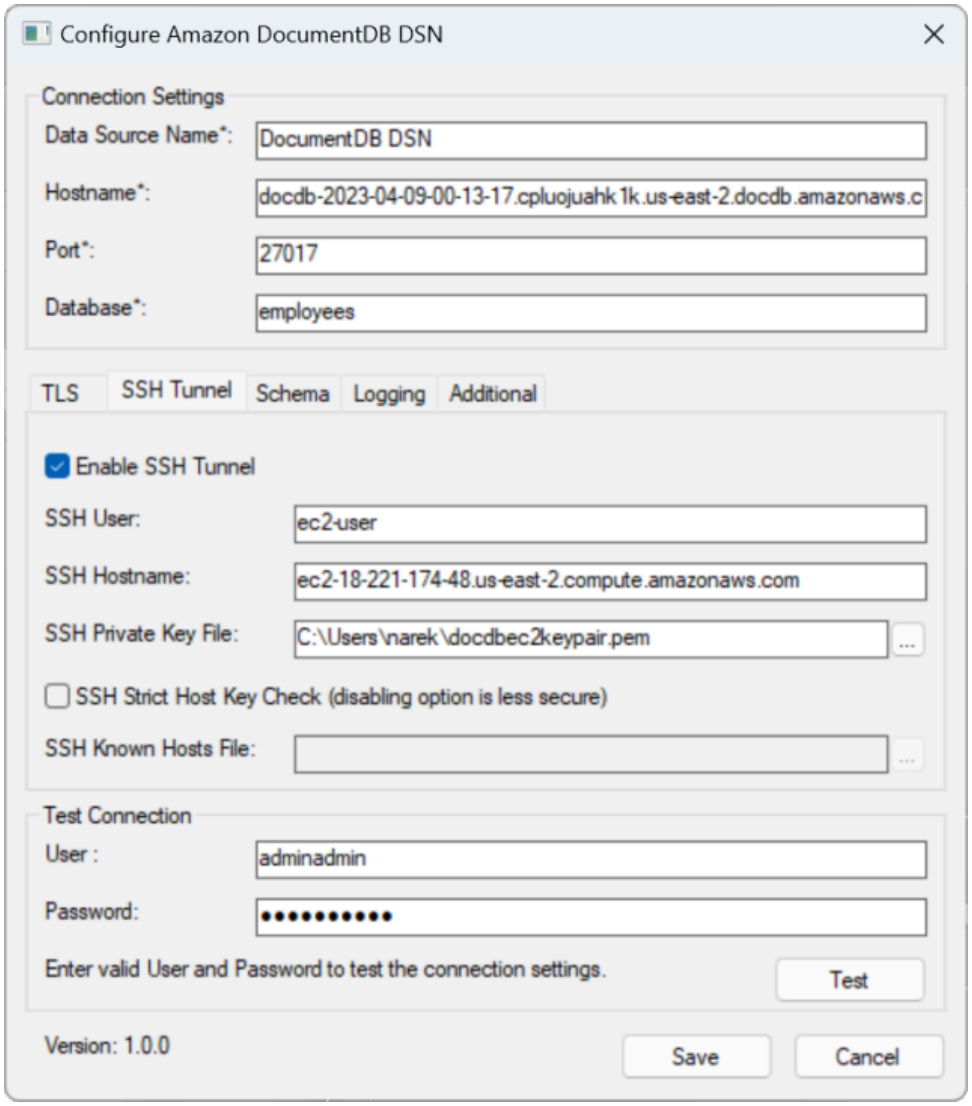
4. Choose **Amazon DocumentDB** from the data source driver list:



5. In the **Configure Amazon DocumentDB DSN** dialog, complete the **Connection Settings**, **TLS** tab, and **Test Connection** fields, then choose **Save**:



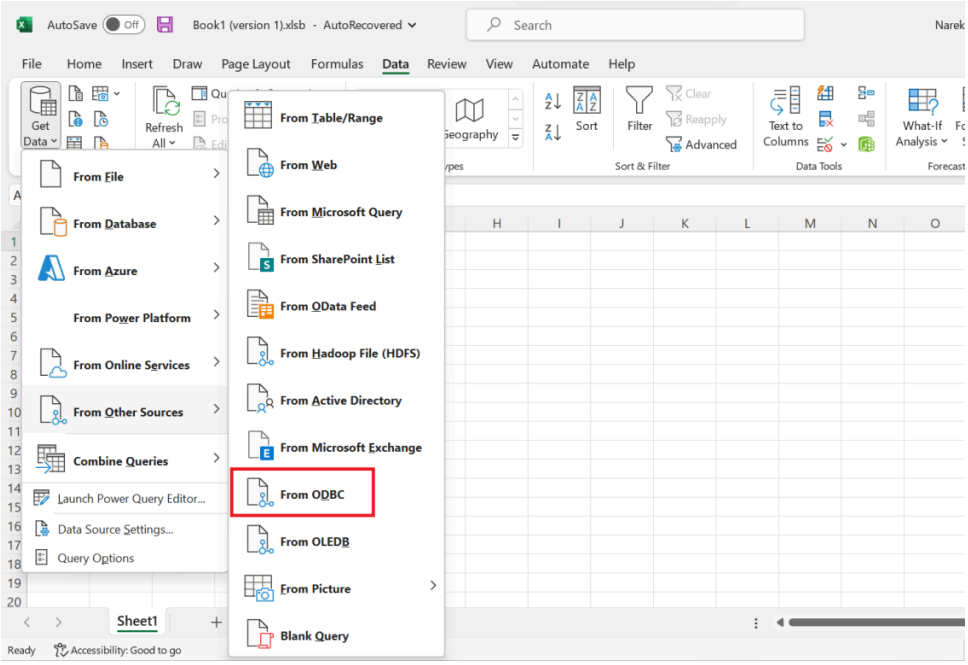
6. Ensure you complete the Windows form accurately, as connection details will differ depending on your chosen SSH tunneling method to the EC2 instance. See SSH tunneling methods here. See Connection String Syntax and Options for more information about each property.



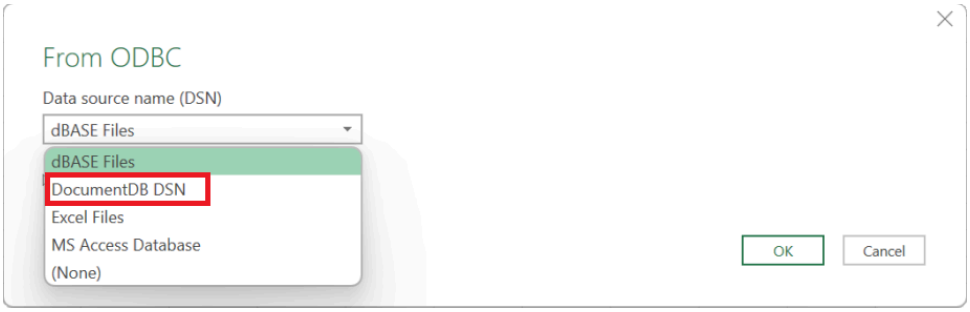
For more information about configuring the Amazon DocumentDB ODBC Driver on Windows, see here.

Connect to Amazon DocumentDB from Microsoft Excel

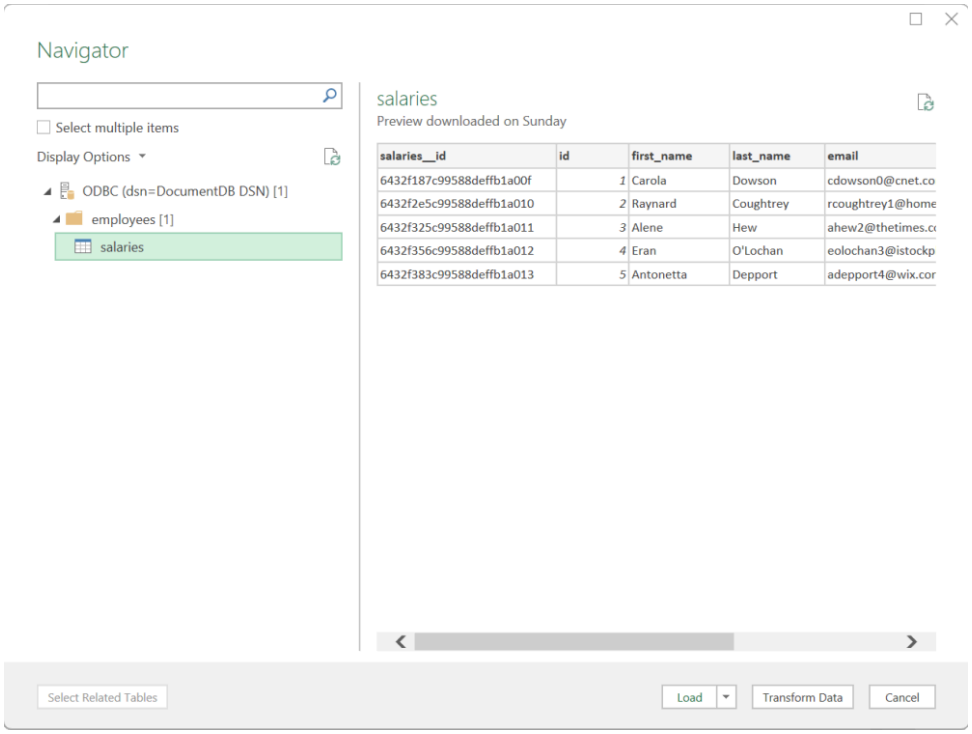
1. Ensure that the Amazon DocumentDB driver has been correctly installed and configured. For more information, see Setting up the Amazon DocumentDB ODBC driver in Windows.
2. Launch Microsoft Excel.
3. Navigate to **Data** > **Get Data** > **From Other Sources**.
4. Choose **From ODBC**:



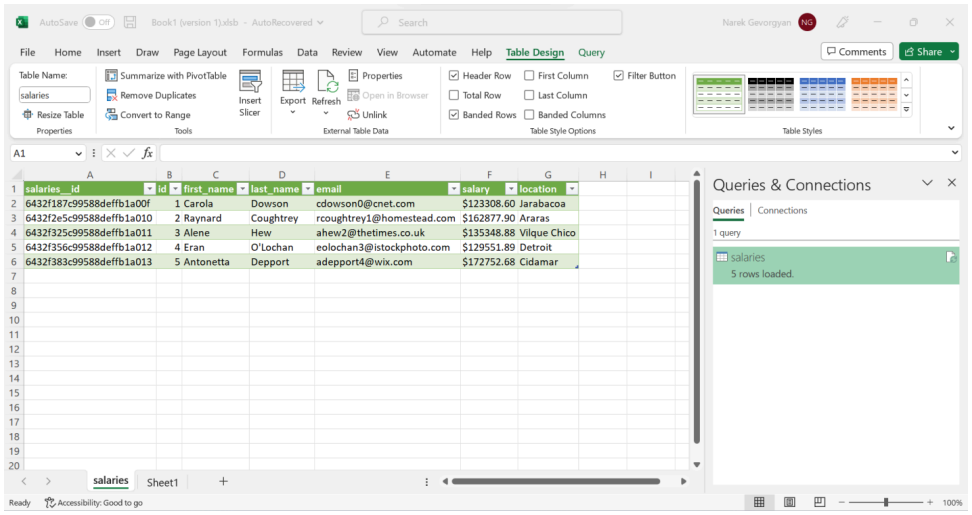
5. Select the data source from the **Data source name (DSN)** drop down menu that is associated with Amazon DocumentDB:



6. Choose the collection from which you want to load data into Excel:



7. Load data into Excel:



Connect to Amazon DocumentDB from Microsoft Power BI Desktop

Topics

- Prerequisites
- Adding Microsoft Power BI Desktop custom connector
- Connecting using the Amazon DocumentDB custom connector
- Configuring Microsoft Power BI Gateway

Prerequisites

Before beginning, ensure that the Amazon DocumentDB ODBC driver is correctly installed.

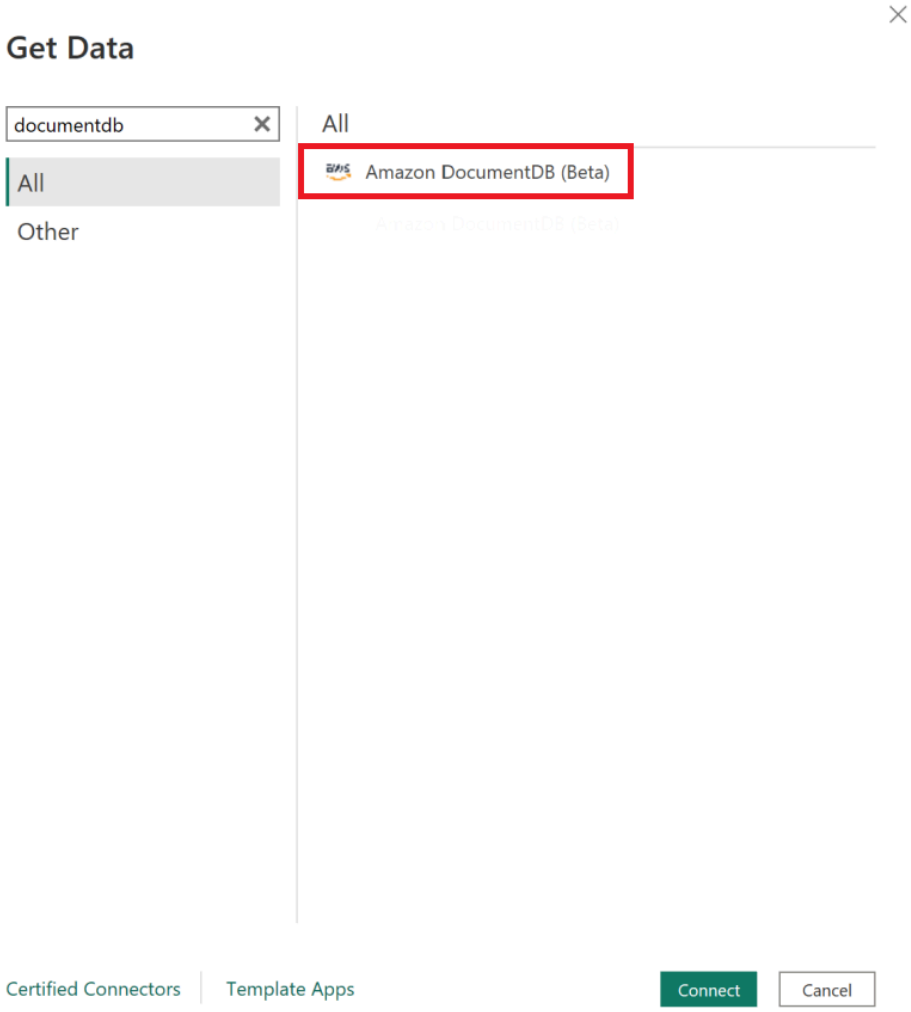
Adding Microsoft Power BI Desktop custom connector

Copy the `AmazonDocumentDBConnector.mez` file to the `<User>\Documents\Power BI Desktop\Custom Connectors\` folder (or to `<User>\OneDrive\Documents\Power BI Desktop\Custom Connectors` if using OneDrive). This will allow Power BI to access custom connector. You can get the connector to Power BI Desktop here. Restart Power BI Desktop to make sure the connector is loaded.

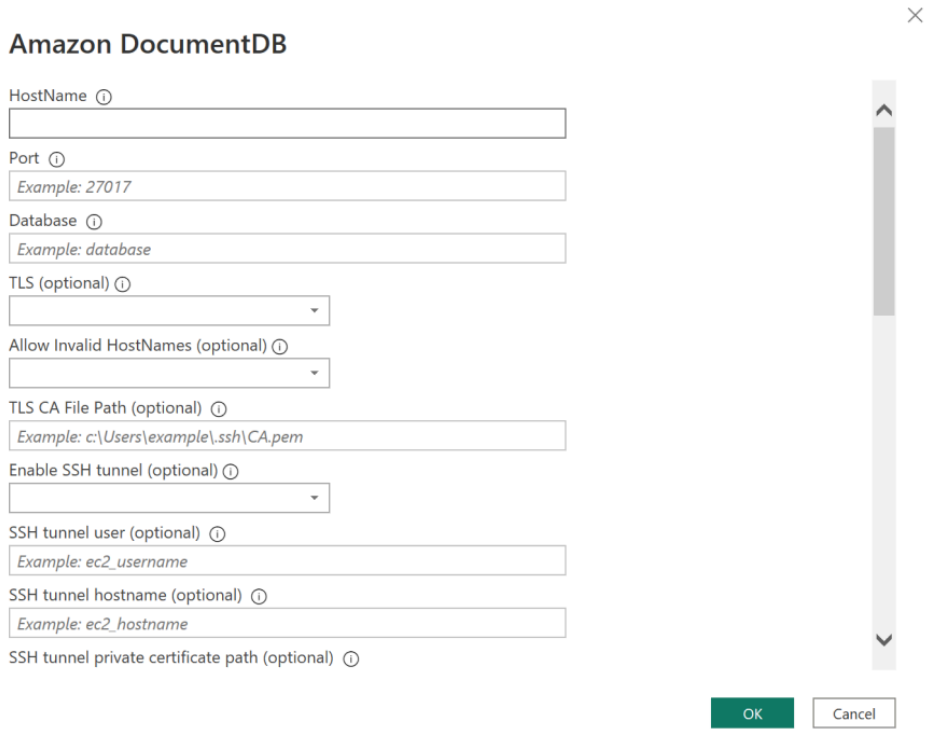
> **Note**
>
> The custom connector only supports Amazon DocumentDB username and password for authentication.

Connecting using the Amazon DocumentDB custom connector

1. Select Amazon DocumentDB (Beta) from **Get Data** and choose **Connect**. If you get a warning for using a third-party service, choose **Continue**.



2. Enter all necessary information to connect to your Amazon DocumentDB cluster, then choose **OK**:



> **Note**
>
> Depending on the configuration of your ODBC driver's Data Source Name (DSN), the SSH connection details screen may not be displayed if you have already provided the necessary information within the DSN settings.

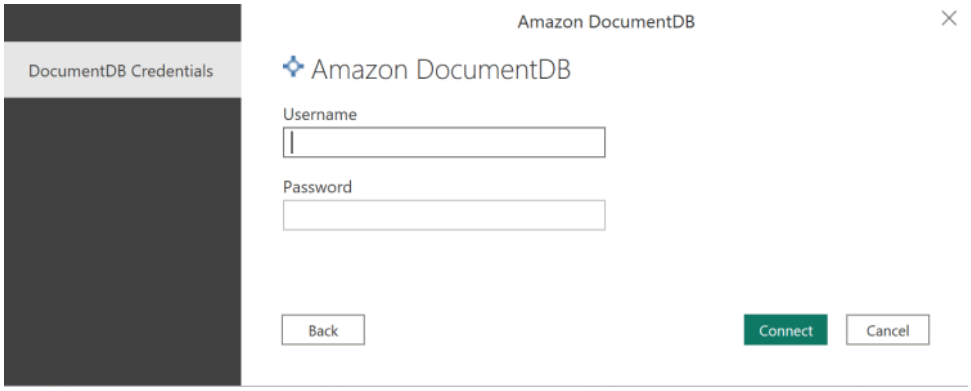
3. Choose the data connectivity mode:

   - **Import** - loads all data and stores the information on disk. The data must be refreshed and reloaded to show data updates.
   - **Direct Query** - does not load data, but does live queries on the data. This means that data does not need to be refreshed and reloaded to show data updates.

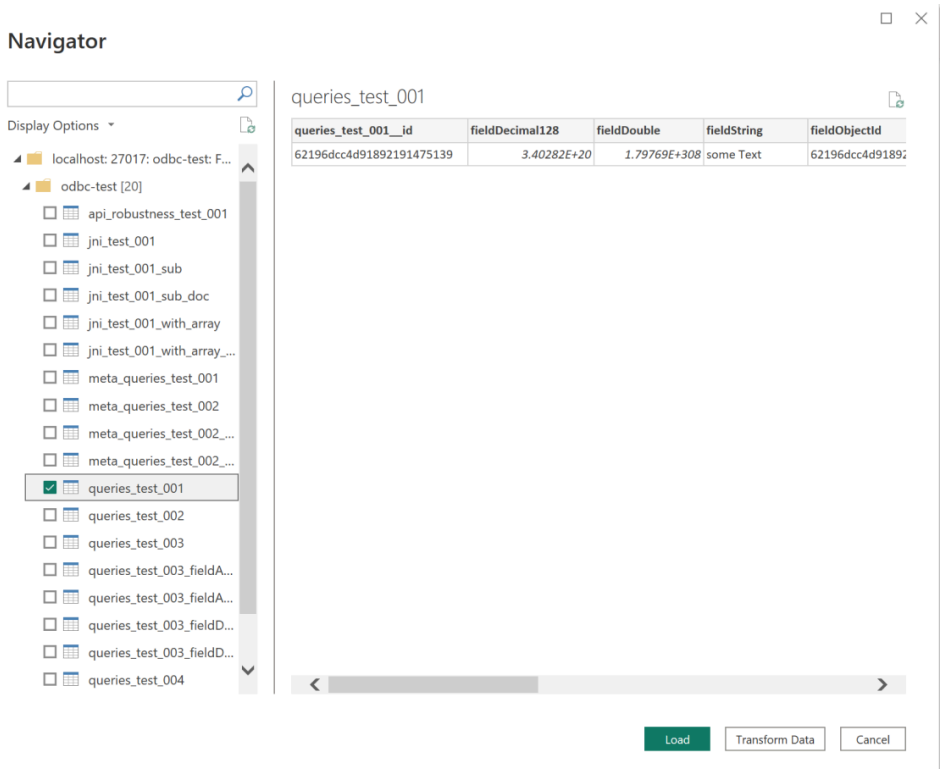


> **Note**
>
> If you are using a very large dataset, importing all of the data may take a longer period of time.

4. If this is the first time connecting to this data source, select the authentication type and input your credentials when prompted. Then choose **Connect**:



5. In the **Navigator** dialog, select the database tables you want, then either choose **Load** to load the data or **Transform Data** to continue transforming the data.



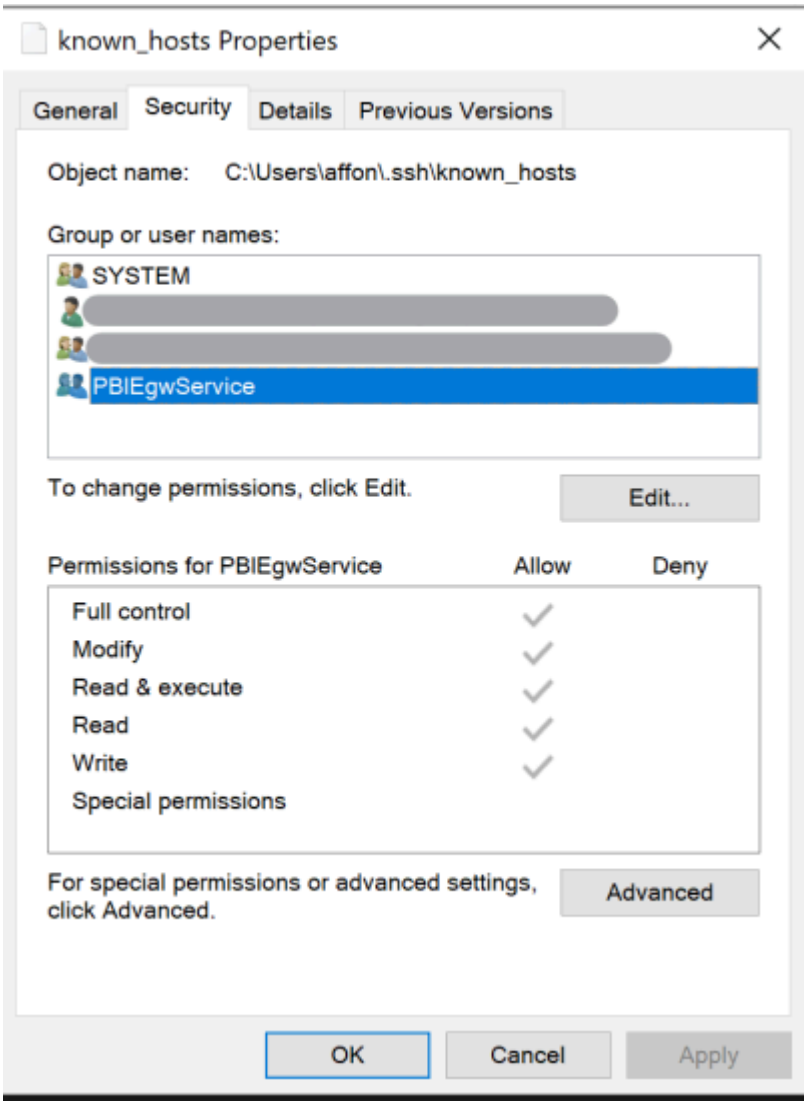
> **Note**
>
> Your data source settings are saved once you connect. To modify them, select **Transform Data** > **Data Source Settings**.

Configuring Microsoft Power BI Gateway

Prerequisites:

- Enure that the custom connector will work with Power BI Gateway.
- Make sure that the ODBC DSN is created in the ODBC data sources in the **System** tab on the machine where Power BI Gateway is installed.

If you are using the internal SSH tunnel feature, the file `known_hosts` needs to be located where the Power BI service account has access to it.



Note

This also applies to any file(s) that you might need to be able to establish a connection to your Amazon DocumentDB cluster, such as a certificate authority (CA) certificate file (pem file).

Automatic schema generation

The ODBC driver is utilizing the Amazon DocumentDB JDBC driver through JNI (Java Native Interface) - making the automatic schema generation feature to work similarly in the JDBC driver. For more information on automatic schema generation, see JDBC automatic schema generation.

For more information about the ODBC driver architecture, see Amazon DocumentDB ODBC driver architecture on GitHub.

SQL support and limitations

The Amazon DocumentDB ODBC driver is a read-only driver that supports a subset of SQL-92 and some common extensions. Refer to the ODBC support and limitations documentation for more information.

Troubleshooting

If you are having problems using the Amazon DocumentDB ODBC driver, refer to the Troubleshooting Guide.

Program with Amazon DocumentDB

The service supports JSON schema validation.

Topics

- Amazon DocumentDB Java programming guide
- Using JSON schema validation

Amazon DocumentDB Java programming guide

This comprehensive guide provides a detailed walk through for working with Amazon DocumentDB using MongoDB's Java drivers, covering essential aspects of database operations and management.

Topics

- Introduction
- Prerequisites
- Data models for this guide
- Connecting to Amazon DocumentDB with a MongoDB Java driver
- Performing CRUD operations in Amazon DocumentDB with Java
- Index management in Amazon DocumentDB with Java
- Event-driven programming with Amazon DocumentDB and Java

Introduction

The guide begins with connectivity, explaining how to establish secure connections to DocumentDB clusters using the MongoDB Java driver. It details the connection string components, SSL/TLS implementation, and various connection options including IAM authentication and connection pooling, along with robust error handling strategies.

In the CRUD (create, read, update, delete) operations section, the guide thoroughly covers document manipulation, demonstrating how to create, read, update, and delete documents using both single and bulk operations. It explains the usage of filters, queries, and various operation options, while emphasizing best practices for error handling and implementing retry logic for improved reliability. The guide also extensively covers index management, detailing the creation and maintenance of different index types including single-field, compound, sparse, and text indexes. It explains how to optimize query performance through proper index selection and usage of the `explain()` function to analyze query execution plans.

The final section focuses on event-driven programming using Amazon DocumentDB's change streams, demonstrating how to implement real-time data change monitoring in Java applications. It covers the implementation of change stream cursors, handling of resume tokens for continuous operation, and time-based operations for historical data processing. Throughout the guide, practical code examples and best practices are provided, making it an invaluable resource for you when building robust Java applications with Amazon DocumentDB.

Prerequisites

Before you begin, make sure you have the following:

- An AWS account with a configured Amazon DocumentDB cluster. For cluster setup, see Getting started with Amazon DocumentDB using Amazon EC2 on the AWS Database Blog.
- Java Development Kit (JDK) installed (we will be using Amazon Corretto 21 for this guide).
- Maven for dependency management.

Data models for this guide

All the example code in this guide assumes a connection to a “ProgGuideData” test database that has a “Restaurants” collection. All the sample codes in this guide work on a restaurant listing system and below is an example of what a document in this system looks like:

```
{
```

```
    "_id": "ab6ad8f119b5bca3efa2c7ae",
    "restaurantId": "REST-CRT9BL",
    "name": "Thai Curry Palace",
    "description": "Amazing Restaurant, must visit",
    "cuisine": "Thai",
    "address": {
        "street": "914 Park Street",
        "city": "Bryan",
        "state": "AL",
        "zipCode": "96865",
        "location": {
            "type": "Point",
            "coordinates": [-25.4619, 8.389]
        }
    },
    "contact": {
        "phone": "(669) 915-9056 x6657"
    },
    "rating": {
        "average": 3.4,
        "totalReviews": 275
    },
    "priceRange": "$",
    "menu": [{
        "category": "Appetizers",
        "items": [{
            "name": "Buffalo Chicken Wings",
            "price": 13.42
        }]
    }],
    "features": [
        "Private Dining"
    ],
    "isActive": false“ michelin”: {“
        star”: 3,
        “ranking_years”: 4
    }
}
```

All code samples showing CRUD, index management, and event-driven programming assumes that you have a `MongoClient` object `dbClient`, a `MongoDatabase` object `connectionDB`, and a `MongoCollection` object `collection`.

> **Note**
>
> All code examples in this guide have been tested with MongoDB Java driver version 5.3.0.

Connecting to Amazon DocumentDB with a MongoDB Java driver

This section provides a step-by-step guide for connecting to Amazon DocumentDB using Java drivers. This will get you started with integrating DocumentDB into your Java applications.

Topics

- Step 1: Set up your project
- Step 2: Create the connection string
- Step 3: Write the connection code
- Step 4: Handle connection exceptions
- Step 5: Running the code
- Connection best practices

Step 1: Set up your project

1. Using Maven, create java project:

```
mvn archetype:generate -DgroupId=com.docdb.guide -DartifactId=my-docdb-project -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2. Add the MongoDB Java driver as a dependency for the project in your 'pom.xml' file:

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-sync</artifactId>
    <version>5.3.0</version>
</dependency>
```

Step 2: Create the connection string

Amazon DocumentDB connection string is essential for establishing a connection between your application and your DocumentDB cluster. This string encapsulates crucial information such as

the cluster endpoint, port, authentication details, and various connection options. To build a DocumentDB connection string, you typically start with the basic format:

```
"mongodb://username:password@cluster-endpoint:port/?[connection options]"
```

You'll need to replace "username" and "password" with your actual credentials. You can find your cluster's endpoint and port number in the AWS Management Console as well as through the AWS CLI. See Finding a cluster's endpoints to find the cluster endpoint for your cluster. The default port for DocumentDB is 27017.

Connection string examples

- Making a connection to DocumentDB using encryption in transit and making sure read requests go to read replicas and write to primary:

```
"mongodb://username:password@cluster-endpoint:27017/?tls=true&
   tlsCAFile=global-bundle.pem&
   readPreference=secondaryPreferred&
   retryWrites=false"
```

- Making a connection to DocumentDB using IAM authentication:

```
"mongodb://cluster-endpoint:27017/?tls=true&
   tlsCAFile=global-bundle.pem&
   readPreference=secondaryPreferred&
   retryWrites=false&
   authSource=%24external&
   authMechanism=MONGODB-AWS"
```

The different options available for the connection string are as follows:

- TLS certificate
- Reading from read replicas
- Write concern and journaling
- RetryWrites
- IAM authentication
- Connection pool
- Connection timeout parameters

TLS certificate

`tls=true|false` — This option enables or disables Transport Layer Security (TLS). By default, encryption in transit is enabled on Amazon DocumentDB cluster and therefore, unless TLS is disabled at the cluster level, the value for this option should be `true`.

When using TLS, the code needs to provide an SSL certificate when creating connection to a DocumentDB cluster. Download the certificate that is required to make the secure connection to the cluster: `global-bundle.pem`. There are two ways to use the `global-bundle.pem` file.

- **Option 1** — Extract all the certificates from the `global-bundle.pem` file and use Java's keytool to store them in a `.jks` file that can be later used in the code. Refer to the Java tab in Connecting with TLS enabled for the script that shows how to do this.
- **Option 2** — Dynamically add the `global-bundle.pem` file in the code, build an in-memory keystore and use `SSLContext` to provide the certificate as part of making the connection.

Reading from read replicas

`replicaSet=rs0&readPreference=secondaryPreferred` — Specifying these two options routes all the read requests to the read replicas, and write requests to the primary instance. Using `replicaSet=rs0` in the connection string enables the MongoDB driver to maintain an automatically updated view of the cluster topology, allowing applications to maintain visibility of current node configurations as instances are added or removed. Not providing these options or specifying `readPreference=primary` sends all the reads and writes to primary instance. For more options for `readPreference`, see Read preference options.

Write concern and journaling

Write concern determines the level of acknowledgment requested from the database for write operations. MongoDB drivers provide an option to tune write concern and journal files. Amazon DocumentDB doesn't expect you to set write concern and journal, and ignores the values sent for `w` and `j` (`writeConcern` and `journal`). DocumentDB always writes data with `writeConcern`: `majority` and `journal`: `true` so the writes are durably recorded on a majority of nodes before sending an acknowledgement to the client.

RetryWrites

`retryWrites` — Starting with engine version 8.0.2, Amazon DocumentDB supports retryable writes. Set `retryWrites=true` to enable. On engine versions earlier than 8.0.2, set this attribute to `false`. For more information, see Retryable writes in Amazon DocumentDB.

IAM authentication

`authSource=%24external and authMechanism=MONGODB-AWS` — These two parameters are used to authenticate using AWS Identity and Access Management. IAM authentication is currently only available in instance-based cluster version 5.0. For more information, see Authentication using IAM identity.

Connection pool

These options are available for connection pooling:

- **`maxPoolSize`** — Sets the maximum number of connections that can be created in the pool. When all connections are in use and a new request comes in, it waits for a connection to become available. The default for MongoDB Java drivers is 100.
- **`minPoolSize`** — Indicates the minimum number of connections that should be maintained in the pool at all times. The default for MongoDB Java drivers is 0.
- **`maxIdleTimeMS`** — Determines how long a connection can remain idle in the pool before being closed and removed. The default for MongoDB Java drivers is 100 milliseconds.
- **`waitQueueTimeoutMS`** — Configures how long a thread should wait for a connection to become available when the pool is at its maximum size. If a connection doesn't become available within this time, an exception is thrown. The default value for MongoDB Java drivers is 120,000 milliseconds (2 minutes).

Connection timeout parameters

Timeout is a mechanism to limit the amount of time an operation or connection attempt can take before it is considered failed. The following timeout parameters are available for preventing indefinite waits and managing resource allocation:

- **`connectTimeoutMS`** — Configures how long the driver will wait to establish a connection to the cluster. The default is 10,000 milliseconds (10 seconds).
- **`socketTimeoutMS`** — Specifies how long the driver will wait for a response from the server for a non-write operation. The default is 0, (no timeout or infinite).

- **`serverSelectionTimeoutMS`** — Species how long the driver will wait to find an available server in the cluster. The default value for this setting is 30 seconds and is sufficient for a new primary instance to be elected during failover.

Step 3: Write the connection code

The following code example shows how to make a TLS connection to Amazon DocumentDB:

- It creates Java's KeyStore and SSLContext> objects.
- It also creates the MongoClientSettings object by passing it to the ConnectionString object. To make TLS connection, you must use the `MongoClientSettings` object to bind the `connectionstring` and `sslcontext`.
- Using MongoClients gets a MongoClient object.

```
public static MongoClient makeDbConnection(String dbName, String DbUserName, String
 DbPassword,
    String DbClusterEndPoint, String keyStorePass) throws Exception {
    MongoClient connectedClient;
    String connectionOptions = "?
replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false";
    String connectionUrl = "mongodb://" + DbUserName + ":" + DbPassword + "@" +
 DbClusterEndPoint + ":27017/" +
        dbName + connectionOptions;

    try {
        KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
        try (FileInputStream fis = new FileInputStream("src/main/resources/certs/
truststore.jks")) {
            trustStore.load(fis, keyStorePass.toCharArray());
            TrustManagerFactory tmf =
 TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
            tmf.init(trustStore);

            SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
            sslContext.init(null, tmf.getTrustManagers(), new SecureRandom());
            ConnectionString connectionString = new ConnectionString(connectionUrl);
            MongoClientSettings settings = MongoClientSettings.builder()
                .applyConnectionString(connectionString)
                .applyToSslSettings(builder - > {
                    builder.enabled(true);
```

```
                    builder.context(sslContext);
                })
                .build();
            connectedClient = MongoClients.create(settings);
        }
        return connectedClient;
    } catch (MongoException e5) {
        throw new RuntimeException(e5);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Step 4: Handle connection exceptions

When working with DocumentDB in Java applications, handling connection exceptions is crucial for maintaining robust and reliable database operations. Properly managing, these exceptions not only help in diagnosing issues quickly, but also ensure that your application can gracefully handle temporary network disruptions or server unavailability, leading to improved stability and user experience. Some of the critical exceptions related to establishing connection include:

- **`MongoException`** — A general exception and could be issued in various scenarios not covered by more specific exceptions. Make sure this exception is handled after all the other specific exceptions as this is a general catch all MongoDB exception.
- **`MongoTimeoutException`** — Issued when an operation times out. For example, querying a nonexistent cluster endpoint.
- **`MongoSocketException`** — Issued for network-related issues. For example, sudden network disconnection during an operation.
- **`MongoSecurityException`** — Issued when authentication fails. For example, connecting with incorrect credentials.
- **`MongoConfigurationException`** — Issued when there is an error in the client configuration. For example, using an invalid connection string.

Step 5: Running the code

The following code sample creates an Amazon DocumentDB connection and prints all the databases:

```
public static void TestConnection() {
```

```
    try (MongoClient mongoClient = makeDbConnection(DATABASE_NAME, DB_USER_NAME,
 DB_PASSWORD, DB_CLUSTER_ENDPOINT, KEYSTORE_PASSWORD)) {
        List < String > databases = mongoClient.listDatabaseNames().into(new ArrayList
 < > ());
        System.out.println("Databases: " + databases);
    } catch (MongoException e) {
        System.err.println("MongoDB error: " + e.getMessage());
        throw new RuntimeException(e);
    }
}
```

Connection best practices

The following are best practices to consider when connecting to Amazon DocumentDB with a MongoDB Java driver:

- Always close your `MongoClient` when you no longer need the client to release resources.
- Handle exceptions appropriately and implement proper error logging.
- Use environment variables or AWS Secrets Manager to store sensitive information like usernames and passwords.

Performing CRUD operations in Amazon DocumentDB with Java

This section discusses performing CRUD (create, read, update, delete) operation in Amazon DocumentDB using MongoDB Java drivers.

Topics

- Creating and inserting documents in a DocumentDB collection
- Reading and retrieving data from a DocumentDB collection
- Updating existing documents in a DocumentDB collection
- Removing documents from a DocumentDB collection
- Error handling with retry logic

Creating and inserting documents in a DocumentDB collection

Inserting documents into Amazon DocumentDB allows you to add new data to your collections. There are several ways to perform insertions, depending on your needs and the volume of data you're working with. The most basic method for inserting an individual document to the collection

is `insertOne()`. For inserting multiple documents at once, you can use the `insertMany()` method, which allows you to add an array of documents in a single operation. Another method for inserting many documents in a DocumentDB collection is `bulkWrite()`. In this guide, we discuss all of these methods for creating documents in a DocumentDB collection.

`insertOne()`

Let's begin by examining how to insert an individual document into an Amazon DocumentDBB collection. Inserting a single document is accomplished by using the `insertOne()` method. This method takes a BsonDocument for insertion and returns an `InsertOneResult` object that can be used to get the object id of the new inserted document. The example code below shows inserting one restaurant document into the collection:

```
Document article = new Document()
    .append("restaurantId", "REST-21G145")
    .append("name", "Future-proofed Intelligent Bronze Hat")
    .append("cuisine", "International")
    .append("rating", new Document()
        .append("average", 1.8)
        .append("totalReviews", 267))
    .append("features", Arrays.asList("Outdoor Seating", "Live Music"));

try {
    InsertOneResult result = collection.insertOne(article);
    System.out.println("Inserted document with the following id: " +
 result.getInsertedId());
} catch (MongoWriteException e) {
    // Handle duplicate key or other write errors
    System.err.println("Failed to insert document: " + e.getMessage());
    throw e;
} catch (MongoException e) {
    // Handle other MongoDB errors
    System.err.println("MongoDB error: " + e.getMessage());
    throw e;
}
```

When using `insertOne()`, make sure to include appropriate error handling. For instance, in the above code, "`restaurantId`" has a unique index and therefore running this code again will raise the following `MongoWriteException`:

```
Failed to insert document: Write operation error on server
 docdbCluster.docdb.amazonaws.com:27017.
```

```
Write error: WriteError{code=11000, message='E11000 duplicate key error collection:
 Restaurants index: restaurantId_1', details={}}.
```

insertMany()

The primary methods used for inserting many document into a collection are insertMany() and `bulkWrite()`.

The `insertMany()` method is the simplest way to insert multiple documents in a single operation. It accepts a list of documents and inserts them into the collection. This method is ideal when you're inserting a batch of new documents that are independent of each other and don't require any special processing or mixed operations. The following code shows reading JSON documents from a file and inserting them into the collection. The `insertMany()` function returns an [InsertManyResult]`InsertManyResult` object that can be used to get the IDs of all the inserted documents.

```
// Read JSON file content
String content = new String(Files.readAllBytes(Paths.get(jsonFileName)));
JSONArray jsonArray = new JSONArray(content);

// Convert JSON articles to Documents
List < Document > restaurants = new ArrayList < > ();
for (int i = 0; i < jsonArray.length(); i++) {
    JSONObject jsonObject = jsonArray.getJSONObject(i);
    Document doc = Document.parse(jsonObject.toString());
    restaurants.add(doc);
}
//insert documents in collection
InsertManyResult result = collection.insertMany(restaurants);

System.out.println("Count of inserted documents: " + result.getInsertedIds().size());
```

bulkWrite()

The `bulkWrite()` method allows to perform multiple write operations (insert, update, delete) in a single batch. You can use `bulkWrite()` when you need to perform different types of operations in a single batch, such as inserting some documents while updating others. `bulkWrite()` support two types of batch write, ordered and unordered:

- *Ordered operations* — (default) Amazon DocumentDB processes the write operations sequentially, and stops at the first error it encounters. This is useful when the order of operations

matters, such as when later operations depend on earlier ones. However, ordered operations are generally slower then unordered operations. With ordered operations, you must address the case where the batch stops at the first error, potentially leaving some operations unprocessed.

- *Unordered operations* — Allows Amazon DocumentDB to process inserts as a single execution in the database. If an error occurs with one document, the operation continues with the remaining documents. This is particularly useful when you're inserting large amounts of data and can tolerate some failures, such as during data migration or bulk imports where some documents might fail due to duplicate keys. With unordered operations, you must address partial success scenarios where some operations succeed while others fail.

When working with the `bulkWrite()` method, there are some essential classes that are required. First, the `WriteModel` class serves as the base class for all write operations and with specific implementations like `InsertOneModel`, `UpdateOneModel`, `UpdateManyModel`, `DeleteOneModel`, and `DeleteManyModel` handling different types of operations.

The `BulkWriteOptions` class is necessary for configuring the behavior of bulk operations, such as setting ordered/unordered execution or bypassing document validation. The `BulkWriteResult` class provides detailed information about the execution results, including counts of inserted, updated, and deleted documents.

For error handling, the `MongoBulkWriteException` class is crucial as it contains information about any failures during the bulk operation, while the `BulkWriteError` class provides specific details about individual operation failures. The following code shows an example of inserting a list of documents, as well as updating and deleting a single document, all within the execution of single `bulkWrite()` method call. The code also shows how to work with `BulkWriteOptions` and `BulkWriteResult`, as well as proper error handling of the `bulkWrite()` operation.

```
List < WriteModel < Document >> bulkOperations = new ArrayList < > ();
// get list of 10 documents representing 10 restaurants
List < Document > restaurantsToInsert = getSampleData();

for (Document doc: restaurantsToInsert) {
    bulkOperations.add(new InsertOneModel < > (doc));
}
// Update operation
bulkOperations.add(new UpdateOneModel < > (
    new Document("restaurantId", "REST-Y2E9H5"),
    new Document("", new Document("stats.likes", 20))
    .append("", new Document("rating.average", 4.5))));
```

```
// Delete operation
bulkOperations.add(new DeleteOneModel < > (new Document("restaurantId", "REST-
D2L431")));

// Perform bulkWrite operation
try {
    BulkWriteOptions options = new BulkWriteOptions()
        .ordered(false); // Allow unordered inserts

    BulkWriteResult result = collection.bulkWrite(bulkOperations, options);

    System.out.println("Inserted: " + result.getInsertedCount());
    System.out.println("Updated: " + result.getModifiedCount());
    System.out.println("Deleted: " + result.getDeletedCount());
} catch (MongoBulkWriteException e) {
    System.err.println("Bulk write error occurred: " + e.getMessage());
    // Log individual write errors
    for (BulkWriteError error: e.getWriteErrors()) {
        System.err.printf("Error at index %d: %s (Code: %d)%n", error.getIndex(),
 error.getMessage(),
            error.getCode());

        // Log the problematic document
        Document errorDoc = new Document(error.getDetails());
        if (errorDoc != null) {
            System.err.println("Problematic document: " + errorDoc);
        }
    }
} catch (Exception e) {
    System.err.println("Error during bulkWrite: " + e.getMessage());
}
```

Retryable writes

Unlike MongoDB, Amazon DocumentDB doesn't support retryable writes. As a result, you must implement custom retry logic in their applications, particularly for handling network issues or temporary service unavailability. A well-implemented retry strategy, typically, involves increasing the delay between retry attempts and limiting the total number of retries. See Error handling with retry logic below for a code sample of building retry logic with error handling.

Reading and retrieving data from a DocumentDB collection

Querying documents in Amazon DocumentDB revolves around several key components that allow you to precisely retrieve and manipulate data. The `find()` method is the fundamental querying APIs in MongoDB Java drivers. It allows for complex data retrieval with numerous options for filtering, sorting, and projecting results. Besides the `find()` method, `Filters` and `FindIterable` are two other fundamental components that provide the building blocks for query operations in MongoDB Java drivers.

The `Filters` class is a utility class in the MongoDB Java driver that provides a fluent API for constructing query filters. This class offers static factory methods that create instances of `Bson` objects representing various query conditions. The most commonly used methods include `eq()` for equality comparisons, `gt()`, `lt()`, `gte()`, and `lte()` for numeric comparisons, `and()` and `or()` for combining multiple conditions, `in()` and `nin()` for array membership tests, and `regex()` for pattern matching. The class is designed to be type-safe and provides better compile-time checking compared to raw document-based queries, making it the preferred approach for constructing DocumentDB queries in Java applications. Error handling is robust, with clear exceptions thrown for invalid filter constructions.

`FindIterable` is a specialized interface designed to handle the result of the `find()` method. It provides a rich set of methods for refining and controlling query execution, offering a fluent API for method chaining. The interface includes essential query modification methods such as `limit()` for restricting the number of returned documents, `skip()` for pagination, `sort()` for ordering results, `projection()` for selecting specific fields, and `hint()` for index selection. The batch, skip, and limit operations in `FindIterable` are essential pagination and data management tools that help control how documents are retrieved and processed from the database.

Batching (`batchSize`) controls how many documents DocumentDB returns to the client in a single network round trip. When you set a batch size, DocumentDB doesn't return all matching documents at once but instead returns them in groups of the specified batch size.

Skip allows you to offset the starting point of your results, essentially telling DocumentDB to skip over a specified number of documents before beginning to return matches. For example, `skip(20)` will bypass the first 20 matching documents. This is commonly used in pagination scenarios where you want to retrieve subsequent pages of results.

Limit restricts the total number of documents that can be returned from a query. When you specify `limit(n)`, DocumentDB will stop returning documents after it has returned 'n' documents, even if there are more matches in the database.

FindIterable supports both iterator and cursor patterns when retrieving documents from Amazon DocumentDB. The benefit of using FindIterable as iterator is that it allows lazy loading of documents and only fetches documents when requested by the application. Another benefit of using iterator is that you are not responsible for maintaining the connection to the cluster and thus no explicit closing of the connection is required.

FindIterable also provide support for MongoCursor which allows cursor patterns to be used when working with Amazon DocumentDB queries. MongoCursor is a MongoDB Java driver-specific implementation that provides control over database operations and resource management. It implements the AutoCloseable interface, allowing for explicit resource management through try-with-resources blocks, which is crucial for properly closing database connections and freeing server resources. By default, the cursor times out in 10 minutes and DocumentDB does not give you the option of changing this time out behavior. When working with batched data, make sure to retrieve the next batch of data before the cursor times out. One key consideration when using MongoCursor is that it requires explicit closing to prevent resource leaks.

In this section, several examples are presented for find(), Filters and FindIterable.

The following code example shows how to use find() to retrieve a single document using its "restaurantId" field:

```
Document filter = new Document("restaurantId", "REST-21G145");
Document result = collection.find(filter).first();
```

Even though using Filters allows for better compile time error checking, the java driver also allows you to specify a Bson filter directly in the find() method. The following example code passes Bson document to find():

```
result = collection.find(new Document("$and", Arrays.asList(
    new Document("rating.totalReviews", new Document("$gt", 1000)),
    new Document("priceRange", "$$"))))
```

The next example code shows several examples of using the Filters class with find():

```
FindIterable < Document > results;

// Exact match
results = collection.find(Filters.eq("name", "Thai Curry Palace"));
```

```
// Not equal
results = collection.find(Filters.ne("cuisine", "Thai"));

// find an element in an array
results = collection.find(Filters.in("features", Arrays.asList("Private Dining")));

// Greater than
results = collection.find(Filters.gt("rating.average", 3.5));

// Between (inclusive)
results = collection.find(Filters.and(
    Filters.gte("rating.totalReviews", 100),
    Filters.lte("rating.totalReviews", 200)));
// AND
results = collection.find(Filters.and(
    Filters.eq("cuisine", "Thai"),
    Filters.gt("rating.average", 4.5)));

// OR
results = collection.find(Filters.or(
    Filters.eq("cuisine", "Thai"),
    Filters.eq("cuisine", "American")));


// All document where the Field exists
results = collection.find(Filters.exists("michelin"));

// Regex
results = collection.find(Filters.regex("name", Pattern.compile("Curry",
 Pattern.CASE_INSENSITIVE)));

// Find all document where the array contain the list of value regardless of its order
results = collection.find(Filters.all("features", Arrays.asList("Private Dining",
 "Parking")));

// Array size
results = collection.find(Filters.size("features", 4));
```

The following example shows how to chain the operations of `sort()`, `skip()`, `limit()`, and `batchSize()` on a `FindIterable` object. The order of how these operations is provided will influence the performance of your query. As a best practice, the order of these operations should be `sort()`, `projection()`, `skip()`, `limit()` and `batchSize()`.

```
FindIterable < Document > results = collection.find(Filters.gt("rating.totalReviews",
 1000))
    // Sorting
    .sort(Sorts.orderBy(
        Sorts.descending("address.city"),
        Sorts.ascending("cuisine")))
    // Field selection
    .projection(Projections.fields(
        Projections.include("name", "cuisine", "priceRange"),
        Projections.excludeId()))

    // Pagination
    .skip(20)
    .limit(10)
    .batchSize(2);
```

The following example code shows creating an iterator on `FindIterable`. It uses Java's `forEach` construct to traverse through the result set.

```
collection.find(Filters.eq("cuisine", "American")).forEach(doc ->
 System.out.println(doc.toJson()));
```

In the last `find()` code example, it shows how to use `cursor()` for document retrieval. It creates the cursor in the try block which ensures that the cursor will get closed when the code exits the try block.

```
try (MongoCursor < Document > cursor = collection.find(Filters.eq("cuisine",
 "American"))
    .batchSize(25)
    .cursor()) {
    while (cursor.hasNext()) {
        Document doc = cursor.next();
        System.out.println(doc.toJson());
    }
} // Cursor automatically closed
```

Updating existing documents in a DocumentDB collection

Amazon DocumentDB provides flexible and powerful mechanisms for modifying existing documents and inserting new ones when they don't exist. The MongoDB Java driver offers several

methods for updates: `updateOne()` for single document updates, `updateMany()` for multiple document updates, and `replaceOne()` for complete document replacement. Beside these three methods, `Updates`, `UpdateOptions`, and `UpdateResult` are other fundamental components that provide the building blocks for update operations in MongoDB Java drivers.

The `Updates` class in the MongoDB Java driver is a utility class that provides static factory methods for creating update operators. It serves as the primary builder for constructing update operations in a type-safe and readable manner. Basic methods like `set()`, `unset()`, and `inc()` allows direct modification of the documents. The power of this class becomes evident when combining multiple operations using the `Updates.combine()` method which allows multiple update operations to be executed atomically, ensuring data consistency.

`UpdateOptions` is a powerful configuration class in MongoDB's Java driver that provides essential customization capabilities for document update operations. Two important aspects of this class is providing upsert and array filter support for update operations. The upsert feature, enabled through `upsert(true)`, allows for the creation of new documents when no matching documents are found during an update operation. Through `arrayFilters()`, the update operation can precisely update array elements that meet specific criteria.

`UpdateResult` in MongoDB's Java driver provides the feedback mechanism detailing the outcome of an update operation. This class encapsulates three key metrics: the number of documents matched by the update criteria (`matchedCount`), the number of documents actually modified (`modifiedCount`), and information about any upserted documents (`upsertedId`). Understanding these metrics is essential for proper error handling, verification of update operations, and maintaining data consistency in applications.

Update and replace a single document

In DocumentDB, updating a single document can be accomplished using the updateOne() method. This method take a filter parameter, usually provided by the `Filters` class, to identify the document to be updated, an `Update` parameter that determines which field(s) to be updated, and an optional `UpdateOptions` parameter to set different options for the update. Using the `updateOne()` method will only update the first document that matches the selection criteria. The following example code updates a single field of one document:

```
collection.updateOne(Filters.eq("restaurantId", "REST-Y2E9H5"),
    Updates.set("name", "Amazing Japanese sushi"));
```

To update multiple fields in one document, use `updateOne()` with `Update.combine()` as shown in the following example. This example also shows how to add an item to an array in the document.

```
List<Bson> updates = new ArrayList<>();
// Basic field updates
updates.add(Updates.set("name", "Shanghai Best"));
// Array operations
updates.add(Updates.addEachToSet("features", Arrays.asList("Live Music")));
// Counter updates
updates.add(Updates.inc("rating.totalReviews", 10));
// Combine all updates
Bson combinedUpdates = Updates.combine(updates);
// Execute automic update with one call
collection.updateOne(Filters.eq("restaurantId","REST-1J83NH"), combinedUpdates);
```

The following code example demonstrates how to update a document in the database. If the specified document doesn't exist, the operation will automatically insert it as a new document instead. This code also shows how to use the metrics available via the `UpdateResult` object.

```
Bson filter = Filters.eq("restaurantId", "REST-0Y9GL0");
Bson update = Updates.set("cuisine", "Indian");
// Upsert operation
UpdateOptions options = new UpdateOptions().upsert(true);
UpdateResult result = collection.updateOne(filter, update, options);

if (result.getUpsertedId() != null) {
    System.out.println("Inserted document with _id: " + result.getUpsertedId());
} else {
     System.out.println("Updated " + result.getModifiedCount() + " document(s)");
}
```

The following code example demonstrates how to completely replace an existing document with a new document using the `replaceOne()` method, rather than updating individual fields. The `replaceOne()` method overwrites the entire document, preserving only the `_id` field of the original. If multiple documents match the filter criteria, only the first encountered document is replaced.

```
Document newDocument = new Document()
                .append("restaurantId", "REST-0Y9GL0")
                .append("name", "Bhiryani Adda")
```

```
                .append("cuisine", "Indian")
                .append("rating", new Document()
                        .append("average", 4.8)
                        .append("totalReviews", 267))
                .append("features", Arrays.asList("Outdoor Seating", "Live Music"));

UpdateResult result = collection.replaceOne(
                    Filters.eq("restaurantId", "REST-0Y9GL0"),
                    newDocument);
System.out.printf("Modified %d document%n", result.getModifiedCount());
```

Update multiple documents

There are two ways to update multiple document in a collection simultaneously. You can use the `updateMany()` method or use the `UpdateManyModel` with the `bulkWrite()` method. The `updateMany()` method uses a filter parameter to select documents for update, the `Update` parameter to identify the fields to be updated, and an optional `UpdateOptions` parameter to specify update options.

The following example code demonstrates the usage of the `updateMany()` method:

```
Bson filter = Filters.and(
    Filters.in("features", Arrays.asList("Private Dining")),
    Filters.eq("cuisine", "Thai"));
UpdateResult result1 = collection.updateMany(filter, Updates.set("priceRange", "$$$"));
```

The following example code demonstrates the `bulkWrite()` method using the same update:

```
BulkWriteOptions options = new BulkWriteOptions().ordered(false);
List < WriteModel < Document >> updates = new ArrayList < > ();
Bson filter = Filters.and(
    Filters.in("features", Arrays.asList("Private Dining")),
    Filters.eq("cuisine", "Indian"));
Bson updateField = Updates.set("priceRange", "$$$");
updates.add(new UpdateManyModel < > (filter, updateField));
BulkWriteResult result = collection.bulkWrite(updates, options);
System.out.printf("Modified %d document%n", result.getModifiedCount());
```

Removing documents from a DocumentDB collection

The MongoDB Java driver offers `deleteOne()` for removing a single document and `deleteMany()` for removing multiple documents that match specific criteria. Just like update,

delete operation can also be used with the `bulkWrite()` method. Both `deleteOne()` and `deleteMany()` return a `DeleteResult` object that provides information about the operation's outcome, including the count of documents deleted. The following is an example of using `deleteMany()` to remove multiple documents:

```
Bson filter = Filters.and(
    Filters.eq("cuisine", "Thai"),
    Filters.lt("rating.totalReviews", 50));
DeleteResult result = collection.deleteMany(filter);
System.out.printf("Deleted %d document%n", result.getDeletedCount());
```

Error handling with retry logic

A robust error handling strategy for Amazon DocumentDB should implement categorization of errors into retryable (like network timeouts, connection issues) and non-retryable (like authentication failures, invalid queries). For operation failures due of errors that should be retried, it should implement a time delay between each retry as well as max retry attempts. The CRUD operations should be in a try-catch block that catches `MongoException` and its subclasses. Additionally, it should include monitoring and logging of errors for operational visibility. The following is sample code that shows how to implement retry error handling:

```
int MAX_RETRIES = 3;
int INITIAL_DELAY_MS = 1000;
int retryCount = 0;

while (true) {
    try {
        crud_operation(); //perform crud that will throw MongoException or one of its
 subclass
        break;
    } catch (MongoException e) {
        if (retryCount < MAX_RETRIES) {
            retryCount++;
            long delayMs = INITIAL_DELAY_MS * (long) Math.pow(2, retryCount - 1);
            try {
                TimeUnit.MILLISECONDS.sleep(delayMs);
            } catch (InterruptedException t) {
                Thread.currentThread().interrupt();
                throw new RuntimeException("Retry interrupted", t);
            }
            continue;
```

```
        } else
            throw new RuntimeException("Crud operation failed", e);
    }
}
```

Index management in Amazon DocumentDB with Java

Indexes allow efficient retrieval of data from an Amazon DocumentDB collection. Without indexes, DocumentDB must scan every document in the collection to return results that satisfy a given query. This topic provides information on how to create, drop, and list indexes using the MongoDB Java drivers. It also discusses how to determine if a particular index is being used in the query and how to give hints to Amazon DocumentDB to use a specific index.

Topics

- Creating indexes with Java
- Dropping indexes
- Determining index selection and providing index hint

Amazon DocumentDB supports many types of indexes. For a comprehensive overview of all the supported indexes, see How to index on Amazon DocumentDB on the AWS Database Blog.

Creating indexes with Java

There are two mechanisms for creating indexes in Amazon DocumentDB using MongoDB Java drivers: through `runCommand()`, and by either the `createIndex()` method for a single index or the `createIndexes()` method for multiple indexes. One reason for using the `createIndex()` and `createIndexes()` methods is that you can build better error handling by catching specific errors related to index creation. Another reason for using these method over `runCommand()` is that the MongDB Java driver provides a rich set of supporting classes for index creation and manipulation. Note that these supporting classes can only be used when you are using the `createIndex()` or `createIndexes()` methods. There are three supporting classes:

- **`Indexes`** — This class serves as a utility class offering static factory methods for creating various types of indexes. It simplifies the process of creating complex index definitions and is commonly used in conjunction with other index-related classes.
- **`IndexModel`** — This is a fundamental class that encapsulates both the index keys definition and its options. It represents a complete index specification, combining what to index (the keys) with how to index (the options). This class is particularly useful when creating multiple indexes

simultaneously, as it allows you to define a collection of index specifications that can be passed to the `createIndexes()` method.

- **`IndexOptions`** — This is a comprehensive configuration class that provides a rich set of methods for customizing index behavior. It includes settings for unique indexes, sparse indexes, expiration time (TTL), and partial filter expressions. Through method chaining, you can configure multiple options like background index building and unique constraints.

Create single index

This example shows how to create a single index using the `createIndex()` method in the background. To understand background and foreground index creation, refer to Index build types. The following code example uses `IndexOptions` to create a unique index with a name of "unique_restaurantId_idx" in the background. This `IndexOptions` object is then passed to the `createIndex()` method.

```
collection.createIndex(
    Indexes.ascending("restaurantId"),
    new IndexOptions()
        .unique(true)
        .name("unique_restaurantId_idx")
        .background(true));
```

Create multiple indexes

This example creates multiple indexes using the `createIndexes()` method. It first builds the option for each index by using the `IndexModel` object and then passes a list of `IndexModel` objects to the `createIndexes()` method. The the following code example shows how to create a composite index by using the `Indexes` utility class. This class is also used to specify if you want to create an index using ascending or descending sort order. After creating multiple indexes, it verifies index creation by calling the `listIndexes()` method.

```
// Single Field Index on cuisine
IndexModel singleIndex = new IndexModel(
    Indexes.ascending("cuisine"),
    new IndexOptions().name("cuisine_idx"));

// Compound Index
IndexModel compoundIndex = new IndexModel(
    Indexes.compoundIndex(
```

```
        Indexes.ascending("address.state"),
        Indexes.ascending("priceRange")),
    new IndexOptions().name("location_price_idx"));

// Build a list of IndexModel for the indexes
List < IndexModel > indexes = Arrays.asList(
    singleIndex,
    compoundIndex
);

collection.createIndexes(indexes);

// Verify created indexes
collection.listIndexes().forEach(index - > System.out.println("Created index: " +
 index.toJson()));
```

Create sparse and partial indexes

This example shows creates a sparse and a partial index by creating an `IndexModel` for each type of index.

```
// Sparse Index Model, this will identify only those documents that have a
// michelin star rating
IndexModel sparseIndex = new IndexModel(
    Indexes.ascending("michelin.star"),
    new IndexOptions()
    .name("michelin_sparse_idx")
    .sparse(true));

// Partial Index Model where the restaurant is active and has a rating of 4 and above
IndexModel partialIndex = new IndexModel(
    Indexes.ascending("rating.average"),
    new IndexOptions()
    .name("high_rated_active_idx")
    .partialFilterExpression(
        Filters.and(
            Filters.eq("isActive", true),
            Filters.gte("rating.average", 4.0))));
```

Create a text index

This example shows how to create a text index. Only one text index is allowed on a collection but that one text index can be a compound index covering multiple fields. When using multiple fields

in the text index, you can also assign weights to each of the fields in the index. Text indexes on array fields are not supported by Amazon DocumentDB and even though you can use up to 30 fields in the compound text index, only three fields can be assigned a weight.

```
IndexModel textIndex = new IndexModel(
    new Document()
    .append("name", "text")
    .append("description", "text")
    .append("cuisine", "text"),
    new IndexOptions()
    .name("restaurant_text_idx")
    .weights(new Document()
        .append("name", 10) // Restaurant name gets highest weight
        .append("description", 5) // Description get medium weight
        .append("cuisine", 2) // Cuisine type gets low weight
    ));

collection.createIndex(textIndex.getKeys(), textIndex.getOptions());
```

Create an index using `runCommand()`

Amazon DocumentDB supports parallel index creation to decrease the time it takes to create indexes. Parallel indexing uses multiple concurrent workers. The default workers used for index creation is two. For more information, see Unlock the power of parallel indexing in Amazon DocumentDB on the AWS Database Blog. Currently, MongDB Java drivers do not support specifying the worker option when you are using `createIndex()` or `createIndexes()` and therefore the only way to specify workers is through the `runCommand`. The following code example demonstrates how to use `runCommand` to create an index that increase the worker to four:

```
Document command = new Document("createIndexes", "Restaurants")
    .append("indexes", Arrays.asList(
        new Document("key", new Document("name", 1))
        .append("name", "restaurant_name_idx")
        .append("workers", 4) // Specify number of workers
    ));

Document commendResult = connectedDB.runCommand(command);
```

Dropping indexes

The MongoDB Java driver provides multiple methods to drop indexes, catering to different scenarios and your preferences. You can drop indexes by name, by key specification, or drop all indexes at once. The methods `dropIndex()` and `dropIndexes()` can be invoked on a collection object to drop an index. When dropping an index by name, you should ensure they use the correct index name which may not always be intuitive, especially for compound or auto-generated indexes. Attempting to drop a non-existent index will result in a `MongoCommandException`. The `default_id` index cannot be dropped as it ensures document uniqueness within the collection.

The following code example shows demonstrates how to drop an index by providing field name where the index is created or by deleting all the indexes:

```
String indexName = "unique_restaurantId_idx";
Document keys = new Document("cuisine", 1);
// Drop index by name
collection.dropIndex(indexName);

// Drop index by keys
collection.dropIndex(keys);

// Drop all indexes
collection.dropIndexes();
```

When dropping indexes using multiple keys, make sure there is a compound index containing all the keys specified and the order of the keys is correct. The above index creation example code shows a compound key on "cuisine" and features. If you try to drop that compound key but the order is not what was used for creation, then a MongoCommnadException error occurs as follows:

```
Document keys = new Document("features", 1)
    .append("cuisine", 1);
try {
    // Drop index by keys
    collection.dropIndex(keys);
    System.out.println("Successfully dropped index with keys: " + keys.toJson());

} catch (MongoCommandException commErr) {
    System.out.println("Error dropping index: " + commErr.getErrorMessage());
    throw new RuntimeException("MongoCommandException was thrown while dropping index",
 commErr);
}
```

The following error is displayed:

```
Error dropping index: Cannot drop index: index not found.
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.819 sec <<< FAILURE!
com.amazon.docdb.guide.DocDBGuideTest.testindexGuide()  Time elapsed: 0.817 sec  <<<
 FAILURE!
org.opentest4j.AssertionFailedError: Unexpected exception thrown:
 java.lang.RuntimeException: MongoCommandException was thrown while dropping index
```

Determining index selection and providing index hint

Working with the explain functionality in Amazon DocumentDB is crucial for you to understand query performance and index usage. When executing a query, you can append the `explain()` method to obtain detailed information about the query plan, including which indexes, if any, are being used. The `explain()` output provides insights into the query execution stages, the number of documents examined, and the time taken for each stage. This information is invaluable for identifying whether a particular index is being used effectively or if the query could benefit from a different index structure.

The `explain()` method can be chained with the `find()` method. The `explain()` method can take an optional `ExplainVerbosity` enum that determines the level of verbosity that is returned by `explain()`. At this time, only the `EXECUTION_STATS` and `QUERY_PLANNER` enumerators are supported by DocumentDB. The following code example shows how to get query planner for a specific query:

```
// Query we want to analyze
Document query = new Document()
    .append("cuisine", "Thai")
    .append("rating.average", new Document("$gte", 4.0));


Document allPlansExplain =
 collection.find(query).explain(ExplainVerbosity.QUERY_PLANNER);
System.out.println("All Plans Explain:\n" + allPlansExplain.toJson());
```

The following JSON document is returned for the query planner verbosity level:

```
{
  "queryPlanner": {
    "plannerVersion": 1,
```

```
    "namespace": "ProgGuideData.Restaurants",
    "winningPlan": {
      "stage": "IXSCAN",
      "indexName": "cuisine_idx",
      "direction": "forward"
    }
  },
  "serverInfo": {
    "host": "guidecluster3",
    "port": 27017,
    "version": "5.0.0"
  },
  "ok": 1,
  "operationTime": {
    "$timestamp": {
      "t": 1739221668,
      "i": 1
    }
  }
}
```

You have several options to influence or force Amazon DocumentDB to use a specific index. The `hint()` and `hintString()` methods allow you to override the query optimizer's default index selection behavior by explicitly specifying which index should be used for a query. While DocumentDB's query optimizer generally makes good choices for index selection, there are scenarios where forcing a specific index through `hint()` or `hintString()` can be beneficial, such as when dealing with skewed data, or testing index performance.

The following code example forces the use of compound index “cuisine_features_idx” for the same query that was run in the above code:

```
// Query we want to analyze
Document query = new Document()
    .append("cuisine", "Thai")
    .append("rating.average", new Document("$gte", 4.0));

List < Document > queryDocs = new ArrayList < > ();
collection.find(query).hintString("cuisine_features_idx").forEach(doc - >
 queryDocs.add(doc));
```

Event-driven programming with Amazon DocumentDB and Java

Event-driven programming in the context of Amazon DocumentDB represents a powerful architectural pattern where database changes serve as the primary event generators that trigger subsequent business logic and processes. When records are inserted, updated, or deleted in a DocumentDB collection, these changes act as events that automatically initiate various downstream processes, notifications, or data synchronization tasks. This pattern is particularly valuable in modern distributed systems where multiple applications or services need to react to data changes in real-time. The primary mechanism of implementing event-driven programming in DocumentDB is by change streams.

> **Note**
>
> This guide assumes you have enabled change streams on a collection that you are working with. See Using change streams with Amazon DocumentDB to learn how to enable change streams on the collection.

Working with change streams from the Java application

The `watch()` method in MongoDB's Java driver is the primary mechanism for monitoring real-time data changes in Amazon DocumentDB. The `watch()` method can be called on by `MongoClient`, `MongoDatabase`, and `MongoCollection` objects.

The `watch()` method returns an instance of `ChangeStreamIterable` that supports various configuration options, including full document lookup for updates, providing resume tokens and timestamp for reliability, and pipeline aggregation stages for filtering changes.

`ChangeStreamIterable` implements the core Java interface `Iterable` and can be used with `forEach()`. To capture events using `forEach()`, pass in a callback function to `forEach()` that processes the changed event. The following code snippet shows how to open a change streams on a collection to start change event monitoring:

```
ChangeStreamIterable < Document > iterator = collection.watch();
iterator.forEach(event - > {
    System.out.println("Received a change: " + event);
});
```

Another way of traversing through all the change events is by opening a cursor that maintains a connection to the cluster and continuously receives new change events as they occur. To obtain a change streams cursor, use the `cursor()` method of `ChangeStreamIterable` object. The following code example shows how to monitor change events using cursor:

```
try (MongoChangeStreamCursor < ChangeStreamDocument < Document >> cursor =
 collection.watch().cursor()) {
    System.out.println(cursor.tryNext());
}
```

As a best practice, either create the `MongoChangeStreamCursor` in a try-with-resource statement or manually close the cursor. Calling the `cursor()` method on `ChangeStreamIterable` returns a `MongoChangeStreamCursor` that is created over a `ChangeStreamDocument` object.

The `ChangeStreamDocument` class is a crucial component that represents individual change events in the stream. It contains detailed information about each modification, including the operation type (insert, update, delete, replace), the document key, namespace information, and the full document content when available. The class provides methods to access various aspects of the change event, such as `getOperationType()` to determine the type of change, `getFullDocument()` to access the complete document state, and `getDocumentKey()` to identify the modified document.

The `ChangeStreamDocument` object provides two important pieces of information, a resume token and time of the change event.

Resume tokens and time-based operations in DocumentDB change streams provide crucial mechanisms for maintaining continuity and managing historical data access. A resume token is a unique identifier generated for each change event, serving as a bookmark that allows applications to restart change stream processing from a specific point after disconnections or failures. When a change stream cursor is created, it can use a previously stored resume token through the `resumeAfter()` option, enabling the stream to continue from where it left off rather than starting from the beginning or losing events.

Time-based operations in change streams offer different approaches to manage the starting point of change event monitoring. The `startAtOperationTime()` option allows you to begin watching changes that occurred at or after a specific timestamp. These time-based features are particularly valuable in scenarios requiring historical data processing, point-in-time recovery, or synchronization between systems.

The following code example retrieves the event associated with the insert document, captures it's resume token, and then provides that token to start monitoring for events after the insert event. The event is associated with the update event, then gets the cluster time when the update happened and uses that timestamp as a starting point for further processing.

```
BsonDocument resumeToken;
BsonTimestamp resumeTime;

try (MongoChangeStreamCursor < ChangeStreamDocument < Document >> cursor =
 collection.watch().cursor()) {
    System.out.println("****************** Insert Document ******************");
    ChangeStreamDocument < Document > insertChange = cursor.tryNext();
    resumeToken = insertChange.getResumeToken();
    printJson(cursor.tryNext());
}
try (MongoChangeStreamCursor < ChangeStreamDocument < Document >> cursor =
 collection.watch()
    .resumeAfter(resumeToken)
    .cursor()) {
    System.out.println("****************** Update Document ******************");
    ChangeStreamDocument < Document > insertChange = cursor.tryNext();
    resumeTime = insertChange.getClusterTime();
    printJson(cursor.tryNext());
}
try (MongoChangeStreamCursor < ChangeStreamDocument < Document >> cursor =
 collection.watch()
    .startAtOperationTime(resumeTime)
    .cursor()) {
    System.out.println("****************** Delete Document ******************");
    printJson(cursor.tryNext());
  }
```

By default, the update change event does not include the full document and it only include the changes that were made. If you need to access the complete document that was updated, you can call the `fullDocument()` method on the `ChangeStreamIterable` object. Keep in mind that when you ask for a full document to be returned for an update event, it returns the document that exists at the time the call to change streams is made.

This method takes a `FullDocument` enum as a parameter. Currently, Amazon DocumentDB only support DEFAULT and UPDATE_LOOKUP values. The following code snippet shows how to ask for full document for update events when starting to watch for changes:

```
try (MongoChangeStreamCursor < ChangeStreamDocument < Document >> cursor =
 collection.watch().fullDocument(FullDocument.UPDATE_LOOKUP).cursor())
```

Using JSON schema validation

Using the `$jsonSchema` evaluation query operator, you can validate documents being inserted into your collections.

Topics

- Creating and using JSON schema validation
- Supported keywords
- bypassDocumentValidation
- Limitations

Creating and using JSON schema validation

Creating a collection with schema validation

You can create a collection with `createCollection` operation and validation rules. These validation rules are applied during inserts or updates of Amazon DocumentDB documents. The following code example shows validation rules for a collection of employees:

```
db.createCollection("employees", {
   "validator": {
      "$jsonSchema": {
         "bsonType": "object",
         "title": "employee validation",
         "required": [ "name", "employeeId"],
         "properties": {
            "name": {
                  "bsonType": "object",
                  "properties": {
                     "firstName": {
                        "bsonType": ["string"]
                     },
                     "lastName": {
                        "bsonType": ["string"]
                     }
```

```
                },
                "additionalProperties" : false
            },
            "employeeId": {
               "bsonType": "string",
               "description": "Unique Identifier for employee"
            },
             "salary": {
               "bsonType": "double"
            },
            "age": {
               "bsonType": "number"
            }
         },
         "additionalProperties" : true
      }
   },
   "validationLevel": "strict", "validationAction": "error"
} )
```

Inserting a valid document

The following example inserts documents that comply with the above schema validation rules:

```
db.employees.insert({"name" : { "firstName" : "Carol" , "lastName" : "Smith"},
 "employeeId": "c720a" , "salary": 1000.0 })
db.employees.insert({ "name" : { "firstName" : "William", "lastName" : "Taylor" },
 "employeeId" : "c721a", "age" : 24})
```

Inserting an invalid document

The following example inserts documents that do not comply with the above schema validation rules. In this example, the employeeId value is not a string:

```
db.employees.insert({
    "name" : { "firstName" : "Carol" , "lastName" : "Smith"},
    "employeeId": 720 ,
    "salary": 1000.0
})
```

This example shows incorrect syntax within the document.

Modifying a collection

The `collMod` command is used to add or modify validation rules of existing collection. The following example adds a salary field to the required field list:

```
db.runCommand({"collMod" : "employees",
   "validator": {
      "$jsonSchema": {
         "bsonType": "object",
         "title": "employee validation",
         "required": [ "name", "employeeId", "salary"],
         "properties": {
            "name": {
                  "bsonType": "object",
                  "properties": {
                     "firstName": {
                        "bsonType": ["string"]
                     },
                     "lastName": {
                        "bsonType": ["string"]
                     }
                  },
                  "additionalProperties" : false
            },
            "employeeId": {
               "bsonType": "string",
               "description": "Unique Identifier for employee"
            },
             "salary": {
               "bsonType": "double"
            },
            "age": {
               "bsonType": "number"
            }
         },
         "additionalProperties" : true
      }
   }
} )
```

Addressing documents added before the validation rules were changed

To address documents that were added to you collection before the validation rules were changed, use the following `validationLevel` modifiers:

- **strict**: Applies validation rules on all inserts and updates.
- **moderate**: Applies validation rules to existing valid documents. During updates, existing invalid documents are not checked.

In the following example, after updating the validation rules on the collection named “employees”, the salary field is required. Updating following document will fail:

```
db.runCommand({
    update: "employees",
    updates: [{
        q: { "employeeId": "c721a" },
        u: { age: 25 , salary : 1000},
        upsert: true }]
})
```

Amazon DocumentDB returns the following output:

```
{
"n" : 0,
    "nModified" : 0,
    "writeErrors" : [
        {
"index" : 0,
            "code" : 121,
            "errmsg" : "Document failed validation"
        }
    ],
    "ok" : 1,
    "operationTime" : Timestamp(1234567890, 1)
}
```

Updating the validation level to `moderate` will allow the above document to be updated successfully:

```
db.runCommand({
    "collMod" : "employees",
```

```
    validationLevel : "moderate"
})

db.runCommand({
    update: "employees",
    updates: [{
        q: { "employeeId": "c721a" },
        u: { age: 25 , salary : 1000},
        upsert: true }]
})
```

Amazon DocumentDB returns the following output:

```
{
"n" : 1,
    "nModified" : 1,
    "ok" : 1,
    "operationTime" : Timestamp(1234567890, 1)
}
```

Retrieving documents with the $jsonSchema

The `$jsonSchema` operator can be used as a filter to query documents that match the JSON schema. This is a top-level operator which can be present in filter documents as a top level field or used with query operators such as `$and`, `$or`, and `$nor`. The following examples show the use of $jsonSchema as an individual filter and with other filter operators:

Document inserted into an "employee" collection:

```
{ "name" : { "firstName" : "Carol", "lastName" : "Smith" }, "employeeId" : "c720a",
 "salary" : 1000 }
{ "name" : { "firstName" : "Emily", "lastName" : "Brown" }, "employeeId" : "c720b",
 "age" : 25, "salary" : 1050.2 }
{ "name" : { "firstName" : "William", "lastName" : "Taylor" }, "employeeId" : "c721a",
 "age" : 24, "salary" : 1400.5 }
{ "name" : { "firstName" : "Jane", "lastName" : "Doe" }, "employeeId" : "c721a",
 "salary" : 1300 }
```

Collection filtered with the `$jsonSchema` operator only:

```
db.employees.find({
       $jsonSchema: { required: ["age"] } })
```

Amazon DocumentDB returns the following output:

```
{ "_id" : ObjectId("64e5f91c6218c620cf0e8f8b"), "name" : { "firstName" : "Emily",
 "lastName" : "Brown" }, "employeeId" : "c720b", "age" : 25, "salary" : 1050.2 }
{ "_id" : ObjectId("64e5f94e6218c620cf0e8f8c"), "name" : { "firstName" : "William",
 "lastName" : "Taylor" }, "employeeId" : "c721a", "age" : 24, "salary" : 1400.5 }
```

Collection filtered with the `$jsonSchema` operator and another operator:

```
db.employees.find({
       $or: [{ $jsonSchema: { required: ["age", "name"]}},
            { salary: { $lte:1000}}]});
```

Amazon DocumentDB returns the following output:

```
{ "_id" : ObjectId("64e5f8886218c620cf0e8f8a"), "name" : { "firstName" : "Carol",
 "lastName" : "Smith" }, "employeeId" : "c720a", "salary" : 1000 }
{ "_id" : ObjectId("64e5f91c6218c620cf0e8f8b"), "name" : { "firstName" : "Emily",
 "lastName" : "Brown" }, "employeeId" : "c720b", "age" : 25, "salary" : 1050.2 }
{ "_id" : ObjectId("64e5f94e6218c620cf0e8f8c"), "name" : { "firstName" : "William",
 "lastName" : "Taylor" }, "employeeId" : "c721a", "age" : 24, "salary" : 1400.5 }
```

Collection filtered with the `$jsonSchema` operator and with `$match` in the aggregate filter:

```
db.employees.aggregate(
    [{ $match: {
        $jsonSchema: {
            required: ["name", "employeeId"],
            properties: {"salary" :{"bsonType": "double"}}
        }
       }
    }]
)
```

Amazon DocumentDB returns the following output:

```
{
"_id" : ObjectId("64e5f8886218c620cf0e8f8a"),
 "name" : { "firstName" : "Carol", "lastName" : "Smith" },
"employeeId" : "c720a",
"salary" : 1000
}
```

```
{
"_id" : ObjectId("64e5f91c6218c620cf0e8f8b"),
"name" : { "firstName" : "Emily", "lastName" : "Brown" },
"employeeId" : "c720b",
"age" : 25,
"salary" : 1050.2
}
{
"_id" : ObjectId("64e5f94e6218c620cf0e8f8c"),
"name" : { "firstName" : "William", "lastName" : "Taylor" },
"employeeId" : "c721a",
"age" : 24,
"salary" : 1400.5
}
{
"_id" : ObjectId("64e5f9786218c620cf0e8f8d"),
"name" : { "firstName" : "Jane", "lastName" : "Doe" },
"employeeId" : "c721a",
"salary" : 1300
}
```

Viewing existing validation rules

To see the existing validation rules on a collection, use:

```
db.runCommand({
    listCollections: 1,
    filter: { name: 'employees' }
})
```

Amazon DocumentDB returns the following output:

```
{
    "waitedMS" : NumberLong(0),
    "cursor" : {
        "firstBatch" : [
            {
                "name" : "employees",
                "type" : "collection",
                "options" : {
                    "autoIndexId" : true,
                    "capped" : false,
                    "validator" : {
```

```
                    "$jsonSchema" : {
                        "bsonType" : "object",
                        "title" : "employee validation",
                        "required" : [
                            "name",
                            "employeeId",
                            "salary"
                        ],
                        "properties" : {
                            "name" : {
                                "bsonType" : "object",
                                "properties" : {
                                    "firstName" : {
                                        "bsonType" : [
                                            "string"
                                        ]
                                    },
                                    "lastName" : {
                                        "bsonType" : [
                                            "string"
                                        ]
                                    }
                                },
                                "additionalProperties" : false
                            },
                            "employeeId" : {
                                "bsonType" : "string",
                                "description" : "Unique Identifier for employee"
                            },
                            "salary" : {
                                "bsonType" : "double"
                            },
                            "age" : {
                                "bsonType" : "number"
                            }
                        },
                        "additionalProperties" : true
                    }
                },
                "validationLevel" : "moderate",
                "validationAction" : "error"
            },
            "info" : {
                "readOnly" : false
```

```
                },
                "idIndex" : {
                    "v" : 2,
                    "key" : {
                        "_id" : 1
                    },
                    "name" : "_id_",
                    "ns" : "test.employees"
                }
            }
        ],
        "id" : NumberLong(0),
        "ns" : "test.$cmd.listCollections"
    },
    "ok" : 1,
    "operationTime" : Timestamp(1692788937, 1)
}
```

Amazon DocumentDB also retains validation rules in the $out aggregation stage.

Supported keywords

The following fields are supported in the `create` and `collMod` commands:

- **`Validator`** — Supports the `$jsonSchema` operator.
- **`ValidationLevel`** — Supports `off`, `strict`, and `moderate` values.
- **`ValidationAction`** — Supports the `error` value.

The $jsonSchema operator supports the following keywords:

- `additionalItems`
- `additionalProperties`
- `allOf`
- `anyOf`
- `bsonType`
- `dependencies`
- `description`
- `enum`

- exclusiveMaximum
- exclusiveMinimum
- items
- maximum
- minimum
- maxItems
- minItems
- maxLength
- minLength
- maxProperties
- minProperties
- multipleOf
- not
- oneOf
- pattern
- patternProperties
- properties
- required
- title
- type
- uniqueItems

bypassDocumentValidation

Amazon DocumentDB supports bypassDocumentValidation for the following commands and methods:

- insert
- update
- findAndModify
- $out stage in aggregate command and in db.collection.aggregate() method

Amazon DocumentDB does not support the following commands for `bypassDocumentValidation`:

- `$merge` in `aggregate` command and in `db.collection.aggregate()` method
- `mapReduce` command and `db.collection.mapReduce()` method
- `applyOps` command

Limitations

The following limitations apply to `$jsonSchema` validation:

- Amazon DocumentDB returns the error "Document failed validation" when an operation fails the validation rule.
- Amazon DocumentDB elastic clusters do not support `$jsonSchema`.

Querying in Amazon DocumentDB

This section explains all aspects of querying with Amazon DocumentDB.

Topics

- Querying documents
- Query plan
- Explain results
- Query planner v2
- Query planner v3
- Querying geospatial data with Amazon DocumentDB
- Partial index
- Performing text search with Amazon DocumentDB

Querying documents

At times, you might need to look up your online store's inventory so that customers can see and purchase what you're selling. Querying a collection is relatively easy, whether you want all documents in the collection or only those documents that satisfy a particular criterion.

To query for documents, use the `find()` operation. The `find()` command has a single document parameter that defines the criteria to use in choosing the documents to return. The output from `find()` is a document formatted as a single line of text with no line breaks. To format the output document for easier reading, use `find().pretty()`. All the examples in this topic use `.pretty()` to format the output.

The following code samples use the four documents you inserted into the `example` collection in the preceding two exercises — `insertOne()` and `insertMany()` that are located in Adding documents.

Topics

- Retrieving all documents in a collection
- Retrieving documents that match a field Value
- Retrieving documents that match an embedded document
- Retrieving documents that match a field value in an embedded document
- Retrieving documents that match an array
- Retrieving documents that match a value in an array
- Retrieving documents using operators

Retrieving all documents in a collection

To retrieve all the documents in your collection, use the `find()` operation with an empty query document.

The following query returns all documents in the `example` collection.

```
db.example.find( {} ).pretty()
```

Retrieving documents that match a field Value

To retrieve all documents that match a field and value, use the `find()` operation with a query document that identifies the fields and values to match.

Using the preceding documents, this query returns all documents where the "Item" field equals "Pen".

```
db.example.find( { "Item": "Pen" } ).pretty()
```

Retrieving documents that match an embedded document

To find all the documents that match an embedded document, use the `find()` operation with a query document that specifies the embedded document name and all the fields and values for that embedded document.

When matching an embedded document, the document's embedded document must have the same name as in the query. In addition, the fields and values in the embedded document must match the query.

The following query returns only the "Poster Paint" document. This is because the "Pen" has different values for "`OnHand`" and "`MinOnHand`", and "Spray Paint" has one more field (`OrderQnty`) than the query document.

```
db.example.find({"Inventory": {
    "OnHand": 47,
    "MinOnHand": 50 } } ).pretty()
```

Retrieving documents that match a field value in an embedded document

To find all the documents that match an embedded document, use the `find()` operation with a query document that specifies the embedded document name and all the fields and values for that embedded document.

Given the preceding documents, the following query uses "dot notation" to specify the embedded document and fields of interest. Any document that matches these are returned, regardless of what other fields might be present in the embedded document. The query returns "Poster Paint" and "Spray Paint" because they both match the specified fields and values.

```
db.example.find({"Inventory.OnHand": 47, "Inventory.MinOnHand": 50 }).pretty()
```

Retrieving documents that match an array

To find all documents that match an array, use the `find()` operation with the array name that you are interested in and all the values in that array. The query returns all documents that have

an array with that name in which the array values are identical to and in the same order as in the query.

The following query returns only the "Pen" because the "Poster Paint" has an additional color (White), and "Spray Paint" has the colors in a different order.

```
db.example.find( { "Colors": ["Red","Green","Blue","Black"] } ).pretty()
```

Retrieving documents that match a value in an array

To find all the documents that have a particular array value, use the `find()` operation with the array name and the value that you're interested in.

```
db.example.find( { "Colors": "Red" } ).pretty()
```

The preceding operation returns all three documents because each of them has an array named `Colors` and the value "`Red`" somewhere in the array. If you specify the value "`White`," the query would only return "Poster Paint."

Retrieving documents using operators

The following query returns all documents where the "`Inventory.OnHand`" value is less than 50.

```
db.example.find(
        { "Inventory.OnHand": { $lt: 50 } } )
```

For a listing of supported query operators, see Query and projection operators.

Query plan

How can I see the `executionStats` for a query plan?

When determining why a query is executing slower than expected, it can be useful to understand what the `executionStats` are for the query plan. The `executionStats` provide the number of documents returned from a particular stage (`nReturned`), the amount of execution time spent at each stage (`executionTimeMillisEstimate`), and the amount of time it takes to generate a query plan (`planningTimeMillis`). You can determine the most time-intensive stages of your query to help focus your optimization efforts from the output of `executionStats`, as shown

in the following query examples. The `executionStats` parameter does not currently support `update` and `delete` commands.

> **Note**
>
> Amazon DocumentDB emulates the MongoDB 3.6 API on a purpose-built database engine that utilizes a distributed, fault-tolerant, self-healing storage system. As a result, query plans and the output of `explain()` may differ between Amazon DocumentDB and MongoDB. Customers who want control over their query plan can use the `$hint` operator to enforce selection of a preferred index.

Run the query that you want to improve under the `explain()` command as follows.

```
db.runCommand({explain: {query document}}).
explain("executionStats").executionStats;
```

The following is an example operation.

```
db.fish.find({}).limit(2).explain("executionStats");
```

Output from this operation looks something like the following.

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "test.fish",
        "winningPlan" : {
            "stage" : "SUBSCAN",
            "inputStage" : {
                "stage" : "LIMIT_SKIP",
                "inputStage" : {
                    "stage" : "COLLSCAN"
                }
            }
        }
    },
    "executionStats" : {
        "executionSuccess" : true,
        "executionTimeMillis" : "0.063",
        "planningTimeMillis" : "0.040",
```

```
        "executionStages" : {
            "stage" : "SUBSCAN",
            "nReturned" : "2",
            "executionTimeMillisEstimate" : "0.012",
            "inputStage" : {
                "stage" : "LIMIT_SKIP",
                "nReturned" : "2",
                "executionTimeMillisEstimate" : "0.005",
                "inputStage" : {
                    "stage" : "COLLSCAN",
                    "nReturned" : "2",
                    "executionTimeMillisEstimate" : "0.005"
                }
            }
        }
    },
    "serverInfo" : {
        "host" : "enginedemo",
        "port" : 27017,
        "version" : "3.6.0"
    },
    "ok" : 1
}
```

If you are interested in seeing only the `executionStats` from the query above, you can use the following command. For small collections, the Amazon DocumentDB query processor can choose to not use an index if the performance gains are negligible.

```
db.fish.find({}).limit(2).explain("executionStats").executionStats;
```

Query plan cache

In order to optimize performance and reduce planning duration, Amazon DocumentDB internally caches query plans. This enables queries with the same shape to be executed directly using a cached plan.

However, this caching may sometimes cause a random delay for the same query; for example, a query that typically takes one second to run may occasionally take ten seconds. This is because over time, the reader instance cached various shapes of the query, thus consuming memory. If you experience this random slowness, there is no action needed you need to do to release the

memory--the system will manage the memory usage for you and once the memory reaches certain threshold, it will be automatically released.

Explain results

If you want to return information on query plans, Amazon DocumentDB supports verbosity mode `queryPlanner`. The `explain` results return the selected query plan chosen by the optimizer in a format similar to the following:

```
{
   "queryPlanner" : {
      "plannerVersion" : <int>,
      "namespace" : <string>,
      "winningPlan" : {
         "stage" : <STAGE1>,
         ...
         "inputStage" : {
            "stage" : <STAGE2>,
            ...
            "inputStage" : {
               ...
            }
         }
      }
   }
}
```

The following sections will define common `explain` results.

Topics

- Scan and filter stage
- Index intersection
- Index union
- Multiple index intersection/union
- Compound index
- Sort stage
- Group stage

Scan and filter stage

The optimizer may choose one of the following scans:

COLLSCAN

This stage is a sequential collection scan.

```
{
    "stage" : "COLLSCAN"
}
```

IXSCAN

This stage scans the index keys. The optimizer may retrieve the document within this stage and this may result in a FETCH stage appended later.

```
db.foo.find({"a": 1})
{
    "stage" : "IXSCAN",
    "direction" : "forward",
    "indexName" : <idx_name>
}
```

FETCH

If the optimizer retrieved documents in a stage other than IXSCAN, the result will include a FETCH stage. For example, the IXSCAN query above may result a combination of FETCH and IXSCAN stages:

```
db.foo.find({"a": 1})
{
    "stage" : "FETCH",
    "inputStage" : {
        "stage" : "IXSCAN",
        "indexName" : <idx_name>
    }
}
```

IXONLYSCAN scans only the index key. Create compound indexes won't avoid FETCH.

Index intersection

IXAND

Amazon DocumentDB may include an IXAND stage with an inputStages array of IXSCAN if it can utilize index intersection. For example, we may see output like:

```
{
    "stage" : "FETCH",
    "inputStage" : {
        "stage" : "IXAND",
        "inputStages" : [
            {
                "stage" : "IXSCAN",
                "indexName" : "a_1"
            },
            {
                "stage" : "IXSCAN",
                "indexName" : "b_1"
            }
        ]
    }
}
```

Index union

IXOR

Similar to index intersection, Amazon DocumentDB may include `IXOR` stage with an `inputStages` array for the `$or` operator.

```
db.foo.find({"$or": [{"a": {"$gt": 2}}, {"b": {"$lt": 2}}]})
```

For the above query, the explain output may look like this:

```
{
    "stage" : "FETCH",
    "inputStage" : {
```

```
        "stage" : "IXOR",
        "inputStages" : [
            {
                "stage" : "IXSCAN",
                "indexName" : "a_1"
            },
            {
                "stage" : "IXSCAN",
                "indexName" : "b_1"
            }
        ]
    }
}
```

Multiple index intersection/union

Amazon DocumentDB can combine multiple index intersection or union stages together then fetch the result. For example:

```
{
    "stage" : "FETCH",
    "inputStage" : {
        "stage" : "IXOR",
        "inputStages" : [
            {
                "stage" : "IXSCAN",
                ...
            },
            {
                "stage" : "IXAND",
                "inputStages" : [
                    {
                        "stage" : "IXSCAN",
                        ...
                    },
                    {
                        "stage" : "IXSCAN",
                        ...
                    }
                ]
            }
        ]
```

```
    }
}
```

The usage of index intersection or union stages are not impacted by the index type (sparse, compound, etc).

Compound index

Amazon DocumentDB compound index usage is not limited in the beginning subsets of indexed fields; it can use index with the suffix part but it may not be very efficient.

For example, the compound index of `{ a: 1, b: -1 }` can support all three queries below:

```
db.orders.find( { a: 1 } )
```

```
db.orders.find( { b: 1 } )
```

```
db.orders.find( { a: 1, b: 1 } )
```

Sort stage

If there is an index on the requested sort key(s), Amazon DocumentDB can use the index to obtain the order. In that case, the result will not include a `SORT` stage, but rather an `IXSCAN` stage. If the optimizer favors a plain sort, it will include a stage like this:

```
{
    "stage" : "SORT",
    "sortPattern" : {
        "a" : 1,
        "b" : -1
    }
}
```

Group stage

Amazon DocumentDB supports two different group strategies:

- `SORT_AGGREGATE`: On disk sort aggregate.
- `HASH_AGGREGATE`: In memory hash aggregate.

Query planner v2

The new query planner for Amazon DocumentDB (planner version 2.0) features advanced query optimization capabilities and improved performance. Planner version 2.0 for Amazon DocumentDB 5.0 delivers up to 10x performance improvement over the prior version when using `find` and `update` operators with indexes. Performance improvements primarily come from using more optimal index plans and enabling index scan support for operators such as negation operators (`$neq`, `$nin`) and nested `$elementMatch`. Planner version 2.0 queries run faster through better cost estimation techniques, optimized algorithms, and enhanced stability. Planner version 2.0 supports plan cache filter APIs as well, which enhances planner stability. With this feature, Amazon DocumentDB 5.0 now offers the ability to select from the different versions of query planner.

Topics

- Prerequisites
- Selecting planner version 2.0 as the default query planner
- Best practices
- Limitations
- Improvements to Find and Update Operators
- Plan cache filters
- Potential behavior differences between planner version 1.0, 2.0, and MongoDB
- Planner version 2.0 bridges behavior gap with MongoDB

Prerequisites

The following prerequisites apply to planner version 2.0:

- Planner version 2.0 is available in all AWS Regions where engine version 5.0 is available.
- To opt for using version 2.0 as the default query planner, your cluster needs to be on engine patch version 3.0.15902 or later of Amazon DocumentDB version 5.0. For steps to update to the latest engine version patch, see Performing a patch update to a cluster's engine version.
- To set planner version 2.0 as the default query planner, you need IAM permissions to update cluster parameter groups.
- For Amazon DocumentDB 8.0, planner version 3.0 is the default query planner. See Query planner v3 for details.

Selecting planner version 2.0 as the default query planner

Use the following steps to select 2.0 as the default query planner from the console or CLI:

- Follow the steps in Modifying Amazon DocumentDB cluster parameters to modify your cluster's parameter group.
- For the parameter titled 'plannerVersion', change the value to 2.0 indicating planner version 2.0.
- Select **Apply immediately** (selecting **Apply at reboot** will render the selection ineffective until next reboot of the cluster).

Best practices

For expected results, use the following best practices when applying planner version 2.0:

- In a global cluster, select the same `plannerVersion` value (1.0 or 2.0) in the cluster parameter groups for both Regions. Note that selecting different planner versions in primary and secondary Regions may cause inconsistent query behavior and performance.
- Updating to planner version 2.0 during a scheduled maintenance windows or during reduced traffic periods will be the least disruptive, as there may be increased error rates if the planner version is changed when workloads are actively running.
- Planner version 2.0 works most optimally with MongoDB shell version 5.0.

Limitations

The following limitations apply to planner version 2.0:

- Planner version 2.0 is not supported in elastic clusters, which will fall back to planner version 1.0.
- Planner version 2.0 is not supported for aggregation and distinct commands, which will fall back to planner version 1.0.
- Some query shapes cannot have a plan cache filter set on them. For the current list, see Supported query shapes.

Improvements to `Find` and `Update` Operators

Planner version 2.0 optimizes fundamental operations including `find`, `update`, `delete`, and `find-and-modify` commands. The following tabbed sections show enhanced capabilities for indexes, as well as the query performance improvements with planner version 2.0:

Enhanced index support

- Planner version 2.0 adds index support for negation operators including `$nin`, `$ne`, `$not {eq}`, and `$not {in}`, as well as `$type` and `$elemMatch`.

```
Sample Document: { "x": 10, "y": [1, 2, 3] }

db.foo.createIndex({ "x": 1, "y": 1 })
db.foo.find({ "x": {$nin: [20, 30] }})
db.foo.find({"x":{ $type: "string" }})

db.foo.createIndex({"x.y": 1})
db.foo.find({"x":{$elemMatch:{"y":{$elemMatch:{"$gt": 3 }}}}})
```

- Planner version 2.0 utilizes sparse or partial indexes even when `$exists` is not present in the query expression.

```
Sample Document: {"name": "Bob", "email": "example@fake.com" }

Using Planner Version 1.0, you can specify the command as shown below:
db.foo.find({email: "example@fake.com", email: {$exists: true}})

Using Planner Version 2.0, you can specify command without $exists:
db.foo.find({ email: "example@fake.com" })
```

- Planner version 2.0 will utilize partial indexes even when the query condition doesn't exactly match the partial index filter expression.

```
Sample Document: {"name": "Bob", "age": 34}
db.foo.createIndex({"age":1},{partialFilterExpression:{"age":{$lt:50}}})

With Planner Version 1.0, index is used only when the query condition meets the
 partial
 index filter criterion:
     db.foo.find({"age":{$lt:50}})
```

```
With Planner Version 2.0, index is used even when the query condition doesn't meet
 the index
criterion:
db.foo.find({"age":{$lt:30}})
```

- Planner version 2.0 utilizes partial index scan with $elemMatch queries.

```
Sample Document: {"name": "Bob", "age": [34,35,36]}
db.foo.createIndex({"age":1},{partialFilterExpression:{"age":{$lt:50,$gt:20}}})
db.foo.find({age:{$elemMatch:{$lt:50,$gt:20}}})
```

- Planner version 2.0 includes index scan support for `$regex`, without the need of providing `$hint` in your application code. `$regex` supports index on prefix searches only.

```
Sample Document: { "x": [1, 2, 3], "y": "apple" }
db.foo.createIndex({ "x": 1, "y": 1 })
db.foo.find({"y":{ $regex: "^a" }})
```

- Planner version 2.0 improves the performance of queries involving multi-key indexes, with equality conditions on multi-key field.

```
Sample Document: {"x": [1, 2, 3],
"y": 5}
db.foo.createIndex({"x": 1, "y":1})
db.foo.find({"x": 2,
"y": {$gt: 1}}).limit(1)
```

- Planner version 2.0 improves the performance of queries that involve multiple filters, especially on collections with documents greater than 8 KB.

```
Sample Document: {"x": 2,
"y": 4,
"z": 9,
"t": 99}
db.foo.find({$and: [{"x": {$gt : 1}, "y": {$gt : 3}, "z": {$lt : 10},
"t":{$lt : 100}}]})
```

- Planner version 2.0 improves the performance of queries when using the `$in` operator with a compound index by eliminating the sort stage.

```
Sample Document: {"x": 2,
"y": 4,
```

```
"z": 9,
"t": 99}
db.foo.createIndex({"x":1, "y":1})
db.foo.find({"x":2,
"y":$in:[1,2,3,4]}).sort({x:1,y:1})
```

It also improves the performance of queries that use multi-key indexes with $in elements.

```
Sample Document: {"x": [1, 2, 3]}
db.foo.createIndex({"x": 1})
db.foo.find("x":{$in:[>100 elements]})
```

Query performance improvements

- Planner version 2.0 improves the performance of queries that involve multiple filters, especially on collections with documents greater than 8 KB.

```
Sample Document: {"x": 2,
"y": 4,
"z": 9,
"t": 99}
db.foo.find({$and: [{"x": {$gt : 1}, "y": {$gt : 3}, "z": {$lt : 10},
"t":{$lt : 100}}]})
```

- Planner version 2.0 improves the performance of queries when using the $in operator with a compound index by eliminating the sort stage.

```
Sample Document: {"x": 2,
"y": 4,
"z": 9,
"t": 99}
db.foo.createIndex({"x":1, "y":1})
db.foo.find({"x":2,
"y":$in:[1,2,3,4]}).sort({x:1,y:1})
```

It also improves the performance of queries that use multikey indexes with $in elements.

```
Sample Document: {"x": [1, 2, 3]}
db.foo.createIndex({"x": 1})
db.foo.find("x":{$in:[>100 elements]})
```

Plan cache filters

Planner version 2.0 adds support for plan cache filters, also called index filters, which let you restrict the set of indexes that the planner considers for a specific query shape. Filters are set with a database command and applied on the server, so if you experience a query regression you can mitigate it without modifying your application code.

On planner version 2.0, plan cache filters apply to the `find` and `count` commands, and starting with Amazon DocumentDB 5.0.2 and 8.0.2 also to the `update`, `delete`, and `findAndModify` commands. The `distinct` and `aggregate` commands require planner version 3.0.

For more information about the command reference, supported query shapes, worked examples, and how filters interact with hints, see Plan cache filters.

Potential behavior differences between planner version 1.0, 2.0, and MongoDB

In some edge cases, it is possible that planner version 2.0 may produce results that slightly vary from results in MongoDB. This section walks through some examples of these possibilities.

$(update) and $(projection)

- In some cases, `$(update)` and `$(projection)` operators in MongoDB may behave differently from Amazon DocumentDB's planner version 1.0. Below are some examples:

```
db.students_list.insertMany( [ { _id: 5, student_ids: [ 100, 200 ], grades: [ 95,
 100 ], grad_year: [ 2024, 2023 ] } ] )
```

```
db.students_list.updateOne({ student_ids: 100, grades: 100, grad_year: 2024 },
{ $set: { “grad_year.$”: 2025 } }
```

  - **Planner version 1.0** — Updates field 2022
  - **MongoDB** — Updates field 2022
  - **Planner version 2.0** — Updates field 2021

- ```
  db.col.insert({x:[1,2,3]})
  db.col.update({$and:[{x:1},{x:3}]},{$set:{"x.$":500}})
  ```

  - **Planner version 1.0** — Randomly updates the first element matched
  - **MongoDB** — Randomly updates the first element matched
  ```

- **Planner version 2.0** — Does not make updates

- ```
  db.col.insert({x:[1,2,3]})
  db.col.find()
  ```

  - **Planner version 1.0** — Randomly selects the matched element
  - **MongoDB** — Randomly selects the matched element
  - **Planner version 2.0** — Does not make a selection

- ```
  db.col.insert({x:100})
  db.col.update({x:100},{x:100})
  ```

  - **Planner version 1.0** — nModified count changes
  - **MongoDB** — nModified count changes
  - **Planner version 2.0** — nModified count does not change when updated with the same value.

- When `$(update)` operator is used with `$setOnInsert`, planner version 1.0 and MongoDB throw an error, but planner version 2.0 does not.
- Renaming a non-existing field to `$field` throws an error in planner version 2.0, whereas does not produce updates in planner version 1.0 and MongoDB.

Index behavior

- Planner version 2.0 throws an error when `$hint` is applied with an unsuitable index, whereas planner version 1.0 and MongoDB do not.

```
// Insert
db.col.insert({x:1})
db.col.insert({x:2})
db.col.insert({x:3})

// Create index on x with partialFilter Expression {x:{$gt:2}}
db.col.createIndex({x:1},{partialFilterExpression:{x:{$gt:2}}})

// Mongodb allows hint on the following queries
db.col.find({x:1}).hint("x_1")
// result is no documents returned because {x:1} is not indexed by the partial
 index
```

```
// Without $hint mongo should return {x:1}, thus the difference in result between
 COLSCAN and IXSCAN

DocumentDB will error out when $hint is applied on such cases.
db.col.find({x:1}).hint("x_1")
Error: error: {
    "ok" : 0,
    "operationTime" : Timestamp(1746473021, 1),
    "code" : 2,
    "errmsg" : "Cannot use Hint for this Query. Index is multi key index , partial
 index or sparse index and query is not optimized to use this index."
}

rs0:PRIMARY> db.runCommand({"planCacheSetFilter": "col", "query": { location:
 {$nearSphere: {$geometry: {type: "Point", coordinates: [1, 1]}}}}, "indexes":
 ["name_1"]})
{
    "ok" : 0,
    "operationTime" : Timestamp(1750815778, 1),
    "code" : 303,
    "errmsg" : "Unsupported query shape for index filter $nearSphere"
}
```

- $near cannot use $hint({“$natural”:1}) in planner version 2.0.

```
// indexes present are index on x and geo index

rs0:PRIMARY> db.usarestaurants.getIndexes()
[
    {
        "v" : 4,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "test.usarestaurants"
    },
    {
        "v" : 4,
        "key" : {
            "location" : "2dsphere"
        },
        "name" : "location_2dsphere",
        "ns" : "test.usarestaurants",
```

```
        "2dsphereIndexVersion" : 1
    }
]

// Planner Version 2.0 will throw an error when $hint is applied with index "x_1"
rs0:PRIMARY> db.usarestaurants.find({    "location":{       "$nearSphere":{
         "$geometry":{             "type":"Point",            "coordinates":
[                -122.3516,                47.6156            ]          },
          "$minDistance":1,          "$maxDistance":2000       }    } }, {
  "name":1 }).hint({"$natural": 1})
Error: error: {
    "ok" : 0,
    "operationTime" : Timestamp(1746475524, 1),
    "code" : 291,
    "errmsg" : "unable to find index for $geoNear query"
}

// Planner Version 1.0 and MongoDB will not throw an error
 db.usarestaurants.find({    "location":{       "$nearSphere":{
  "$geometry":{             "type":"Point",             "coordinates":[
            -122.3516,               47.6156             ]          },
        "$minDistance":1,          "$maxDistance":2000       }    } }, {
  "name":1 }).hint({"$natural": 1})
{ "_id" : ObjectId("681918e087dadfd99b7f0172"), "name" : "Noodle House" }
```

- While MongoDB supports complete regex index scans, planner version 2.0 supports regex index scan on prefix fields only.

```
// index on x
db.col.createIndex({x:1})

// index scan is used only for prefix regexes
rs0:PRIMARY> db.col.find({x: /^x/}).explain()
{
    "queryPlanner" : {
        "plannerVersion" : 2,
        "namespace" : "test.col",
        "winningPlan" : {
            "stage" : "IXSCAN",
            "indexName" : "x_1",
            "direction" : "forward",
            "indexCond" : {
                "$and" : [
```

```
                    {
                        "x" : {
                            "$regex" : /^x/
                        }
                    }
                ]
            },
            "filter" : {
                "x" : {
                    "$regex" : /^x/
                }
            }
        }
    },
    "indexFilterSet" : false,
    "indexFilterApplied" : false,
    "ok" : 1,
    "operationTime" : Timestamp(1746474527, 1)
}

// COLSCAN is used for non-prefix regexes
rs0:PRIMARY> db.col.find({x: /x$/}).explain()
{
    "queryPlanner" : {
        "plannerVersion" : 2,
        "namespace" : "test.col",
        "winningPlan" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "x" : {
                    "$regex" : /x$/
                }
            }
        }
    },
    "indexFilterSet" : false,
    "indexFilterApplied" : false,
    "ok" : 1,
    "operationTime" : Timestamp(1746474575, 1)
```

- There are some inherent differences in using plan cache filters with planner version 2.0 as compared to MongoDB. While planner version 2.0 does not support specifying “projection” and “collation” with plan cache filters, MongoDB does. However, MongoDB index filter is

in-memory only and is lost after restart. Planner version 2.0 persists index filters through restarts and patches.

Others

- The format of DML audit logs when using planner version 2.0 varies slightly from that of planner version 1.0.

```
command -  db.col.find({x:1})

************** Audit logs generated ******************

// v1 format for dml audit logs
{"atype":"authCheck","ts":1746473479983,"timestamp_utc":"2025-05-05
 19:31:19.983","remote_ip":"127.0.0.1:47022","users":
[{"user":"serviceadmin","db":"test"}],"param":
{"command":"find","ns":"test.col","args":{"batchSize":101,"filter":
{"x":1},"find":"col","limit":18446744073709551615,"lsid":
{"id":{"$binary":"P6RCGz9ZS4iWBSSHWXW15A==","$type":"4"},"uid":
{"$binary":"6Jo8PisnEi3dte03+pJFjdCyn/5cGQL8V2KqaoWsnk8=","$type":"0"}},"maxScan":18446744

// v2 formal for dml audit logs
{"atype":"authCheck","ts":1746473583711,"timestamp_utc":"2025-05-05
 19:33:03.711","remote_ip":"127.0.0.1:37754","users":
[{"user":"serviceadmin","db":"test"}],"param":
{"command":"find","ns":"test.col","args":{"find":"col","filter":{"x":1},"lsid":
{"id":{"$binary":"nJ88TGCSSd
+BeD2+ZtrhQg==","$type":"4"}},"$db":"test"},"result":0}}
```

- The index condition as part of the explain plan:

```
rs0:PRIMARY> db.col.createIndex({index1:1})
{
 "createdCollectionAutomatically" : false,
 "numIndexesBefore" : 1,
 "numIndexesAfter" : 2,
 "ok" : 1,
 "operationTime" : Timestamp(1761149251, 1)
}
```

Planner version 2.0 explain plan output displaying index condition and filter:

```
rs0:PRIMARY> db.col.find({$and:[{price:{$eq:300}},{item:
{$eq:"apples"}}]}).explain()
{
 "queryPlanner" : {
  "plannerVersion" : 2,
  "namespace" : "test.col",
  "winningPlan" : {
   "stage" : "IXSCAN",
   "indexName" : "price_1",
   "direction" : "forward",
   "indexCond" : {
    "$and" : [
     {
      "price" : {
       "$eq" : 300
      }
     }
    ]
   },
   "filter" : {
    "$and" : [
     {
      "item" : {
       "$eq" : "apples"
      }
     }
    ]
   }
  }
 },
 "indexFilterSet" : false,
 "indexFilterApplied" : false,
 "ok" : 1,
 "operationTime" : Timestamp(1761149497, 1)
}
```

Planner version 1.0 explain plan output:

```
rs0:PRIMARY> db.col.find({$and:[{price:{$eq:300}},{item:
{$eq:"apples"}}]}).explain()
{
 "queryPlanner" : {
```

```
  "plannerVersion" : 1,
  "namespace" : "test.col",
  "winningPlan" : {
   "stage" : "IXSCAN",
   "indexName" : "price_1",
   "direction" : "forward"
  }
 },
 "ok" : 1,
 "operationTime" : Timestamp(1761149533, 1)
}
```

Planner version 2.0 bridges behavior gap with MongoDB

There are some areas where planner version 2.0 closes the behavior gaps from MongoDB:

- Planner version 2.0 allows numeric index lookup on flattened arrays for `$elemMatch`:

```
doc: {"x" : [ [ { "y" : 1 } ] ] }

// Planner Version 2 and mongo
> db.bar.find({"x.0": {$elemMatch: {y: 1}}})
{ "_id" : ObjectId("68192947945e5846634c455a"), "x" : [ [ { "y" : 1 } ] ] }
> db.bar.find({"x": {$elemMatch: {"0.y": 1}}})
{ "_id" : ObjectId("68192947945e5846634c455a"), "x" : [ [ { "y" : 1 } ] ] }

//Whereas Planner Version 1 wouldn't return any results.
> db.bar.find({"x.0": {$elemMatch: {y: 1}}})
> db.bar.find({"x": {$elemMatch: {"0.y": 1}}})
```

- While planner version 1.0 excluded strings in projection, planner version 2.0's behavior aligns with MongoDB and treats them as literal values"

```
// Planner V2/ MongoDB
> db.col.find()
{ "_id" : ObjectId("681537738aa101903ed2fe05"), "x" : 1, "y" : 1 }
> db.col.find({},{x:"string"})
{ "_id" : ObjectId("681537738aa101903ed2fe05"), "x" : "string" }

// Planner V1 treats strings as exclude in projection
rs0:PRIMARY> db.col.find()
{ "_id" : ObjectId("68153744d42969f11d5cca72"), "x" : 1, "y" : 1 }
```

```
rs0:PRIMARY> db.col.find({},{x:"string"})
{ "_id" : ObjectId("68153744d42969f11d5cca72"), "y" : 1 }
```

- Planner version 2.0, like MongoDB, does not allow projection on same fields “x” and “x.a”:

```
// Planner version 2/MongoDB will error out
> db.col.find()
{ "_id" : ObjectId("68153da2012265816bc9ba23"), "x" : [ { "a" : 1 }, 3 ] }
 db.col.find({},{"x.a":1,"x":1}) // error

// Planner Version 1 does not error out
db.col.find()
{ "_id" : ObjectId("68153da2012265816bc9ba23"), "x" : [ { "a" : 1 }, 3 ] }

db.col.find({},{"x.a":1,"x":1})
{ "_id" : ObjectId("68153d60143af947c720d099"), "x" : [ { "a" : 1 }, 3 ] }
```

- Planner version 2.0, like MongoDB, allows projection on subdocuments:

```
// Planner Version2/MongoDB supports projections on subdocuments
 db.col.find()
{ "_id" : ObjectId("681542d8f35ace71f0a50004"), "x" : [ { "y" : 100 } ] }
> db.col.find({},{"x":{"y":1}})
{ "_id" : ObjectId("681542b7a22d548e4ac9ddea"), "x" : [ { "y" : 100 } ] }

// Planner V1 throws error if projection is subdocument
 db.col.find()
{ "_id" : ObjectId("681542d8f35ace71f0a50004"), "x" : [ { "y" : 100 } ] }
rs0:PRIMARY> db.col.find({},{"x":{"y":1}})
Error: error: {
    "ok" : 0,
    "operationTime" : Timestamp(1746223914, 1),
    "code" : 2,
    "errmsg" : "Unknown projection operator y"
}
```

- With Planner version 2.0, like MongoDB, projection does not support fields after the $ operator:

```
// Mongo and Planner Version 2 will error out
  db.col.find()
{ "_id" : ObjectId("68155fa812f843439b593f3f"), "x" : [ { "a" : 100 } ] }
 db.col.find({"x.a":100},{"x.$.a":1}) - // error
```

```
// v1 will not error out
 db.col.find()
{ "_id" : ObjectId("68155fa812f843439b593f3f"), "x" : [ { "a" : 100 } ] }
db.col.find({"x.a":100},{"x.$.a":1})
{ "_id" : ObjectId("68155dee13b051d58239cd0a"), "x" : [ { "a" : 100 } ] }
```

- Planner version 2.0, like MongoDB, allows $hint usage:

```
// v1 will error out on $hint if there are no filters
db.col.find({}).hint("x_1")
Error: error: {
    "ok" : 0,
    "operationTime" : Timestamp(1746466616, 1),
    "code" : 2,
    "errmsg" : "Cannot use Hint for this Query. Index is multi key index , partial
 index or sparse index and query is not optimized to use this index."
}


// Mongo and Planner Version 2 will allow $hint usage
 db.col.find({}).hint("x_1")
{ "_id" : ObjectId("6818f790d5ba9359d68169cf"), "x" : 1 }
```

Query planner v3

Planner Version 3 in Amazon DocumentDB 8.0 supports 21 aggregation stages, including 6 new stages. Planner V3 comes with inbuilt support for distinct commands. It offers upto 2x overall performance improvement over Planner v2 in Amazon DocumentDB 5.0. All new features and operators in Amazon DocumentDB 8.0 are compatible with Planner v3. New aggregation stages in Amazon DocumentDB 8.0 that are supported by Planner v3 include $replaceWith, $vectorSearch, $merge, $set, $unset, $bucket. Planner v3 also supports new features and operators in Amazon DocumentDB 8.0 including Collation, Views, $merge, $pow, $rand, $dateTrunc, $dateToParts, $dateFromParts.

Topics

- Prerequisites
- Selecting planner version 3.0 as the default query planner
- Best practices
- Limitations

- Improvements to aggregate and distinct Operators
- Plan cache filters
- Potential behavior differences between planner version 1.0, 3.0, and MongoDB

Prerequisites

The following prerequisites apply to planner version 3.0:

- Planner version 3.0 is available in all AWS Regions where engine version 8.0 is available.
- Planner version 3.0 is the default query planner when engine version 8.0 is selected.

Selecting planner version 3.0 as the default query planner

If you changed your default query planner in Amazon DocumentDB 8.0, and need to revert to planner v3, you can do so from the console or CLI:

- Follow the steps in Modifying Amazon DocumentDB cluster parameters to modify your cluster's parameter group.
- For the parameter titled 'plannerVersion', change the value to 3.0 indicating planner version 3.0.
- Select **Apply immediately** (selecting **Apply at reboot** will render the selection ineffective until next reboot of the cluster).

Best practices

For expected results, use the following best practices when applying planner version 3.0:

- In a global cluster, select the same `plannerVersion` value (1.0 or 2.0 or 3.0) in the cluster parameter groups for both Regions. Note that selecting different planner versions in primary and secondary Regions may cause inconsistent query behavior and performance.
- Updating to planner version 3.0 during a scheduled maintenance windows or during reduced traffic periods will be the least disruptive, as there may be increased error rates if the planner version is changed when workloads are actively running.

Limitations

The following limitations apply to planner version 3.0:

- Planner version 3.0 is not supported in elastic clusters, which will fall back to planner version 1.0.

- While Planner v1 allows the use of 'planHint' to ensure that a specific query plan is selected by the query optimizer, Planner v3 does not allow the use of 'planHint' and relies on internal optimizations to choose the best plan for the given query.

Improvements to `aggregate` and `distinct` Operators

Planner version 3.0 introduces improvements across $aggregate stages, and $distinct command. The following are some of the most noteworthy improvements..

- The planner moves $match stages earlier in the pipeline when possible, reducing the number of documents processed by subsequent stages.

```
//Planner v1
db.orders.aggregate([
  { $project: { customerId: 1, orderDate: 1, totalAmount: 1 } },
  { $match: { customerId: { $gt: 1000 } } }
])

// Planner v3 pulls up the match since customerId exists in original documents
// Optimized internally as:
db.orders.aggregate([
  { $match: { customerId: { $gt: 1000 } } },  // Pulled up before project
  { $project: { customerId: 1, orderDate: 1, totalAmount: 1 } }
])
```

- The planner automatically combines $lookup and $unwind stages when they operate on the same field, reducing intermediate data processing and improving performance.

```
                Example query:
//Planner v1
db.orders.aggregate([
  { $lookup: { from: "products", localField: "productId", foreignField: "_id", as:
 "productInfo" } },
  { $unwind: "$productInfo" },
  { $project: { orderDate: 1, "productInfo.name": 1, "productInfo.price": 1 } }
])
```

```
// Planner version 3.0 optimizes this internally by coalescing the $lookup and
 $unwind stages
```

- Planner version 3.0 introduces a new Distinct Scan execution strategy that significantly improves performance for distinct operations on low cardinality indexes.

```
                 Example query:
//// If there is a low cardinality index on "category", you may see a query plan like
 below
db.explain().products.distinct("category")

"queryPlanner" : {
        "plannerVersion" : 3,
        "namespace" : "db.products",
        "winningPlan" : {
                "stage" : "AGGREGATE",
                "inputStage" : {
                        "stage" : "DISTINCT_SCAN",
                        "inputStage" : {
                                "stage" : "IXONLYSCAN",
                                "indexName" : "category_1",
                                "direction" : "forward"
                        }
                }
        }
}
```

Plan cache filters

Planner version 3.0 supports plan cache filters, also called index filters, which let you restrict the set of indexes that the planner considers for a specific query shape. Filters are set with a database command and applied on the server, so if you experience a query regression you can mitigate it without modifying your application code.

Planner version 3.0 applies plan cache filters to more commands than planner version 2.0 does:

- The `distinct` command requires planner version 3.0.

- The `aggregate` command requires planner version 3.0 and Amazon DocumentDB 8.0.2. A filter applies when the pipeline reads the collection through a leading `$match` stage.
- A filter set on the foreign collection of a `$lookup` or `$graphLookup` stage drives the index used to scan that foreign collection. This also requires planner version 3.0 and Amazon DocumentDB 8.0.2.

For more information about the command reference, supported query shapes, worked examples, and how filters interact with hints, see Plan cache filters.

Potential behavior differences between planner version 1.0, 3.0, and MongoDB

In some edge cases, it is possible that planner version 3.0 may produce results that slightly vary from planner version 1.0. This section walks through some examples of these possibilities.

Feature Differences

- On an empty collection or when previous stages like match filters all documents, Planner v1 gives the output “field”:0. Planner v3 and MongoDB won't give any outputs.
- With Plannerv1, {"$skip":n} does not skip if there are less than n documents returned by the previous stage. Plannerv3 and MongoDB correctly skip irrespective of number of documents returned.
- When a foreign collection referenced in $lookup does not exist, plannerv1 throws an error. Planner v3 and MongoDB treat the foreign collection as empty collection and performs $lookup.

```
db.coll.aggregate([
    {$lookup: {from: "does_not_exist", localField: "a", foreignField: "a", as:
 "c"}}
])
```

- Only Planner v1 will allow using multiple $search in a pipeline Planner v2/v3 will throw an error and MongoDB does not support it.

```
VectorSearch = { "$search": { "vectorSearch": {
   "vector": [0.2, 0.5, 0.8],
   "path": "vectorEmbedding",
   "similarity": "cosine",
   "k": 2,
   "efSearch": 1
```

```
   }}}
db.coll.aggregate([VectorSearch, VectorSearch])
```

- Only Planner v3 works when vectorSearch stage is not the first stage. In this case, MongoDB will throw an error and Planner v1 does not support $vectorSearch stage.

```
VectorSearch = { {"$vectorSearch": {
    "queryVector": [0.2, 0.5, 0.8],
    "path": "vectorEmbedding",
    "similarity": "euclidean",
    "limit": 4,
    "numCandidates": 100} }

db.coll.aggregate([$match:{}, VectorSearch])
```

Querying geospatial data with Amazon DocumentDB

This section covers how you can query Geospatial data with Amazon DocumentDB. After you read this section, you will be able to answer how do store, query and index Geospatial data in Amazon DocumentDB.

Topics

- Overview
- Indexing and storing geospatial data
- Querying geospatial data
- Limitations

Overview

Common use cases for Geospatial involve proximity analysis from your data. For example, “finding all airports within 50 miles of Seattle”, or “find the closest restaurants from a given location”. Amazon DocumentDB uses the GeoJSON specification to represent geospatial data. GeoJSON is an open-source specification for the JSON-formatting of shapes in a coordinate space. GeoJSON coordinates captures both longitude and latitude, representing positions on an earth-like sphere.

Indexing and storing geospatial data

Amazon DocumentDB uses the 'Point' GeoJSON type to store geospatial data. Each GeoJSON document (or subdocument) is generally composed of two fields:

- **type** - the shape being represented, which informs Amazon DocumentDB how to interpret the "coordinates" field. At this moment, Amazon DocumentDB only supports points
- **coordinates** – a latitude and longitude pair represented as an object in an array – [longitude, latitude]

Amazon DocumentDB also uses 2dsphere indexes to index Geospatial data. Amazon DocumentDB supports indexing points. Amazon DocumentDB supports proximity querying with 2dsphere indexing.

Let's consider a scenario where you are building an application for food delivery service. You want to store restaurant latitude and longitude pairs in Amazon DocumentDB. To do so, first create an index on the geospatial field that holds the latitude and longitude pair.

```
use restaurantsdb
db.usarestaurants.createIndex({location:"2dsphere"})
```

The output of this command would look something like this:

```
{
 "createdCollectionAutomatically" : true,
 "numIndexesBefore" : 1,
 "numIndexesAfter" : 2,
 "ok" : 1
}
```

Once you have created an index, you can start inserting data into your Amazon DocumentDB collection.

```
db.usarestaurants.insert({
   "state":"Washington",
   "city":"Seattle",
   "name":"Thai Palace",
   "rating": 4.8,
   "location":{
```

```
      "type":"Point",
      "coordinates":[
         -122.3264,
         47.6009
      ]
   }
});

db.usarestaurants.insert({
   "state":"Washington",
   "city":"Seattle",
   "name":"Noodle House",
   "rating": 4.8,
   "location":{
      "type":"Point",
      "coordinates":[
        -122.3517,
         47.6159
      ]
   }
});

db.usarestaurants.insert({
   "state":"Washington",
   "city":"Seattle",
   "name":"Curry House",
   "rating": 4.8,
   "location":{
      "type":"Point",
      "coordinates":[
         -121.4517,
         47.6229
      ]
   }
});
```

Querying geospatial data

Amazon DocumentDB supports proximity, inclusion and intersection querying of Geospatial data. A good example of a proximity query is finding all points (all airports) that are less than a certain distance and more than a distance from another point (city). A good example of inclusion querying is to find all points (all airports) that located in a specified area/polygon (state of New York). A

good example of intersection query is finding a polygon (state) which intersects with a point (city). You can use the following Geospatial operators to gain insights from your data.

- **`$nearSphere`** - `$nearSphere` is a find operator that supports finding points from nearest to farthest from a GeoJSON point.
- **`$geoNear`** - `$geoNear` is an aggregation operator that supports calculating the distance in meters from a GeoJSON point.
- **`$minDistance`** - `$minDistance` is a find operator that is used in conjunction with `$nearSphere` or `$geoNear` to filter documents that are at least at the specified minimum distance from the center point.
- **`$maxDistance`** - `$maxDistance` is a find operator that is used in conjunction with `$nearSphere` or `$geoNear` to filter documents that are at most at the specified maximum distance from the center point.
- **`$geoWithin`** - `$geoWithin` is a find operator that supports finding documents with geospatial data that exists entirely within a specified shape such as a polygon.
- **`$geoIntersects`** - `$geoIntersects` is a find operator that supports finding documents whose geospatial data intersects with a specified GeoJSON object.

> **Note**
>
> `$geoNear` and `$nearSphere` require a 2dsphere index on the GeoJSON field that you use in your proximity query.

Example 1

In this example, you will learn how to find all restaurants (points) sorted by closest distance from an address (point).

To perform such a query, you can use `$geoNear` to calculate distance of set of points from another point. You can also add the `distanceMultiplier` to measure the distance in kilometers.

```
db.usarestaurants.aggregate([
   {
      "$geoNear":{
         "near":{
            "type":"Point",
```

```
            "coordinates":[
               -122.3516,
               47.6156
            ]
         },
         "spherical":true,
         "distanceField":"DistanceKilometers",
         "distanceMultiplier":0.001
      }
   }
])
```

The command above would return restaurants sorted by distance (closest to furthest) from the point specified. The output of this command would look something like this

```
{ "_id" : ObjectId("611f3da985009a81ad38e74b"), "state" : "Washington", "city" :
 "Seattle", "name" : "Noodle House", "rating" : 4.8, "location" : { "type" : "Point",
 "coordinates" : [ -122.3517, 47.6159 ] }, "DistanceKilometers" : 0.03422834547294996 }
{ "_id" : ObjectId("611f3da185009a81ad38e74a"), "state" : "Washington", "city" :
 "Seattle", "name" : "Thai Palace", "rating" : 4.8, "location" : { "type" : "Point",
 "coordinates" : [ -122.3264, 47.6009 ] }, "DistanceKilometers" : 2.5009390081704277 }
{ "_id" : ObjectId("611f3dae85009a81ad38e74c"), "state" : "Washington", "city" :
 "Seattle", "name" : "Curry House", "rating" : 4.8, "location" : { "type" : "Point",
 "coordinates" : [ -121.4517, 47.6229 ] }, "DistanceKilometers" : 67.52845344856914 }
```

Note

The `limit` and `num` options for the `$geoNear` stage, as well as the default limit of 100 documents, are deprecated in Amazon DocumentDB 8.0 (Engine patch Version 4.0.5748) and later. The following examples using `limit` and `num` apply only to Amazon DocumentDB 8.0 (Engine patch Version 4.0.4448) and earlier versions.

To limit the number of results in a query, use the `limit` or `num` option.

`limit`:

```
db.usarestaurants.aggregate([
   {
      "$geoNear":{
         "near":{
```

```
            "type":"Point",
            "coordinates":[
               -122.3516,
               47.6156
            ]
         },
         "spherical":true,
         "distanceField":"DistanceKilometers",
         "distanceMultiplier":0.001,
         "limit": 10
      }
   }
])
```

num:

```
db.usarestaurants.aggregate([
   {
      "$geoNear":{
         "near":{
            "type":"Point",
            "coordinates":[
               -122.3516,
               47.6156
            ]
         },
         "spherical":true,
         "distanceField":"DistanceKilometers",
         "distanceMultiplier":0.001,
         "num": 10
      }
   }
])
```

Example 2

In this example, you will learn how to find all restaurants (points) within 2 kilometers of a specific address (point). To perform such a query, you can use `$nearSphere` within a minimum `$minDistance` and maximum `$maxDistance` from a GeoJSON Point

```
db.usarestaurants.find({
   "location":{
```

```
      "$nearSphere":{
         "$geometry":{
            "type":"Point",
            "coordinates":[
               -122.3516,
               47.6156
            ]
         },
         "$minDistance":1,
         "$maxDistance":2000
      }
   }
},
{
   "name":1
})
```

The command above would return restaurants at a maximum distance of 2 kilometers from the point specified. The output of this command would look something like this

```
{ "_id" : ObjectId("611f3da985009a81ad38e74b"), "name" : "Noodle House" }
```

Limitations

Amazon DocumentDB does not support querying or indexing of Polygons, LineString, MultiPoint, MultiPolygon, MultiLineString, and GeometryCollection.

Partial index

A partial index indexes documents in a collection that meets a specified filter criterion. The partial index feature is supported in Amazon DocumentDB 5.0 instance-based clusters.

Topics

- Create a partial index
- Supported operators
- Query using a partial index
- Partial index functionalities
- Partial index limitations

Create a partial index

To create a partial index, use the `createIndex()` method with the `partialFilterExpression` option. For example, the following operation creates a unique compound index in the orders collection that indexes documents having an `OrderID` and having the `isDelivered` field as true:

```
db.orders.createIndex(
  {"category": 1, "CustomerId": 1, "OrderId": 1},
  {"unique": true, "partialFilterExpression":
    {"$and": [
      {"OrderId": {"$exists": true}},
      {"isDelivered": {"$eq": false}}
    ]}
  }
)
```

Supported operators

- **$eq**
- **$exists**
- **$and** (only at top-level)
- **$gt/$gte/$lt/$lte** (index scan is only used when the filter, predicated in the query, matches the partial filter expression exactly) (See Limitations)

Query using a partial index

The following query patterns are possible using partial indexes:

- The query predicate exactly matches the partial index filter expression:

```
db.orders.find({"$and": [
    {"OrderId": {"$exists": true}},
    {"isDelivered": {"$eq": false}}
  ]}).explain()
```

- The query filter's expected result is a logical subset of the partial filter:

```
db.orders.find({"$and": [
```

```
    {"OrderId": {"$exists": true}},
    {"isDelivered": {"$eq": false}},
    {"OrderAmount": {"$eq": "5"}}
  ]}).explain()
```

- A sub-predicate of the query can be used in conjunction with other indexes:

```
db.orders.createIndex({"anotherIndex":1})
db.orders.find({ "$or": [
      {"$and": [
        {"OrderId": {"$exists": true}},
        {"isDelivered": {"$eq": false}}
      ]},
      {"anotherIndex": {"$eq": 5}}
    ]
  }).explain()
```

> **Note**
>
> A query planner may opt to use a collection scan rather than an index scan if it is efficient to do so. This is typically seen for very small collections or queries that would return a large portion of a collection.

Partial index functionalities

List partial indexes

List partial indexes with partialFilterExpression using the `getIndex` operation. For example, the `getIndex` operation issued in lists partial indexes with key, name, and partialfilterExpressions fields:

```
db.orders.getIndexes()
```

This example returns the following output:

```
[
  {
    "v" : 4,
    "key" : {
```

```
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "ecommerceApp.orders"
  },
  {
    "v" : 4,
    "unique" : true,
    "key" : {
      "category" : 1,
      "" : 1,
      "CustomerId" : 1,
      "OrderId" : 1
    },
    "name" : "category_1_CustID_1_OrderId_1",
    "ns" : "ecommerceApp.orders",
    "partialFilterExpression" : {
      "$and" : [
        {"OrderId": {"$exists": true}},
        {"isDelivered": {"$eq": false}}
      ]
    }
  }
]
```

Multiple partial filter expression on same key:order

Different partial indexes can be created for the same field combinations (key:order). These indexes must have a different name.

```
db.orders.createIndex(
  {"OrderId":1},
  {
    name:"firstPartialIndex",
    partialFilterExpression:{"OrderId":{"$exists": true}}
  }
)
```

```
db.orders.createIndex(
  {"OrderId":1},
  {
    name:"secondPartialIndex",
    partialFilterExpression:{"OrderId":{"$gt": 1000}}
```

```
  }
)
```

Run `getIndexes` operation to list all indexes in the collection:

```
db.orders.getIndexes()
```

These examples returns the following output:

```
[
  {
    "v" : 4,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "ecommerceApp.orders"
  },
  {
    "v" : 4,
    "key" : {
      "OrderId" : 1
    },
    "name" : "firstPartialIndex",
    "ns" : "ecommerceApp.orders",
    "partialFilterExpression" : {"OrderId":{"$exists": true}}
  },
  {
    "v" : 4,
    "key" : {
      "OrderId" : 1
    },
    "name" : "secondPartialIndex",
    "ns" : "ecommerceApp.orders",
    "partialFilterExpression" : {"OrderId":{"$gt": 1000}}
  }
]
```

Important

Index names must be different and must be deleted by name only.

Indexes with partial and TTL properties

You can also create indexes having partial and TTL properties by specifying both `partialFilterExpression` and `expireAfterSeconds` options during index creation. This allows you to provide more control over which documents are now removed from a collection.

For example, you may have a TTL index that identifies documents to be deleted after a certain time period. You can now provide extra conditions on when to delete documents using the partial index option:

```
db.orders.createIndex(
    { "OrderTimestamp": 1 },
    {
        expireAfterSeconds: 3600 ,
        partialFilterExpression: { "isDelivered": { $eq: true } }
    }
)
```

This example returns the following output:

```
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1,
        "operationTime" : Timestamp(1234567890, 1)
}
```

Run the `getIndexes` operation to list indexes present in the collection:

```
db.orders.getIndexes()
[
    {
        "v" : 4,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "test.orders"
    }
```

This example returns the following output:

```
[
    {
        "v": 4,
        "key": {
            "_id": 1
        },
        "name": "_id_",
        "ns": "ecommerceApp.orders"
    },
    {
        "v": 4,
        "key": {
            "OrderTimestamp": 1
        },
        "name": "OrderTimestamp_1",
        "ns": "ecommerceApp.orders",
        "partialFilterExpression": {
            "isDelivered": {
                "$eq": true
            }
        },
        "expireAfterSeconds": 3600
    }
]
```

Partial index limitations

The following limitations apply to the partial index feature:

- Inequality queries in Amazon DocumentDB will only use a partial index when the query filter predicate exactly matches the `partialFilterExpression` and is of the same datatype.

  > **Note**
  >
  > Even `$hint` cannot be used to force IXSCAN for the above case.

  In the following example, the `partialFilterExpression` is only applied to `field1` but not `field2`:

```
db.orders.createIndex(
```

```
  {"OrderAmount": 1},
  {"partialFilterExpression": { OrderAmount : {"$gt" : 5}}}
)

db.orders.find({OrderAmount : {"$gt" : 5}}) // Will use partial index
db.orders.find({OrderAmount : {"$gt" : 6}}) // Will not use partial index
db.orders.find({OrderAmount : {"$gt" : Decimal128(5.00)}}) // Will not use partial
 index
```

- A `partialFilterExpression` with array operators are not supported. The following operation will generate an error:

```
db.orders.createIndex(
  {"CustomerId":1},
  {'partialFilterExpression': {'OrderId': {'$eq': [1000, 1001, 1002]}}}
)
```

- The following operators are not supported in partialFilterExpression field:
  - `$all` (array operator)
  - $mod (array operator)
  - `$or`
  - `$xor`
  - `$not`
  - `$nor`
- The data type of the filter expression and the filter should be the same.

Performing text search with Amazon DocumentDB

Amazon DocumentDB's native full text search feature (Text Index v1) allows you to perform text search on large textual data sets using special purpose text indexes. This section describes the functionalities of the text index feature and provides steps on how to create and use text indexes in Amazon DocumentDB. Text search limitations are also listed.

Topics

- Supported functionalities
- Using Amazon DocumentDB text index
- Differences with MongoDB

- Best practices and guidelines
- Text Index V2
- Limitations

Supported functionalities

Amazon DocumentDB text search supports the following MongoDB API compatible functionalities:

- Create text indexes on a single field.
- Create compound text indexes that include more than one text field.
- Perform single word or multi-word searches.
- Control search results using weights.
- Sort search results by score.
- Use text index in aggregation pipeline.
- Search for exact phrase.

Using Amazon DocumentDB text index

To create a text index on a field containing string data, specify the string “text” as shown below:

Single field index:

```
db.test.createIndex({"comments": "text"})
```

This index supports text search queries in the "comments" string field in the specified collection.

Create a compound text index on more than one string field:

```
db.test.createIndex({"comments": "text", "title":"text"})
```

This index supports text search queries in the "comments" and "title" string fields in the specified collection. You can specify up to 30 fields when creating a compound text index. Once created, your text search queries will query all the indexed fields.

> **Note**
>
> Only one text index is allowed on each collection.

Listing a text index on an Amazon DocumentDB collection

You can use `getIndexes()` on your collection to identify and describe indexes, including text indexes, as shown in the following example:

```
rs0:PRIMARY> db.test.getIndexes()
[
   {
      "v" : 4,
      "key" : {
         "_id" : 1
      },
      "name" : "_id_",
      "ns" : "test.test"
   },
   {
      "v" : 1,
      "key" : {
         "_fts" : "text",
         "_ftsx" : 1
      },
      "name" : "contents_text",
      "ns" : "test.test",
      "default_language" : "english",
      "weights" : {
         "comments" : 1
      },
      "textIndexVersion" : 1
   }
]
```

Once you have created an index, start inserting data into your Amazon DocumentDB collection.

```
db.test.insertMany([{"_id": 1, "star_rating": 4, "comments": "apple is red"},
                    {"_id": 2, "star_rating": 5, "comments": "pie is delicious"},
                    {"_id": 3, "star_rating": 3, "comments": "apples, oranges - healthy
 fruit"},
```

```
                    {"_id": 4, "star_rating": 2, "comments": "bake the apple pie in the
oven"},
                    {"_id": 5, "star_rating": 5, "comments": "interesting couch"},
                    {"_id": 6, "star_rating": 5, "comments": "interested in couch for
sale, year 2022"}])
```

Running text search queries

Run a single-word text search query

You will need to use `$text` and `$search` operators to perform text searches. The following example returns all documents where your text indexed field contain the string “apple” or “apple” in other formats such as “apples”:

```
db.test.find({$text: {$search: "apple"}})
```

Output:

The output of this command looks something like this:

```
{ "_id" : 1, "star_rating" : 4, "comments" : "apple is red" }
{ "_id" : 3, "star_rating" : 3, "comments" : "apples, oranges - healthy fruit" }
{ "_id" : 4, "star_rating" : 2, "comments" : "bake the apple pie in the oven" }
```

Run a multi-word text search

You can also perform multi-word text searches on your Amazon DocumentDB data. The command below returns documents with a text indexed field containing “apple” or “pie”:

```
db.test.find({$text: {$search: "apple pie"}})
```

Output:

The output of this command looks something like this:

```
{ "_id" : 1, "star_rating" : 4, "comments" : "apple is red" }
{ "_id" : 2, "star_rating" : 5, "comments" : "pie is delicious" }
{ "_id" : 3, "star_rating" : 3, "comments" : "apples, oranges - healthy fruit" }
{ "_id" : 4, "star_rating" : 2, "comments" : "bake the apple pie in the oven" }
```

Run a multi-word phrase text search

For a multi-word phrase search, use this example:

```
db.test.find({$text: {$search: "\"apple pie\""}})
```

Output:

The command above returns documents with text indexed field containing the exact phrase “apple pie”. The output of this command looks something like this:

```
{ "_id" : 4, "star_rating" : 2, "comments" : "bake the apple pie in the oven" }
```

Run a text search with filters

You can also combine text search with other query operators to filter results based on additional criteria:

```
db.test.find({$and: [{star_rating: 5}, {$text: {$search: "interest"}}]})
```

Output:

The command above returns documents with a text indexed field containing any form of “interest” and a “star_rating” equal to 5. The output of this command looks something like this:

```
{ "_id" : 5, "star_rating" : 5, "comments" : "interesting couch" }
{ "_id" : 6, "star_rating" : 5, "comments" : "interested in couch for sale, year
 2022" }
```

Limit the number of documents returned in a text search

You can choose to restrict the number of documents returned by using `limit`:

```
db.test.find({$and: [{star_rating: 5}, {$text: {$search: "couch"}}]}).limit(1)
```

Output:

The command above returns one result that satisfies the filter:

```
{ "_id" : 5, "star_rating" : 5, "comments" : "interesting couch" }
```

Sort results by text score

The following example sorts the text search results by text score:

```
db.test.find({$text: {$search: "apple"}}, {score: {$meta: "textScore"}}).sort({score:
 {$meta: "textScore"}})
```

Output:

The command above returns documents with a text indexed field containing “apple”, or “apple” in it's other formats like “apples”, and sorts the result based on how relevant the document is related to the search term. The output of this command looks something like this:

```
{ "_id" : 1, "star_rating" : 4, "comments" : "apple is red", "score" :
 0.6079270860936958 }
{ "_id" : 3, "star_rating" : 3, "comments" : "apples, oranges - healthy fruit",
 "score" : 0.6079270860936958 }
{ "_id" : 4, "star_rating" : 2, "comments" : "bake the apple pie in the oven",
 "score" : 0.6079270860936958 }
```

`$text` and `$search` are also supported for `aggregate`, `count`, `findAndModify`, `update`, and `delete` commands.

Aggregation operators

Aggregation pipeline using `$match`

```
db.test.aggregate(
   [{ $match: { $text: { $search: "apple pie" } } }]
)
```

Output:

The command above returns the following results:

```
{ "_id" : 1, "star_rating" : 4, "comments" : "apple is red" }
{ "_id" : 3, "star_rating" : 3, "comments" : "apples, oranges - healthy fruit" }
{ "_id" : 4, "star_rating" : 2, "comments" : "bake the apple pie in the oven" }
{ "_id" : 2, "star_rating" : 5, "comments" : "pie is delicious" }
```

A combination of other aggregation operators

```
db.test.aggregate(
   [
```

```
      { $match: { $text: { $search: "apple pie" } } },
      { $sort: { score: { $meta: "textScore" } } },
      { $project: { score: { $meta: "textScore" } } }
   ]
)
```

Output:

The command above returns the following results:

```
{ "_id" : 4, "score" : 0.6079270860936958 }
{ "_id" : 1, "score" : 0.3039635430468479 }
{ "_id" : 2, "score" : 0.3039635430468479 }
{ "_id" : 3, "score" : 0.3039635430468479 }
```

Specify multiple fields when creating a text index

You can assign weights to up to three fields in your compound text index. The default weight assigned to a field in a text index is one (1). Weight is an optional parameter and must be in the range from 1 to 100000.

```
db.test.createIndex(
   {
     "firstname": "text",
     "lastname": "text",
     ...
   },
   {
     weights: {
       "firstname": 5,
       "lastname":10,
       ...
     },
     name: "name_text_index"
   }
 )
```

Differences with MongoDB

Amazon DocumentDB's text index feature uses inverted index with a term-frequency algorithm. Text indexes are sparse by default. Due to differences in parsing logic, tokenization delimiters, and others, the same result set as MongoDB may not be returned for the same dataset or query shape.

The following additional differences between Amazon DocumentDB text index and MongoDB exist:

- Compound indexes using non-text indexes are not supported.
- Amazon DocumentDB text indexes are case insensitive.
- Only English language is supported with text index.
- Text indexing of array (or multi-key) fields is not supported. For example, creating a text index on “a“ with the document {“a”:[“apple”, “pie”]} will fail.
- Wildcard text indexing is not supported.
- Unique text indexes are not supported.
- Excluding a term is not supported.

Best practices and guidelines

- For optimal performance on text search queries involving sorting by text scores, create the text index before loading data.
- Text indexes require additional storage for an optimized internal copy of the indexed data. This has additional cost implications.

Text Index V2

Amazon DocumentDB 8.0 introduces a new version of text index (V2) that changes underlying text search parser to bring more compatibility with the MongoDB.

In addition to the functionalities provided by the V1 text indexes, V2 text indexes also provide the following support:

- The planner moves $match stages earlier in the pipeline when possible, reducing the number of documents processed by subsequent stages.
- Improved tokenization of special characters in text fields, such as emails, URLs, and file paths. V2 can parse and match individual tokens within these strings, whereas V1 could not.

```
rs0:PRIMARY> db.coll.createIndex({ "a": "text" });
rs0:PRIMARY> db.coll.find()
{ "_id" : 1, "a" : "jane.doe_1234@company.com" }
{ "_id" : 2, "a" : "janedoe@company.com" }
{ "_id" : 3, "a" : "/home/user/company/thesis.pdf" }
```

```
{ "_id" : 4, "a" : "/home/user/path/jane.pdf" }
{ "_id" : 5, "a" : "http://www.company.com/path" }
{ "_id" : 6, "a" : "https://company.com/path/../home" }

// Sample query
rs0:PRIMARY> db.coll.find({ $text: { $search: "jane" } });

// V1 text index – no results
None

// V2 text index – matches tokens within emails and file paths
{ "_id" : 1, "a" : "jane.doe_1234@company.com" }
{ "_id" : 4, "a" : "/home/user/path/jane.pdf" }
```

Limitations

Text search has the following limitations in Amazon DocumentDB:

- Text indexes store lexemes and their position information. The combined size of all lexemes and their position information, within a single document, is limited to 1MB.

Indexes

Indexes improve query performance by allowing Amazon DocumentDB to quickly locate documents without scanning every document in a collection. You can create indexes on fields that your application queries frequently to reduce latency and I/O costs.

For additional index-related topics, see:

- Managing Amazon DocumentDB indexes — creating and building indexes
- Working with indexes — indexing best practices
- Troubleshooting indexes — resolving index build failures and bloat
- How do I analyze index usage and identify unused indexes? — finding unused and missing indexes

Topics

- Index management

- Index types
- Index properties

Index management

This section covers index management topics for Amazon DocumentDB.

Topics

- Index creation
- Maintaining indexes
- Best practices for working with indexes

Index creation

Building indexes in Amazon DocumentDB requires a number of decisions to be made:

- How quickly does it need to be completed?
- Can the collection be inaccessible while the build is occurring?
- How much of an instance's compute power can be allocated to the build?
- What type of index should be created?

This section helps you answer these questions and provides the commands and monitoring examples to create an Amazon DocumentDB index on your instance-based cluster collection.

Guidelines

The following guidelines include basic limits and configuration tradeoffs when creating new indexes:

- **Amazon DocumentDB version support** - While single worker indexing is supported on all Amazon DocumentDB versions, multiple worker indexing is supported only on Amazon DocumentDB versions 4.0 and 5.0.
- **Performance trade-off** - Increasing the number of workers in the index creation process increases CPU utilization and read IO on the primary instance of your Amazon DocumentDB database. The resources needed to create a new index will not be available to your running workload.

- **Elastic clusters** - Parallel indexing is not supported on Amazon DocumentDB elastic clusters.
- **Serverless clusters** - Parallel indexing on serverless clusters requires a minimum capacity of at least 32 DCUs. Below this threshold, the `workers` parameter is accepted but the index build executes with a single worker. The maximum number of parallel workers scales with your configured minimum capacity: approximately `min_DCU / 8` workers are available. For example, at 32 DCUs you can use up to 4 workers, and at 64 DCUs you can use up to 8 workers.

  > **Note**
  >
  > A cluster restart is required after changing the minimum capacity setting for the new parallel worker limit to take effect.

- **Maximum workers (instance-based)** - The maximum number of workers you can configure depends on the size of your primary instance in your database cluster. It is half the total number of vCPUs on the primary instance of your database cluster. For example, you can run a maximum of 32 workers on a db.r6g.16xlarge instance that has 64 vCPUs.

> **Note**
>
> Parallel index build is supported on instances 2xlarge and above.

- **Minimum workers** - The minimum number of workers you can configure is one. The default setting for index creation on instance-based clusters is two workers. However, you can reduce the number of workers to one by using the "worker threads" option. This will run the process with a single worker.
- **Index compression** - Amazon DocumentDB doesn't support index compression. Data sizes for indexes might be larger than when you use other options.
- **Indexing multiple collections** - Half the vCPUs on your database cluster's primary instance can be used for configured workers performing index creation on multiple collections.
- **Index types** - See Index types for the list of supported index types on Amazon DocumentDB.

Index build types

The four types of index builds are:

- **Foreground:** The foreground index build blocks all other database operations on the collection the index is created on, until the index creation is completed. The Amazon DocumentDB foreground build is comprised of five stages.
- **Foreground (unique):** Single document (unique) foreground index builds block other database operations like regular foreground builds. Unlike the basic foreground build, the unique build uses an additional stage (sorting keys 2) to look for duplicate keys. The foreground (unique) build is comprised of six stages.
- **Background:** The background index build allows other database operations to run in the foreground while the index is being created. The Amazon DocumentDB background build is comprised of eight stages.
- **Background (unique):** Single document (unique) background index builds allow other database operations to run in the foreground while the index is being created. Unlike the basic background build, the unique build uses an additional stage (sorting keys 2) to look for duplicate keys. The background (unique) build is comprised of nine stages.

The default index build type is foreground.

To build an index in background, you add the "background": true parameter when creating the index.

Getting started

To start index creation on a collection, use the createIndexes command. By default, the command will run two parallel workers that increases the speed of the index creation process by two times.

For example, the following command process demonstrates how to create an index for the "user_name" field in a document and increase the indexing process speed to four workers:

1. Create indexes in the background using two parallel workers on the cluster:

```
db.runCommand({"createIndexes":"test","indexes":[{"key": {"user_name":1},
 "name":"username_idx", "background":true }]})
```

2. To optimize the speed of the index creation process, you can specify the number of workers by using the "worker threads" option ("workers":<number>) in the db.runCommand createIndexes command.

Increase the speed of the process to four parallel workers:

```
db.runCommand({"createIndexes":"test","indexes":[{"key": {"user_name":1},
 "name":"username_idx", "workers":4}]})
```

> **Note**
>
> The higher the number of workers, the faster the index creation progresses. However, the higher the number of workers increases, the higher the load increases on the vCPUs and read IO of your primary instance. Ensure that your cluster is sufficiently provisioned to handle the increased burden without degrading other workloads.

Indexing progress status

The index creation process works by initializing, scanning collections, sorting keys, and, finally, inserting keys by way of an index builder. The process has up to six stages when you run it in the foreground, and up to nine stages when you run it in the background. You can view status metrics such as percentage completion, total number of scanned storage blocks, sorted keys, and inserted keys on stage by stage basis.

Monitor the progress on the indexing process by using the db.currentOp() command in the mongo shell. A 100% completion of the last stage shows that all the indexes have been successfully created:

```
db.currentOp({"command.createIndexes": { $exists : true } })
```

> **Note**
>
> Viewing the indexing progress status is only supported on Amazon DocumentDB 5.0.

Index build stages

Stage	Foreground	Foreground (unique)	Background	Background (unique)
Initializing	1	1	1	1
building index: initializing	2	2	2	2
building index: scanning collection	3	3	3	3
building index: sorting keys 1	4	4	4	4
building index: sorting keys 2		5		5
building index: inserting keys	5	6	5	6
validating: scanning index			6	7
validating: sorting tuples			7	8
validating: scanning collection			8	9

- **initializing** - createIndex is preparing the index builder. This phase should be very brief.
- **building index: initializing** - The index builder is preparing to create the index. This phase should be very brief.
- **building index: scanning collection** - The index builder is performing a collection scan to collect index keys. The unit of measure is "blocks".

> **Note**
>
> If more than one worker is configured for the index build, it is displayed in this stage. The "scanning collection" stage is the only stage that uses multiple workers during the index build process. All other stages will display a single worker.

- **building index: sorting keys 1** - The index builder is sorting the collected index keys. The unit of measure is "keys".
- **building index: sorting keys 2** - The index builder is sorting the collected index keys that correspond to dead tuples. This phase only exists for unique index building. The unit of measure is "keys".
- **building index: inserting keys** - The index builder is inserting index keys into the new index. The unit of measure is "keys".
- **validating: scanning index** - createIndex is scanning the index to find keys that need to be validated. The unit of measure is "blocks".
- **validating: sorting tuples** - createIndex is sorting the output of the index scanning phase.
- **validating: scanning collection** - createIndex is scanning the collection to validate the index keys found in the previous two phases. The unit of measure is "blocks".

Index build output example

In the following output example (foreground index build), the status of the index creation is shown. The "msg" field summarizes the build progress by indicating the stage and the completion percentage of the build. The "workers" field indicates the number of workers used during that stage of the index build. The "progress" field shows the actual numbers used to calculate the percentage of completion.

> **Note**
>
> The "currentIndexBuildName", "msg", and "progress" fields are not supported on Amazon DocumentDB version 4.0.

```
{
```

```
    "inprog" : [{
    …
        "command": {
            "createIndexes": "test",
            "indexes": [{
                "v": 2,
                "key": {
                    "user_name": 1
                },
                "name": "user_name_1"
            }],
            "lsid": {
                "id": UUID("094d0fba-8f41-4373-82c3-7c4c7b5ff13b")
            },
            "$db": "test"
        },
        "currentIndexBuildName": user_name_1,
        "msg": "Index Build: building index number_1, stage 6/6 building index:
 656860/1003520 (keys) 65%",
        "workers": 1,
        "progress": {
            "done": 656861,
            "total": 1003520
        },
    …
    ],

    "ok" : 1
}
```

Troubleshooting resources

See the following resources on troubleshooting index creation issues:

- Index build fails
- Background index build latency issues and fails

Maintaining indexes

Index bloat

Amazon DocumentDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent transactions. When documents are deleted or updated, their previous versions remain in collections and indexes as "dead" versions. The garbage collection process automatically reclaims space from these dead versions for future operations.

Index bloat occurs when a collection's indexes become larger due to the accumulation of dead or obsolete index entries or fragmentation within the pages. The percentage reported represents the amount of index space that can be used by future index entries. This bloat consumes space in both the buffer cache and storage. If you want to remove the bloat, you will need to rebuild indexes.

Example: Run the following command to determine unused storage for your index:

```
db.coll.aggregate({$indexStats:{}});
```

The result looks similar to this:

```
{
    "name" : "_id_",
    "key" : {
        "_id" : 1
    },
    "host" : "devbox-test.localhost.a2z.com:27317",
    "size" : NumberLong(827392),
    "accesses" : {
        "ops" : NumberLong(40000),
        "docsRead" : NumberLong(46049),
        "since" : ISODate("2025-04-03T21:44:51.251Z")
    },
    "cacheStats" : {
        "blksRead" : NumberLong(264),
        "blksHit" : NumberLong(140190),
        "hitRatio" : 99.8121
    },
    "unusedStorageSize" : {
```

```
        "unusedSizeBytes" : 409600,
        "unusedSizePercent" : 49.51
    }
}
```

You can rebuild indexes without downtime using the reIndex command, which requires a scan of the entire collection. See Index maintenance using reIndex.

Index maintenance using reIndex

reIndex is a command used to rebuild an index. It is typically used when an index has become corrupted or inefficient. Over time, indexes can accumulate unused space due to many updates, inserts, or deletes, leading to degraded performance. Reindexing helps to remove such unused space and restore the efficiency of the index.

reIndex guidelines

- reIndex is only supported on Amazon DocumentDB 5.0 and above
- reIndex always runs in the background.
- Amazon DocumentDB supports reindex of a single index in the background, allowing for multiple workers. The old index is usable by queries when the reIndex process is running.
- Amazon DocumentDB supports indexing progress report through currentOp. You can see index build stages similar to the Index build stages viewed during index creation. The only difference is that reIndex always has eight stages, regardless if it's unique or not. There's no "building index: sorting keys 2" stage.
- reIndex can run concurrently with any command except index-related commands on the same collection: createIndexes, dropIndexes, collMod, and renameCollection.
- reIndex is currently not supported for text, geospatial, and vector indexes.
- reIndex support for partial indexes is available in engine version 8.0.2 and later.

reIndex build

Use the following command to rebuild your index:

```
db.runCommand({ reIndex: "collection-name", index: "index-name"})
```

Optionally, you can also control the number of workers assigned to the rebuild process:

```
db.runCommand({ reIndex: "collection-name", index: "index-name", workers: number })
```

For information specific to managing indexes with Java, see Index management in Amazon DocumentDB with Java.

Best practices for working with indexes

Building indexes

When importing data into Amazon DocumentDB, you should create your indexes before importing large datasets. You can use the Amazon DocumentDB Index Tool to extract indexes from a running MongoDB instance or mongodump directory, and create those indexes in an Amazon DocumentDB cluster. For more guidance on migrations, see *Migrating and upgrading*.

Index selectivity

Limit index creation to fields where the number of duplicate values is less than 1% of the total number of documents in the collection. As an example, if your collection contains 100,000 documents, only create indexes on fields where the same value occurs 1,000 times or fewer.

Choosing an index with a high number of unique values (that is, a high cardinality) ensures that filter operations return a small number of documents, thereby yielding good performance during index scans. An example of a high-cardinality index is a unique index, which guarantees that equality predicates return at most a single document. Examples of low-cardinality include an index over a Boolean field and an index over day of the week. Due to their poor performance, low cardinality indexes are unlikely to be chosen by the database's query optimizer. At the same time, low cardinality indexes continue to consume resources such as disk space and I/Os. As a rule of thumb, you should target indexes on fields where the typical value frequency is 1% of the total collection size or less.

Additionally, it is recommended to only create indexes on fields that are commonly utilized as a filter and regularly look for unused indexes. For more information, see the section called “Analyze index usage and identify unused indexes”.

Impact of indexes on writing data

While indexes can improve query performance by avoiding the need to scan every document in a collection, this improvement comes with a tradeoff. For each index on a collection, every time a document is inserted, updated, or deleted, the database must update the collection and write the fields to each of the indexes for the collection. For example, if a collection has nine indexes, the database must perform ten writes before acknowledging the operation to the client. Thus, each additional index incurs additional write latency, I/Os, and increase in the overall utilized storage.

Cluster instances need to be appropriately sized to keep all working set memory. This avoids the need to continuously read index pages from the storage volume, which negatively impacts performance and generates higher I/O costs. For more information, see Instance sizing.

For best performance, minimize the number of indexes in your collections, adding only those indexes necessary to improve performance for common queries. While workloads vary, a good guideline is to keep the number of indexes per collection to five or fewer.

Identifying missing indexes

As a best practice, identify missing indexes on a regular basis. For more information, see the section called “Identify missed indexes”.

Identifying unused indexes

As a best practice, identify and remove unused indexes on a regular basis. For more information, see the section called “Analyze index usage and identify unused indexes”.

Index types

Topics

- Single field indexes
- Compound indexes
- Multi-key indexes
- Wildcard indexes
- Text indexes
- Geospatial indexes
- Vector indexes

Single field indexes

Single field indexes store information from any single field in a collection of documents. By default, all collections have an index on the `_id` field. You can add additional indexes to speed up important queries and operations.

Single field indexes are beneficial when:

- Your application frequently queries or filters documents based on the values of a specific field.
- You need to efficiently sort documents by a particular field.
- You want to ensure uniqueness for a specific field across documents in a collection by creating a unique index.

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	Yes	Yes	Yes	Yes	Yes
unique	Yes	Yes	Yes	Yes	Yes
sparse *	Yes	Yes	Yes	Yes	Yes
partialFilterExpression *	No	No	Yes	Yes	No
expireAfterSeconds	Yes	Yes	Yes	Yes	Yes

* The `sparse` and `partialFilterExpression` options cannot be used together in the same index definition. If you attempt to create an index with these options, it will fail with the following error:

```
Error in specification: cannot mix partialFilterExpression and sparse options
```

Creating a single field index

Use the `createIndex()` method to create a single field index. The syntax is: `db.collection.createIndex(<key>, <options>)`.

The `key` parameter is a JSON document that specifies the field and the index sort order:

```
{
  "<field>": <1 (ascending)|-1 (descending)>
}
```

The `options` parameter is a JSON document that specifies the options for the index:

```
{
  "name": "<name>",
  "unique": <true | false>,
  "sparse": <true | false>,
  "partialFilterExpression": <filter expression>,
  "expireAfterSeconds": <seconds before expiry>
}
```

See Index properties for examples of creating single field indexes.

Compound indexes

Compound indexes store information from two or more fields in a collection of documents, allowing queries on the first field or any prefix fields. These indexes optimize performance for queries that filter on multiple fields simultaneously or combine filtering with sorting operations. They're also effective for single-condition queries on the leftmost indexed fields. The database leverages these index entries to efficiently locate matching documents without performing full collection scans.

Compound field indexes are beneficial when:

- You need to filter on multiple fields simultaneously.
- You need to combine filtering with sorting operations.

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	Yes	Yes	Yes	Yes	Yes
unique	Yes	Yes	Yes	Yes	Yes
sparse *	Yes	Yes	Yes	Yes	Yes
partialFilterExpression *	No	No	Yes	Yes	No

* The `sparse` and `partialFilterExpression` options cannot be used together in the same index definition. If you attempt to create an index with these options, it will fail with the following error:

```
Error in specification: cannot mix partialFilterExpression and sparse options
```

Creating a compound index

Use the `createIndex()` method to create a compound index. The method syntax is: `db.collection.createIndex(<keys>, <options>)`

The `keys` parameter is a JSON document that specifies the fields and the index sort order:

```
{
  "<field 1>": <1 (ascending)|-1 (descending)>,
  "<field 2>": <1 (ascending)|-1 (descending)>,
  ...
}
```

Note that only one field can be an array in a compound index. If you attempt to create a compound index on two or more array fields, it will fail with the following error:

```
multiple fields of compound index cannot be arrays
```

The `options` parameter is a JSON document that specifies the options for the index:

```
{
  "name": "<name>",
  "unique": <true | false>,
  "sparse": <true | false>,
  "partialFilterExpression": <filter expression>
}
```

See Index properties for examples of creating compound indexes.

Multi-key indexes

For fields that have an array value, a multi-key index allows you to create an index key for each element in the array. Indexing an array creates an index entry for each element of the array.

Multi-key indexes are beneficial when your application frequently queries or filters documents based on values in arrays.

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	Yes	Yes	Yes	Yes	Yes
unique	Yes	Yes	Yes	Yes	Yes
sparse *	Yes	Yes	Yes	Yes	Yes
partialFilterExpression *	No	No	Yes	Yes	No
expireAfterSeconds	Yes	Yes	Yes	Yes	Yes

* The `sparse` and `partialFilterExpression` options cannot be used together in the same index definition. If you attempt to create an index with these options, it will fail with the following error:

```
Error in specification: cannot mix partialFilterExpression and sparse options
```

Creating a multi-key index

Use the `createIndex()` method to create a multi-key index. The method syntax is: `db.collection.createIndex(<key>, <options>)`

The `key` parameter is a JSON document that specifies the field and the index sort order:

```
{
  "<field>": <1 (ascending)|-1 (descending)>
}
```

The `options` parameter is a JSON document that specifies the options for the index:

```
{
  "name": "<name>",
  "unique": <true | false>,
  "sparse": <true | false>,
  "partialFilterExpression": <filter expression>,
  "expireAfterSeconds": <seconds before expiry>
}
```

The following example creates a multi-key index on the `categories` field sorted in ascending order with the name `book_categories`:

```
db.collection.createIndex(
  {
    "categories": 1
  },
  {
    "name": "book_categories"
  }
)
```

See Index properties for examples of creating multi-key indexes.

Wildcard indexes

Amazon DocumentDB supports wildcard indexes beginning with Amazon DocumentDB 8.0.3.

Wildcard indexes store information from every field in a document, or from every field under a path that you specify. You do not need to know the field names when you create the index. If a field contains an embedded document or an array, the index includes every scalar value nested inside it.

Wildcard indexes are beneficial when:

- Field names vary between documents in a collection, such as user-defined attributes or metadata.
- You often query an embedded document whose subfields are not the same in every document.
- You query many different fields, and creating a separate index for each field is not practical.

Wildcard indexes work best when field names vary between documents or change over time. When your queries use a small set of known fields, single field or compound indexes on those fields provide the best performance. Because each wildcard index entry stores the full field path, a wildcard index can be larger when documents have long or deeply nested field names.

Supported index properties

Option	3.6	4.0	5.0	8.0.3+	Elastic Cluster
name	No	No	No	Yes	No
partialFilterExpression	No	No	No	Yes	No
wildcardProjection	No	No	No	Yes	No
hidden	No	No	No	Yes	No

The `unique`, `sparse`, and `expireAfterSeconds` options are not supported on wildcard indexes. Wildcard indexes are sparse. They contain entries only for documents that have an indexed field, even if the field value is `null`.

Creating a wildcard index

Use the `createIndex()` method to create a wildcard index. The method syntax is: `db.collection.createIndex(<key>, <options>)`

The `key` parameter is a JSON document that specifies the wildcard path and the index sort order:

```
{
  "<path>.$**": <1 (ascending)|-1 (descending)>
}
```

Use `{"$**": 1}` to index all fields in each document, or `{"a.$**": 1}` to index all fields under the field `a`. `$**` must be the last element of the path.

The `options` parameter is a JSON document that specifies the options for the index:

```
{
  "name": "<name>",
  "partialFilterExpression": <filter expression>,
  "wildcardProjection": <projection>
}
```

The following example creates a wildcard index on all fields under the `supplier` field:

```
db.collection.createIndex(
  {
    "supplier.$**": 1
  },
  {
    "name": "supplier_wildcard"
  }
)
```

Wildcard projection

The `wildcardProjection` option lets you include or exclude fields from a wildcard index. Use it when you create a `{"$**": 1}` or `{"$**": -1}` index, or a compound wildcard index whose wildcard term is `$**`. In the projection, `1` includes a field and `0` excludes it. Include the fields that you query often to keep unneeded entries out of the index, or exclude fields that you never query.

The following example indexes all fields except `productEmbedding` and `reviews`:

```
db.collection.createIndex(
  {
    "$**": 1
  },
  {
    "name": "all_fields",
    "wildcardProjection": { "productEmbedding": 0, "reviews": 0 }
  }
)
```

The following rules apply to `wildcardProjection`:

- You can use `wildcardProjection` only when the wildcard term is `$**`. You cannot use it when the wildcard term is a field path, such as `"supplier.$**"`.
- You cannot mix inclusion and exclusion statements in one projection. The only exception is `_id`: you can exclude `_id` from an inclusion projection, such as `{a: 1, _id: 0}`, or include `_id` in an exclusion projection, such as `{a: 0, _id: 1}`.
- The projection cannot be empty.
- Wildcard indexes omit the `_id` field by default. To index `_id`, include `_id: 1` in the projection.
- The index supports queries only on the fields that the projection covers. A query on any other field cannot use the index.

Compound wildcard indexes

A compound wildcard index combines one wildcard term with one or more regular fields. Use it when your queries filter on a field that every document has, such as `manufacturer`, and also on fields whose names vary, such as the subfields of `supplier`.

The following example creates a compound index with a regular field and a wildcard field:

```
db.collection.createIndex(
  {
    "manufacturer": 1,
    "supplier.$**": 1
  },
  {
    "name": "manufacturer_supplier_wildcard"
  }
)
```

If the wildcard term is $**, you must specify a wildcardProjection. The projection applies only to the wildcard term. It selects which fields the wildcard term indexes. The fields that the projection includes must not overlap with the regular fields. If the projection is an exclusion, it must exclude every regular field. The following example indexes manufacturer as a regular field and every other field except productEmbedding and reviews:

```
db.collection.createIndex(
  {
    "manufacturer": 1,
    "$**": 1
  },
  {
    "name": "manufacturer_all_fields",
    "wildcardProjection": { "manufacturer": 0, "productEmbedding": 0, "reviews": 0 }
  }
)
```

The projection excludes manufacturer, but the index still indexes manufacturer as the regular field.

The following rules apply to compound wildcard indexes:

- A compound wildcard index can have only one wildcard term. The wildcard term can be in any position.
- You cannot include the same field in the wildcard term and in a regular field. For example, {o: 1, "o.$**": 1} is not allowed.
- Regular fields in a compound wildcard index cannot be arrays.
- Regular fields cannot include a numeric path component, such as 0 in reviews.0.review_date. For example, { "reviews.0.review_date": 1, "supplier.$**": 1 } is not allowed.
- A compound wildcard index cannot include a text or 2dsphere field.
- A query uses the index only if it includes a condition on the leading field of the index.

Limitations

A wildcard index can serve only one field in a query. If a find or $match filter has conditions on more than one field that the index includes, Amazon DocumentDB uses the index for the condition that appears first in the filter. It then checks the remaining conditions against the documents that

the index returns. For best performance, put the condition that matches the fewest documents first.

For example, with a `{"supplier.$**": 1}` index, the following query uses the index for `supplier.supplierId`, and then checks `supplier.location.type` on each document that the index returns:

```
db.collection.find({
  "supplier.supplierId": "SUP4",
  "supplier.location.type": "Point"
})
```

Wildcard indexes do not support queries that use `$exists: true`. Amazon DocumentDB runs these queries with a collection scan.

Each wildcard index entry stores the full field path with the value, so the index key size limit applies to the field path and value together. An insert, update, or `createIndex` operation fails if an indexed field exceeds the limit. To avoid this, exclude fields that can hold large values from the index. For more information, see Wildcard projection.

Text indexes

Text indexes are useful in searching for words or phrases in text fields within documents, allowing you to perform full-text search functionality. You can create single field text indexes or compound text indexes with more than one text field. However, you can only have one text index per collection. For more information, see Performing text search with Amazon DocumentDB.

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	No	No	Yes	Yes	No

Creating a text index

Use the `createIndex()` method to create a text index. The method syntax is: `db.collection.createIndex(<keys>, <options>)`

The `keys` parameter is a JSON document that specifies the field(s) and text index type:

```
{
  "<field 1>": "text",
  "<field 2>": "text"
}
```

The `options` parameter is a JSON document that specifies the name of the index:

```
{
  "name": "<name>"
}
```

See Index properties for examples of creating text indexes.

Geospatial indexes

Geospatial indexes are a specialized type of index designed to efficiently query and manage geospatial data stored within a collection of documents. Amazon DocumentDB supports 2dsphere indexes, which are specifically designed to handle geospatial data on a sphere (like the Earth). This allows for accurate calculations and queries based on spherical geometry.

Geospatial indexes are beneficial when your applications need to perform location-based queries, such as:

- finding nearby points of interest,
- determining if a location falls within a specific area
- calculating distances between locations

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	Yes	Yes	Yes	Yes	Yes

Creating a geospatial index

Use the `createIndex()` method to create a geospatial index. The method syntax is: `db.collection.createIndex(<key>, <options>)`

The `key` parameter is a JSON document that specifies the field and 2dsphere index type:

```
{
  "<field>": "2dsphere"
}
```

The `options` parameter is a JSON document that specifies the options for the index:

```
{
  "name": "<name>"
}
```

See Index properties for examples of creating geospatial indexes.

Vector indexes

Vector indexes are a specialized type of index designed to efficiently query and manage vector data stored within a collection of documents. Amazon DocumentDB supports Hierarchical Navigable Small World (HNSW) and Inverted File with Flat Compression (IVFFlat) indexes.

For more information, see Vector search for Amazon DocumentDB.

Vector indexes are beneficial for machine learning and generative AI use cases, such as:

- semantic search
- product recommendation
- personalization
- chatbots
- fraud detection
- anomaly detection

Supported index properties

Option	3.6	4.0	5.0	8.0	Elastic Cluster
name	No	No	Yes	Yes	No

Creating a vector index

Use the createIndex command with the `runCommand()` method to create a vector index. The syntax is:

```
db.runCommand({
  "createIndexes": "<collection>",
  "indexes": [{
    "key": {
      "<field>": "vector"
    },
    "name": "<name>",
    "vectorOptions": {
      "type": "<hnsw> | <ivfflat>",
      "dimensions": <number of dimensions>,
      "similarity": "<euclidean>|<cosine>|<dotProduct>",
      "lists": <number_of_lists> [applicable for IVFFlat],
      "m": <max number of connections> [applicable for HNSW],
      "efConstruction": <size of the dynamic list for index build> [applicable for
 HNSW]
    }
  }]
})
```

The key parameter is a JSON document that specifies the field and vector index type:

```
{
  "<field>": "vector"
}
```

See Index properties for examples of creating vector indexes.

Index properties

Index properties define the behavior and characteristics of indexes in Amazon DocumentDB. Understanding these properties helps you design efficient indexes that meet your application's specific requirements.

Topics

- Index Property: name

- Index Property: unique
- Index Property: sparse
- Index Property: partialFilterExpression
- Index Property: expireAfterSeconds
- Index property: hidden

Index Property: name

Supported index types

Option	3.6	4.0	5.0	8.0	Elastic Cluster
single field	Yes	Yes	Yes	Yes	Yes
compound	Yes	Yes	Yes	Yes	Yes
multi-key	Yes	Yes	Yes	Yes	Yes
text	No	No	Yes	Yes	No
geospatial	Yes	Yes	Yes	Yes	Yes
vector	No	No	Yes	Yes	No
wildcard	No	No	No	Yes	No

Use the name option to provide an optional name for the index.

All examples use the following sample document:

```
{
  "productId": "PROD133726",
  "sku": "SKU24224",
  "name": "Basic Printer",
  "manufacturer": "The Manufacturer",
  "tags": [ "printer", "basic", "electronics", "business" ],
  "barcodes": [ "542364671", "886330670", "437445606" ],
```

```
  "reviews": [
    {
      "review_date": ISODate('2024-01-19T21:37:10.585Z'),
      ...
    }
  ],
  "material": "Polycarbonate",
  "color": "Space Gray",
  "supplier": {
    "supplierId": "SUP4",
    "location": {
      "type": "Point",
      "coordinates": [ -71.0589, 42.3601 ]
    }
  },
  "productEmbedding": [
    -0.019320633663838058,
    0.019672111388113596
  ],
  "lastUpdated": ISODate('2025-10-20T21:37:10.585Z')
}
```

Single field

```
db.collection.createIndex(
  {
    "productId": 1
  },
  {
    "name": "single_field_index"
  }
)
```

Compound

```
db.collection.createIndex(
  {
    "productId": 1,
    "manufacturer": 1
  },
  {
    "name": "compound_index"
  }
```

```
)
```

Multi-key

```
db.collection.createIndex(
  {
    "tags": 1
  },
  {
    "name": "multikey_index"
  }
)
```

Text

```
db.collection.createIndex(
  {
    "name": "text"
  },
  {
    "name": "text_index"
  }
)
```

Geospatial

```
db.collection.createIndex(
  {
    "supplier.location": "2dsphere"
  },
  {
    "name": "geospatial_index"
  }
)
```

Vector

```
db.runCommand({
  "createIndexes": "collection",
  "indexes": [{
    "key": {
```

```
      "productEmbedding": "vector"
    },
    "name": "hnsw_index",
    "vectorOptions": {
      "type": "hnsw",
      "dimensions": 2,
      "similarity": "euclidean",
      "m": 16,
      "efConstruction": 64
    }
  }]
})
```

Wildcard

```
db.collection.createIndex(
  {
    "supplier.$**": 1
  },
  {
    "name": "wildcard_index"
  }
)
```

Index Property: unique

Supported index types

Option	3.6	4.0	5.0	8.0	Elastic Cluster
single field	Yes	Yes	Yes	Yes	Yes
compound	Yes	Yes	Yes	Yes	Yes
multi-key	Yes	Yes	Yes	Yes	Yes
wildcard	No	No	No	No	No

Use the unique option to ensure uniqueness for the field(s) across documents in a collection.

Examples

The following examples show how to create unique indexes on the following sample document:

```
{
  "productId": "PROD133726",
  "sku": "SKU24224",
  "name": "Basic Printer",
  "manufacturer": "The Manufacturer",
  "tags": [ "printer", "basic", "electronics", "business" ],
  "barcodes": [ "542364671", "886330670", "437445606" ],
  "reviews": [
    {
      "review_date": ISODate('2024-01-19T21:37:10.585Z'),
      ...
    }
  ],
  "material": "Polycarbonate",
  "color": "Space Gray",
  "supplier": {
    "supplierId": "SUP4",
    "location": {
      "type": "Point",
      "coordinates": [ -71.0589, 42.3601 ]
    }
  },
  "productEmbedding": [
    -0.019320633663838058,
    0.019672111388113596
  ],
  "lastUpdated": ISODate('2025-10-20T21:37:10.585Z')
}
```

single field

Create a unique single field index on productId to ensure that the same productId does not exist in more than 1 document:

```
db.collection.createIndex(
  {
    "productId": 1
  },
  {
```

```
    name: "productId_unique",
    unique: true
  }
)
```

Compound

Create a unique compound index on sku and manufacturer to ensure that the same combination of sku and manufacturer does not exist in more than 1 document:

```
db.collection.createIndex(
  {
    "sku": 1,
    "manufacturer": 1
  },
  {
    name: "sku_and_manufacturer_unique",
    unique: true
  }
)
```

multi-key

Create a unique multi-key index on barcodes to ensure that any value in the barcodes array does not exist in more than 1 document:

```
db.collection.createIndex(
  {
    "barcodes": 1
  },
  {
    name: "barcodes_unique",
    unique: true
  }
)
```

Indexing an array creates an index entry for each element of the array. For example, if an array has 50 items, it has 50 index entries. Because of this, unique multi-key indexes enforce uniqueness across all individual items. For example, the following documents will violate the unique constraint on the values array field index:

```
{ "values": [ 1, 2, 3] }
```

```
{ "values": [ 3, 2 ] }   --> 3 and 2 already exist
{ "values": [ 1 ] }      --> 1 already exists
```

Note the following behavior with unique indexes:

1. If you create a unique index on existing data where two (or more) documents have the same values for the indexed fields, the index build will fail with the following error: `could not create unique index: <collection> index: <index name>`
2. If you insert a document where the value of the indexed field matches the value of that field in another document, the insert will fail with the following error: `E11000 duplicate key error collection: <collection> index: <index name>`
3. If you update an existing document so the new value of the indexed field matches the value of that field in another document, the update will fail with the following error: `E11000 duplicate key error collection: <collection> index: <index name>`
4. If the indexed field is missing from a document, the value will be treated as null. Index builds, inserts, and updates will fail as described above if the indexed field is missing from two (or more) documents.

Index Property: sparse

Supported index types

Option	3.6	4.0	5.0	8.0	Elastic Cluster
single field	Yes	Yes	Yes	Yes	Yes
compound	Yes	Yes	Yes	Yes	Yes
multi-key	Yes	Yes	Yes	Yes	Yes
wildcard	No	No	No	No	No

Use the sparse option to skip indexing documents missing the indexed field(s), reducing the index size and saving space in memory. Because the index size is smaller, the queries that use it are more efficient. For a query to utilize a sparse index, you must use the $exists clause on the indexed fields. If you omit the $exists clause, Amazon DocumentDB will not use the sparse index.

Examples

The following examples show how to create sparse indexes on the following sample document:

```
{
  "productId": "PROD133726",
  "sku": "SKU24224",
  "name": "Basic Printer",
  "manufacturer": "The Manufacturer",
  "tags": [ "printer", "basic", "electronics", "business" ],
  "barcodes": [ "542364671", "886330670", "437445606" ],
  "reviews": [
    {
      "review_date": ISODate('2024-01-19T21:37:10.585Z'),
      ...
    }
  ],
  "material": "Polycarbonate",
  "color": "Space Gray",
  "supplier": {
    "supplierId": "SUP4",
    "location": {
      "type": "Point",
      "coordinates": [ -71.0589, 42.3601 ]
    }
  },
  "productEmbedding": [
    -0.019320633663838058,
    0.019672111388113596
  ],
  "lastUpdated": ISODate('2025-10-20T21:37:10.585Z')
}
```

Note that the reviews, material, and color fields do not exist in all documents.

single field

Create a sparse single field index on the material field:

```
db.collection.createIndex(
  {
    "material": 1
  },
```

```
  {
    "name": "material_sparse",
    "sparse": true
  }
)
```

This index will be used when finding all products that have a value for material:

```
db.collection.find({
  "material": {
    $exists: true
  }
})
```

compound

Create a sparse compound index on the material and color fields:

```
db.collection.createIndex(
  {
    "material": 1,
    "color": 1
  },
  {
    "name": "material_and_color_sparse",
    "sparse": true
  }
)
```

This index will be used when finding all products that have any combination of material and color values:

```
db.collection.find({
  "material": {
    $exists: true
  }
})

db.collection.find({
  "color": {
    $exists: true
  }
```

```
})

db.collection.find({
  $and: [
    {
      "material": {
        $exists: true
      }
    },
    {
      "color": {
        $exists:true
      }
    }
  ]
})

db.collection.find({
  $and: [
    {
      "color": {
        $exists: true
      }
    },
    {
      "material": {
        $exists:true
      }
    }
  ]
})
```

multi-key

Create a sparse multi-key index on the reviews array:

```
db.collection.createIndex(
  {
    "reviews": 1
  },
  {
    "name": "reviews_sparse",
    "sparse": true
  }
```

```
)
```

This index will be used when finding all products that have a reviews array:

```
db.collection.find({
  "reviews": {
    $exists: true
  }
})
```

Index Property: partialFilterExpression

Supported index types

Option	3.6	4.0	5.0	8.0	Elastic Cluster
single field	No	No	Yes	Yes	No
compound	No	No	Yes	Yes	No
multi-key	No	No	Yes	Yes	No
geospatial	No	No	No	No	No
wildcard	No	No	No	Yes	No

Use the partialFilterExpression option to create a partial index that only includes documents that meet a specified filter condition. This allows you to create more efficient indexes by indexing only a subset of documents in a collection, rather than indexing all documents, reducing the index size and saving space in memory. Because the index size is smaller, the queries that use it are more efficient. Amazon DocumentDB will use the partial index in the following scenarios:

- The query predicate exactly matches the partial index filter expression.
- The query filter's expected result is a logical subset of the partial filter.
- A sub-predicate of the query can be used in conjunction with other indexes.

For more information, see Partial index.

Examples

The following examples show how to create partial indexes on the following sample document:

```
{
  "productId": "PROD133726",
  "sku": "SKU24224",
  "name": "Basic Printer",
  "manufacturer": "The Manufacturer",
  "tags": [ "printer", "basic", "electronics", "business" ],
  "barcodes": [ "542364671", "886330670", "437445606" ],
  "reviews": [
    {
      "review_date": ISODate('2024-01-19T21:37:10.585Z'),
      ...
    }
  ],
  "material": "Polycarbonate",
  "color": "Space Gray",
  "supplier": {
    "supplierId": "SUP4",
    "location": {
      "type": "Point",
      "coordinates": [ -71.0589, 42.3601 ]
    }
  },
  "productEmbedding": [
    -0.019320633663838058,
    0.019672111388113596
  ],
  "lastUpdated": ISODate('2025-10-20T21:37:10.585Z')
}
```

single field

Create a partial single field index on manufacturer for products that have a color of Space Gray:

```
db.collection.createIndex(
  {
    "manufacturer": 1
  },
  {
    "name": "manufacturer_space_gray",
```

```
    "partialFilterExpression": {
      "color": {
        $eq: "Space Gray"
      }
    }
  }
)
```

This index will be used when finding Space Gray color products by manufacturer:

```
db.collection.find({
  "manufacturer": "The Manufacturer",
  "color": {
    $eq: "Space Gray"
  }
})
```

compound

Create a partial compound index on manufacturer and color for products that have a material of Polycarbonate:

```
db.collection.createIndex(
  {
    "manufacturer": 1,
    "color": 1
  },
  {
    "name": "manufacturer_and_color_polycarbonate",
    "partialFilterExpression": {
      "material": {
        $eq: "Polycarbonate"
      }
    }
  }
)
```

This index will be used when finding Polycarbonate material products by manufacturer and color:

```
db.collection.find({
  "manufacturer": "The Manufacturer",
  "color": "Space Gray",
```

```
  "material": {
    $eq: "Polycarbonate"
  }
})
```

multi-key

Create a partial multi-key index on tags for products that have a manufacturer of The Manufacturer:

```
db.collection.createIndex(
  {
    "tags": 1
  },
  {
    "name": "tags_the_manufacturer",
    "partialFilterExpression": {
      "manufacturer": {
        $eq: "The Manufacturer"
      }
    }
  }
)
```

This index will be used when finding products tagged with printer that have a manufacturer of The Manufacturer:

```
db.collection.find({
  "tags": "printer",
  "manufacturer": {
    $eq: "The Manufacturer"
  }
})
```

wildcard

Create a partial wildcard index on all fields under supplier for products that have a manufacturer of The Manufacturer:

```
db.collection.createIndex(
  {
```

```
    "supplier.$**": 1
  },
  {
    "name": "supplier_the_manufacturer",
    "partialFilterExpression": {
      "manufacturer": {
        $eq: "The Manufacturer"
      }
    }
  }
)
```

This index will be used when finding products by supplier ID that have a manufacturer of The Manufacturer:

```
db.collection.find({
  "supplier.supplierId": "SUP4",
  "manufacturer": {
    $eq: "The Manufacturer"
  }
})
```

Index Property: expireAfterSeconds

Supported index types

Index type	3.6	4.0	5.0	8.0	Elastic Cluster
single field	Yes	Yes	Yes	Yes	Yes
multi-key	Yes	Yes	Yes	Yes	Yes
wildcard	No	No	No	No	No

Use the expireAfterSeconds option to create a time to live (TTL) index. TTL indexes enable you to delete documents based on their age based on the timeout condition for each document. When a document reaches the specified TTL age, it is deleted from the collection.

For best practices with TTL deletes, see TTL and time series workloads.

Examples

The following examples show how to create TTL indexes on the following sample document:

```
{
  "productId": "PROD133726",
  "sku": "SKU24224",
  "name": "Basic Printer",
  "manufacturer": "The Manufacturer",
  "tags": [ "printer", "basic", "electronics", "business" ],
  "barcodes": [ "542364671", "886330670", "437445606" ],
  "reviews": [
    {
      "review_date": ISODate('2024-01-19T21:37:10.585Z'),
      "rating": 4,
      "comment": "Good product"
    },
    {
      "review_date": ISODate('2024-03-15T14:22:33.120Z'),
      "rating": 5,
      "comment": "Excellent printer"
    },
    {
      "review_date": ISODate('2024-06-08T09:45:18.890Z'),
      "rating": 3,
      "comment": "Average quality"
    }
  ],
  "material": "Polycarbonate",
  "color": "Space Gray",
  "supplier": {
    "supplierId": "SUP4",
    "location": {
      "type": "Point",
      "coordinates": [ -71.0589, 42.3601 ]
    }
  },
  "productEmbedding": [
    -0.019320633663838058,
    0.019672111388113596
  ],
  "lastUpdated": ISODate('2025-10-20T21:37:10.585Z')
}
```

single field

Create a TTL index on lastUpdated to delete documents that have not been updated in 90 days:

```
db.collection.createIndex(
  {
    "lastUpdated": 1
  },
  {
    "name": "lastUpdated_ttl",
    "expireAfterSeconds": 7776000
  }
)
```

multi-key

Create a TTL index on reviews.review_date to delete documents that have had no reviews in the past year:

```
db.collection.createIndex(
  {
    "reviews.review_date": 1
  },
  {
    "name": "reviews_review_date_ttl",
    "expireAfterSeconds": 31536000
  }
)
```

Note that if the TTL index is on an array field, all items in the array are checked. If any timestamp in the array satisfies the timeout condition, the document will be deleted.

Index property: hidden

Amazon DocumentDB supports the `hidden` index property beginning with Amazon DocumentDB 8.0.1+ and Amazon DocumentDB 5.0.2+.

Hidden indexes are not visible to the query planner and cannot be used to support a query. By hiding an index from the planner, you can evaluate the potential impact of dropping the index without actually dropping it. If the impact is negative, you can unhide the index instead of having to recreate a dropped index.

The `hidden` property is supported on all index types, including single field, compound, multikey, wildcard, text, geospatial, and vector indexes.

Behavior

Apart from being hidden from the planner, hidden indexes are fully maintained and enforced. For example:

- If a hidden index is a unique index, the index still applies its unique constraint to documents.
- If a hidden index is a TTL index, the index still expires documents.
- Hidden indexes are included in `listIndexes` and `db.collection.getIndexes()` results.
- Hidden indexes are updated on write operations to the collection and continue to consume disk space and memory. As a result, they are included in statistics operations such as `db.collection.stats()` and `$indexStats`.
- Hiding an unhidden index, or unhiding a hidden index, resets its `$indexStats`. Hiding an already hidden index, or unhiding an already unhidden index, does not reset `$indexStats`.

Restrictions

- You cannot hide the `_id` index.
- You cannot use `cursor.hint()` to force the use of a hidden index.
- `$merge` fails if the required unique index on the on fields is hidden. Make the index visible before running `$merge`.

Examples

Create a hidden index

To create an index in the hidden state, use the `db.collection.createIndex()` method and set the `hidden` option to `true` in the index definition.

For example, the following operation creates a hidden ascending index on the `productId` field:

```
db.collection.createIndex(
  { "productId": 1 },
  { "name": "productId_1", "hidden": true }
)
```

To verify, run `db.collection.getIndexes()` or `listIndexes` on the collection. A hidden index includes `"hidden": true` in its definition.

```
db.collection.getIndexes()
[
  {
    "v": 4,
    "key": { "_id": 1 },
    "name": "_id_"
  },
  {
    "v": 4,
    "key": { "productId": 1 },
    "name": "productId_1",
    "hidden": true
  }
]
```

The index option `hidden` is only returned if the value is `true`.

Hide an existing index

Use the `collMod` command to hide an existing index. Identify the index either by its `name` or by its `keyPattern`, and set `hidden` to `true`.

The following example hides an index by name.

```
db.runCommand({
  collMod: "collection",
  index: {
    name: "productId_1", // Specify the index name
    hidden: true
  }
})
```

You can also identify the index by its key pattern.

```
db.runCommand({
  collMod: "collection",
  index: {
    keyPattern: { productId: 1 }, // Specify the index key specification document
```

```
    hidden: true
  }
})
```

Unhide an existing index

To make a hidden index visible again, use the `collMod` command with `hidden` set to `false`.

```
db.runCommand({
  collMod: "collection",
  index: { name: "productId_1", hidden: false }
})
```

Because indexes are fully maintained while hidden, the index is immediately available for use once unhidden.

Using change streams with Amazon DocumentDB

The change streams feature in Amazon DocumentDB (with MongoDB compatibility) provides a time-ordered sequence of change events that occur within your cluster's collections. You can read events from a change stream to implement many different use cases, including the following:

- Change notification
- Full-text search with Amazon OpenSearch Service (OpenSearch Service)
- Analytics with Amazon Redshift

Applications can use change streams to subscribe to data changes on individual collections. Change streams events are ordered as they occur on the cluster and are stored for 3 hours (by default) after the event has been recorded. The retention period can be extended up to 7 days using the `change_stream_log_retention_duration` parameter. To modify the change stream retention period, see Modifying the change stream log retention duration.

Topics

- Supported operations
- Expanded events
- Billing

- Limitations
- Enabling change streams
- Example: using change streams with Python
- Full document lookup
- Using $changeStreamSplitLargeEvent to split large events into fragments
- Resuming a change stream
- Resuming a change stream with startAtOperationTime
- Resuming a change stream with postBatchResumeToken
- Transactions in change streams
- Modifying the change stream log retention duration
- Using change streams on secondary instances

Supported operations

Amazon DocumentDB supports the following operations for change streams:

- All change events supported in the MongoDB `db.collection.watch()`, `db.watch()` and `client.watch()` API.
- Full document lookup for updates.
- Aggregation stages: `$match`, `$project`, `$redact`, `$addFields`, `$replaceRoot`, and `$changeStreamSplitLargeEvent`.
- Resuming a change stream from a resume token
- Resuming a change stream from a timestamp using `startAtOperation` (applicable to Amazon DocumentDB 4.0+)
- DDL event notification for `createCollection` using the `showExpandedEvents` parameter (applicable to Amazon DocumentDB 8.0 starting with engine version 8.0.2).
- DDL event notification for `createIndexes` using the `showExpandedEvents` parameter (applicable to Amazon DocumentDB 8.0 starting with engine version 8.0.2).
- DDL event notification for `dropIndexes` using the `showExpandedEvents` parameter (applicable to Amazon DocumentDB 8.0 starting with engine version 8.0.3).

Expanded events

> **Note**
>
> Expanded events are available in Amazon DocumentDB 8.0 starting with engine version 8.0.2.

Change streams support data definition language (DDL) event notifications for `createCollection` and `createIndexes` starting with engine version 8.0.2, and `dropIndexes` starting with engine version 8.0.3. To include expanded events, open a change stream cursor with the `showExpandedEvents` option.

For example:

mongosh Method

```
let cur = db.names.watch( [ ], {
   showExpandedEvents: true
} )
cur.next()
```

Aggregation Stage

```
let cur = db.names.aggregate( [
   { $changeStream: { showExpandedEvents: true } }
] )
cur.next()
```

Billing

The Amazon DocumentDB change streams feature is disabled by default and does not incur any additional charges until the feature is enabled. Using change streams in a cluster incurs additional read and write IOs and storage costs. You can use the `modifyChangeStreams` API operation to enable this feature for your cluster. For more information on pricing, see Amazon DocumentDB pricing.

Limitations

Change streams have the following limitations in Amazon DocumentDB:

- On Amazon DocumentDB 3.6. and Amazon DocumentDB 4.0, change streams can only be opened from a connection to the primary instance of an Amazon DocumentDB cluster. Reading from change streams on a replica instance is not supported on Amazon DocumentDB 3.6. and Amazon DocumentDB 4.0. When invoking the `watch()` API operation, you must specify a `primary` read preference to ensure that all reads are directed to the primary instance (see the Example section).
- On Amazon DocumentDB 5.0, change streams can be opened from both primary instance and secondary instances, including global clusters. You can specify a secondary read preference to redirect the change streams to secondary instances. See Using change streams on secondary instances for additional best practices and limitations.
- Events written to a change stream for a collection are available for up to 7 days (the default is 3 hours). Change streams data is deleted after the log retention duration window, even if no new changes have occurred.
- A long-running write operation on a collection like `updateMany` or `deleteMany` can temporarily stall the writing of change streams events until the long running write operation is complete.
- Amazon DocumentDB does not support the MongoDB operations log (`oplog`).
- With Amazon DocumentDB, you must explicitly enable change streams on a given collection.
- If the total size of a change streams event (including the change data and full document, if requested) is greater than `16 MB`, the client will experience a read failure on the change stream. To avoid this, use the `$changeStreamSplitLargeEvent` stage to split the event into fragments.
- The Ruby driver is currently not supported when using `db.watch()` and `client.watch()` with Amazon DocumentDB 3.6.
- The output from the `updateDescription` command in change streams is different in Amazon DocumentDB than in MongoDB when the updated value of the field is the same as the previous one:
  - Amazon DocumentDB doesn't return a field in the `updateDescription` output if the provided field is specified in the `$set` command and its target value is already equal to the source value.
  - MongoDB returns the field in the output, even if the specified value is equal to the current value.

Enabling change streams

You can enable Amazon DocumentDB change streams for all collections within a given database, or only for selected collections. The following are examples of how to enable change streams for different use cases using the mongo shell. Empty strings are treated as wildcards when specifying database and collection names.

```
//Enable change streams for the collection "foo" in database "bar"
db.adminCommand({modifyChangeStreams: 1,
    database: "bar",
    collection: "foo",
    enable: true});
```

```
//Disable change streams on collection "foo" in database "bar"
db.adminCommand({modifyChangeStreams: 1,
    database: "bar",
    collection: "foo",
    enable: false});
```

```
//Enable change streams for all collections in database "bar"
db.adminCommand({modifyChangeStreams: 1,
    database: "bar",
    collection: "",
    enable: true});
```

```
//Enable change streams for all collections in all databases in a cluster
db.adminCommand({modifyChangeStreams: 1,
    database: "",
    collection: "",
    enable: true});
```

Change streams will be enabled for a collection if any of the following are true:

- Both the database and collection are explicitly enabled.
- The database containing the collection is enabled.
- All databases are enabled.

Dropping a collection from a database does not disable change streams for that collection if the parent database also has change streams enabled, or if all databases in the cluster are enabled. If

a new collection is created with the same name as the deleted collection, change streams will be enabled for that collection.

You can list all of your cluster's enabled change streams by using the `$listChangeStreams` aggregation pipeline stage. All aggregation stages supported by Amazon DocumentDB can be used in the pipeline for additional processing. If a previously enabled collection has been disabled, it will not appear in the `$listChangeStreams` output.

```
//List all databases and collections with change streams enabled
cursor = new DBCommandCursor(db,
    db.runCommand(
        {aggregate: 1,
        pipeline: [{$listChangeStreams: 1}],
        cursor:{}}));
```

```
//List of all databases and collections with change streams enabled
{ "database" : "test", "collection" : "foo" }
{ "database" : "bar", "collection" : "" }
{ "database" : "", "collection" : "" }
```

```
//Determine if the database “bar” or collection “bar.foo” have change streams enabled
cursor = new DBCommandCursor(db,
  db.runCommand(
      {aggregate: 1,
       pipeline: [{$listChangeStreams: 1},
                  {$match: {$or: [{database: "bar", collection: "foo"},
                                  {database: "bar", collection: ""},
                                  {database: "", collection: ""}]}}
                ],
      cursor:{}}));
```

Example: using change streams with Python

The following is an example of using an Amazon DocumentDB change stream with Python at the collection level.

```
import os
import sys
from pymongo import MongoClient, ReadPreference

username = "DocumentDBusername"
```

```
password = <Insert your password>

clusterendpoint = "DocumentDBClusterEndpoint”
client = MongoClient(clusterendpoint, username=username, password=password, tls='true',
 tlsCAFile='global-bundle.pem')

db = client['bar']

#While ‘Primary’ is the default read preference, here we give an example of
#how to specify the required read preference when reading the change streams
coll = db.get_collection('foo', read_preference=ReadPreference.PRIMARY)
#Create a stream object
stream = coll.watch()
#Write a new document to the collection to generate a change event
coll.insert_one({'x': 1})
#Read the next change event from the stream (if any)
print(stream.try_next())

"""
Expected Output:
{'_id': {'_data': '015daf94f600000002010000000200009025'},
'clusterTime': Timestamp(1571788022, 2),
'documentKey': {'_id': ObjectId('5daf94f6ea258751778163d6')},
'fullDocument': {'_id': ObjectId('5daf94f6ea258751778163d6'), 'x': 1},
'ns': {'coll': 'foo', 'db': 'bar'},
'operationType': 'insert'}
"""

#A subsequent attempt to read the next change event returns nothing, as there are no
 new changes
print(stream.try_next())

"""
Expected Output:
None
"""

#Generate a new change event by updating a document
result = coll.update_one({'x': 1}, {'$set': {'x': 2}})
print(stream.try_next())

"""
Expected Output:
{'_id': {'_data': '015daf99d400000001010000000100009025'},
```

```
'clusterTime': Timestamp(1571789268, 1),
'documentKey': {'_id': ObjectId('5daf9502ea258751778163d7')},
'ns': {'coll': 'foo', 'db': 'bar'},
'operationType': 'update',
'updateDescription': {'removedFields': [], 'updatedFields': {'x': 2}}}
"""
```

The following is an example of using an Amazon DocumentDB change stream with Python at the database level.

```
import os
import sys
from pymongo import MongoClient

username = "DocumentDBusername"
password = <Insert your password>
clusterendpoint = "DocumentDBClusterEndpoint”
client = MongoClient(clusterendpoint, username=username, password=password, tls='true',
 tlsCAFile='global-bundle.pem')

db = client['bar']
#Create a stream object
stream = db.watch()
coll = db.get_collection('foo')
#Write a new document to the collection foo to generate a change event
coll.insert_one({'x': 1})

#Read the next change event from the stream (if any)
print(stream.try_next())

"""
Expected Output:
{'_id': {'_data': '015daf94f600000002010000000200009025'},
'clusterTime': Timestamp(1571788022, 2),
'documentKey': {'_id': ObjectId('5daf94f6ea258751778163d6')},
'fullDocument': {'_id': ObjectId('5daf94f6ea258751778163d6'), 'x': 1},
'ns': {'coll': 'foo', 'db': 'bar'},
'operationType': 'insert'}
"""
#A subsequent attempt to read the next change event returns nothing, as there are no
 new changes
print(stream.try_next())
```

```
"""
Expected Output:
None
"""

coll = db.get_collection('foo1')

#Write a new document to another collection to generate a change event
coll.insert_one({'x': 1})
print(stream.try_next())

"""
Expected Output: Since the change stream cursor was the database level you can see
 change events from different collections in the same database
{'_id': {'_data': '015daf94f600000002010000000200009025'},
'clusterTime': Timestamp(1571788022, 2),
'documentKey': {'_id': ObjectId('5daf94f6ea258751778163d6')},
'fullDocument': {'_id': ObjectId('5daf94f6ea258751778163d6'), 'x': 1},
'ns': {'coll': 'foo1', 'db': 'bar'},
'operationType': 'insert'}
"""
```

Full document lookup

The update change event does not include the full document; it includes only the change that was made. If your use case requires the complete document affected by an update, you can enable full document lookup when opening the stream.

The `fullDocument` document for an update change streams event represents the most current version of the updated document at the time of document lookup. If changes occurred between the update operation and the `fullDocument` lookup, the `fullDocument` document might not represent the document state at update time.

To create a stream object with update lookup enabled, use this example:

```
stream = coll.watch(full_document='updateLookup')

#Generate a new change event by updating a document
result = coll.update_one({'x': 2}, {'$set': {'x': 3}})

stream.try_next()
```

The output of the stream object will look something like this:

```
{'_id': {'_data': '015daf9b7c000000010100000100009025'},
'clusterTime': Timestamp(1571789692, 1),
'documentKey': {'_id': ObjectId('5daf9502ea258751778163d7')},
'fullDocument': {'_id': ObjectId('5daf9502ea258751778163d7'), 'x': 3},
'ns': {'coll': 'foo', 'db': 'bar'},
'operationType': 'update',
'updateDescription': {'removedFields': [], 'updatedFields': {'x': 3}}}
```

Using $changeStreamSplitLargeEvent to split large events into fragments

> **Note**
>
> The $changeStreamSplitLargeEvent stage is available in Amazon DocumentDB 8.0 starting with engine version 8.0.2.

When you use the `$changeStreamSplitLargeEvent` stage, change events larger than 16 MB are split into fragments that are returned sequentially through the change stream cursor. Events smaller than 16 MB are returned unsplit. The stage splits an event along its top-level field boundaries; it does not split within a single field or recurse into subdocuments. Because of this, if any single top-level field is itself larger than 16 MB, the stage can't produce a valid fragment for it and returns an error instead.

As a best practice, we recommend avoiding the use of `$changeStreamSplitLargeEvent` when possible by reducing the size of the event.

The `$changeStreamSplitLargeEvent` stage can only be used in a `$changeStream` pipeline, and it must be used only once as the last stage.

Each fragment has its own `resumeToken` and its own `postBatchResumeToken`. Resuming from a non-final fragment token (using either a regular resume token or a postBatchResumeToken) delivers the subsequent fragment. In the case where the token corresponds to the final fragment of an event, the change stream cursor returns the subsequent event.

Resuming a change stream

You can resume a change stream later by using a resume token, which is equal to the `_id` field of the last retrieved change event document.

```
import os
import sys
from pymongo import MongoClient

username = "DocumentDBusername"
password = <Insert your password>
clusterendpoint = "DocumentDBClusterEndpoint”
client = MongoClient(clusterendpoint, username=username, password=password, tls='true',
 tlsCAFile='global-bundle.pem', retryWrites='false')

db = client['bar']
coll = db.get_collection('foo')
#Create a stream object
stream = db.watch()
coll.update_one({'x': 1}, {'$set': {'x': 4}})
event = stream.try_next()
token = event['_id']
print(token)

"""
Output: This is the resume token that we will later us to resume the change stream
{'_data': '015daf9c5b0000000101000000100009025'}
"""
#Python provides a nice shortcut for getting a stream’s resume token
print(stream.resume_token)

"""
Output
{'_data': '015daf9c5b0000000101000000100009025'}
"""
#Generate a new change event by updating a document
result = coll.update_one({'x': 4}, {'$set': {'x': 5}})
#Generate another change event by inserting a document
result = coll.insert_one({'y': 5})
#Open a stream starting after the selected resume token
stream = db.watch(full_document='updateLookup', resume_after=token)
#Our first change event is the update with the specified _id
print(stream.try_next())
```

```
"""
#Output: Since we are resuming the change stream from the resume token, we will see all
 events after the first update operation. In our case, the change stream will resume
 from the update operation {x:5}

{'_id': {'_data': '015f7e8f0c0000000601000000006000fe038'},
'operationType': 'update',
'clusterTime': Timestamp(1602129676, 6),
'ns': {'db': 'bar', 'coll': 'foo'},
'documentKey': {'_id': ObjectId('5f7e8f0ac423bafbfd9adba2')},
'fullDocument': {'_id': ObjectId('5f7e8f0ac423bafbfd9adba2'), 'x': 5},
'updateDescription': {'updatedFields': {'x': 5}, 'removedFields': []}}
"""
#Followed by the insert
print(stream.try_next())

"""
#Output:
{'_id': {'_data': '015f7e8f0c0000000701000000007000fe038'},
'operationType': 'insert',
'clusterTime': Timestamp(1602129676, 7),
'ns': {'db': 'bar', 'coll': 'foo'},
'documentKey': {'_id': ObjectId('5f7e8f0cbf8c233ed577eb94')},
'fullDocument': {'_id': ObjectId('5f7e8f0cbf8c233ed577eb94'), 'y': 5}}
"""
```

Resuming a change stream with `startAtOperationTime`

You can resume a change stream later from a particular time stamp by using `startAtOperationTime`.

> **Note**
>
> The ability to use `startAtOperationTime` is available in Amazon DocumentDB 4.0+. When using `startAtOperationTime`, the change stream cursor will only return changes that occurred at or after the specified Timestamp. The `startAtOperationTime` and `resumeAfter` commands are mutually exclusive and thus cannot be used together.

```
import os
```

```
import sys
from pymongo import MongoClient

username = "DocumentDBusername"
password = <Insert your password>
clusterendpoint = "DocumentDBClusterEndpoint”
client = MongoClient(clusterendpoint, username=username, password=password, tls='true',
 tlsCAFile='rds-root-ca-2020.pem',retryWrites='false')
db = client['bar']
coll = db.get_collection('foo')
#Create a stream object
stream = db.watch()
coll.update_one({'x': 1}, {'$set': {'x': 4}})
event = stream.try_next()
timestamp = event['clusterTime']
print(timestamp)
"""
Output
Timestamp(1602129114, 4)
"""
#Generate a new change event by updating a document
result = coll.update_one({'x': 4}, {'$set': {'x': 5}})
result = coll.insert_one({'y': 5})
#Generate another change event by inserting a document
#Open a stream starting after specified time stamp

stream = db.watch(start_at_operation_time=timestamp)
print(stream.try_next())

"""
#Output: Since we are resuming the change stream at the time stamp of our first update
 operation (x:4), the change stream cursor will point to that event
{'_id': {'_data': '015f7e941a0000000301000000003000fe038'},
'operationType': 'update',
'clusterTime': Timestamp(1602130970, 3),
'ns': {'db': 'bar', 'coll': 'foo'},
'documentKey': {'_id': ObjectId('5f7e9417c423bafbfd9adbb1')},
'updateDescription': {'updatedFields': {'x': 4}, 'removedFields': []}}
"""

print(stream.try_next())
"""
#Output: The second event will be the subsequent update operation (x:5)
{'_id': {'_data': '015f7e9502000000050100000005000fe038'},
```

```
'operationType': 'update',
'clusterTime': Timestamp(1602131202, 5),
'ns': {'db': 'bar', 'coll': 'foo'},
'documentKey': {'_id': ObjectId('5f7e94ffc423bafbfd9adbb2')},
'updateDescription': {'updatedFields': {'x': 5}, 'removedFields': []}}
"""

print(stream.try_next())

"""
#Output: And finally the last event will be the insert operation (y:5)
{'_id': {'_data': '015f7e9502000000060100000006000fe038'},
'operationType': 'insert',
'clusterTime': Timestamp(1602131202, 6),
'ns': {'db': 'bar', 'coll': 'foo'},
'documentKey': {'_id': ObjectId('5f7e95025c4a569e0f6dde92')},
'fullDocument': {'_id': ObjectId('5f7e95025c4a569e0f6dde92'), 'y': 5}}
"""
```

Resuming a change stream with `postBatchResumeToken`

Amazon DocumentDB change stream now returns an additional field called `postBatchResumeToken`. This field is returned from the `$changestream` command and `getMore` command.

Example of the `$changestream` command in Python:

```
db.command({"aggregate": "sales", "pipeline": [{ "$changeStream": {}}], "cursor":
 {"batchSize": 1}})
```

Expected output:

```
cursor" : {
   "firstBatch" : [ ],
   "postBatchResumeToken" : {"_data" : "0167c8cbe60000000004"},
   "id" : NumberLong("9660788144470"),
   "ns" : "test.sales"
}
```

Example of the `getMore` command in Python:

```
db.command({"getMore": NumberLong(<cursor id>), "collection": "sales", "batchSize":
 1 })
```

Expected output

```
cursor" : {
   "nextBatch" : [ ],
   "postBatchResumeToken" : {"_data" : "0167c8cbe60000000004"},
   "id" : NumberLong("9660788144470"),
   "ns" : "test.sales"
}
```

The `postBatchResumeToken` field can be used to open new change stream cursors in the `resumeAfter` field, similar to how the resume token is used.

Open a stream starting after the selected `postBatchResumeToken`:

```
post_batch_resume_token = output['cursor']['postBatchResumeToken']
stream = db.watch(full_document='updateLookup', resume_after=post_batch_resume_token)
```

Unlike a regular resume token that always corresponds to an operations log (oplog) entry that reflects an actual event, `postBatchResumeToken` corresponds to an oplog entry the change stream has scanned up to on the server, which is not necessarily a matching change.

Attempting to resume with an old regular resume token will force the database to scan all the oplog entries between the specified time stamp and the current time. This may generate a lot of queries internally with each sub-query scanning for a small period of time. This will cause a spike in CPU usage and degrade the database performance. Resuming with the last `postBatchResumeToken` skips the scanning of unmatched oplog entries.

Transactions in change streams

Change stream events will not contain events from uncommitted and/or aborted transactions. For example, if you start a transaction with one `INSERT` operation and one `UPDATE` operation, and if your `INSERT` operation succeeds, but the `UPDATE` operation fails, the transaction will be rolled back. Since this transaction was rolled back, your change stream will not contain any events for this transaction.

Modifying the change stream log retention duration

You can modify the change stream log retention duration to be between 1 hour and 7 days using the AWS Management Console or the AWS CLI.

Using the AWS Management Console

To modify the change stream log retention duration

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups** .

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Parameter groups** pane, choose the cluster parameter group that is associated with your cluster. To identify the cluster parameter group that is associated with your cluster, see Determining an Amazon DocumentDB cluster's parameter group.
4. The resulting page shows the parameters and their corresponding details for your cluster parameter group. Select the parameter `change_stream_log_retention_duration`.
5. On the top right of the page, choose **Edit** to change the value of the parameter. The `change_stream_log_retention_duration` parameter can be modified to be between 1 hour and 7 days.
6. Make your change, and then choose **Modify cluster parameter** to save the changes. To discard your changes, choose **Cancel**.

Using the AWS CLI

To modify your cluster parameter group's `change_stream_log_retention_duration` parameter, use the `modify-db-cluster-parameter-group` operation with the following parameters:

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group that you are modifying. To identify the cluster parameter group that is associated with your cluster, see Determining an Amazon DocumentDB cluster's parameter group.
- **`--parameters`** — Required. The parameter that you are modifying. Each parameter entry must include the following:
  - **`ParameterName`** — The name of the parameter that you are modifying. In this case, it is `change_stream_log_retention_duration`
  - **`ParameterValue`** — The new value for this parameter.
  - **`ApplyMethod`** — How you want changes to this parameter applied. Permitted values are `immediate` and `pending-reboot`.

    > **Note**
    >
    > Parameters with the `ApplyType` of `static` must have an `ApplyMethod` of `pending-reboot`.

1. To change the values of the parameter `change_stream_log_retention_duration`, run the following command and replace `parameter-value` with the value you want to modify the parameter to.

   For Linux, macOS, or Unix:

   ```
   aws docdb modify-db-cluster-parameter-group \
       --db-cluster-parameter-group-name sample-parameter-group \
       --parameters
    "ParameterName=change_stream_log_retention_duration,ParameterValue=<parameter-
   value>,ApplyMethod=immediate"
   ```

   For Windows:

   ```
   aws docdb modify-db-cluster-parameter-group ^
       --db-cluster-parameter-group-name sample-parameter-group ^
       --parameters
    "ParameterName=change_stream_log_retention_duration,ParameterValue=<parameter-
   value>,ApplyMethod=immediate"
   ```

   Output from this operation looks something like the following (JSON format).

```
{
    "DBClusterParameterGroupName": "sample-parameter-group"
}
```

2. Wait at least 5 minutes.
3. List the parameter values of `sample-parameter-group` to ensure that your changes have been made.

   For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
    --db-cluster-parameter-group-name sample-parameter-group
```

   For Windows:

```
aws docdb describe-db-cluster-parameters ^
    --db-cluster-parameter-group-name sample-parameter-group
```

   Output from this operation looks something like the following (JSON format).

```
{
    "Parameters": [
        {
            "ParameterName": "audit_logs",
            "ParameterValue": "disabled",
            "Description": "Enables auditing on cluster.",
            "Source": "system",
            "ApplyType": "dynamic",
            "DataType": "string",
            "AllowedValues": "enabled,disabled",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot"
        },
        {
            "ParameterName": "change_stream_log_retention_duration",
            "ParameterValue": "12345",
            "Description": "Duration of time in seconds that the change stream
 log is retained and can be consumed.",
            "Source": "user",
            "ApplyType": "dynamic",
```

```
            "DataType": "integer",
            "AllowedValues": "3600-86400",
            "IsModifiable": true,
            "ApplyMethod": "immediate"
        }
    ]
}
```

Note

Change stream log retention will not delete logs older than the configured `change_stream_log_retention_duration` value until log size is greater than (>) 51,200MB.

Using change streams on secondary instances

To get started on using change stream on secondary instances, open the change stream cursor with `readPreference` as the secondary.

You can open a change stream cursor to watch for change events on a specific collection or all collections in a cluster or database. You can open a change stream cursor on any Amazon DocumentDB instance and fetch change stream documents from both writer and reader instances. You can share change stream tokens (such as `resumeToken` or `startOperationTime`) across different change stream cursors opened on a writer and reader instance.

Example

```
import os
import sys
from pymongo import MongoClient, ReadPreference

username = "DocumentDBusername"
password = <Your password>

clusterendpoint = "DocumentDBClusterEndpoint"

client = MongoClient(clusterendpoint, username=username, password=password, tls='true',
 tlsCAFile='global-bundle.pem')
```

```
db = client['bar']

# Make sure to use SECONDARY to redirect cursor reads from secondary instances
coll = db.get_collection('foo', read_preference=ReadPreference.SECONDARY)

# Create a stream object on RO. The token needs to generated from PRIMARY.
stream = coll.watch(resumeAfter=token)

for event in stream:
   print(event)
```

Guidelines and limitations for change streams on secondary instances

- Change stream events need to be replicated from the primary instance to the secondary instances. You can monitor the lag from the `DBInstanceReplicaLag` metric in Amazon CloudWatch.
- Timestamps on secondary instances may not always be in sync with the primary instance. In this case, expect delays on the secondary instance timestamp so it can catch up. As a best practice, use `startAtOperationTime` or `resumeToken` to start the watch on the secondary instance.
- You might experience lower throughput on secondary instances compared to the primary instance if your document size is large, you are doing `fullDocumentLookup`, and there is high concurrent write workload on the primary instance. As a best practice, monitor your buffer cache hit ratio on the secondary and make sure that buffer cache hit ratio is high.

Transactions in Amazon DocumentDB

Amazon DocumentDB (with MongoDB compatibility) supports transactions in Amazon DocumentDB 4.0 and later. You can perform transactions across multiple documents, statements, collections, and databases. Transactions simplify application development by enabling you to perform atomic, consistent, isolated, and durable (ACID) operations across one or more documents within an Amazon DocumentDB cluster. Common use cases for transactions include financial processing, fulfilling and managing orders, and building multi-player games.

There is no additional cost for transactions. You only pay for the read and write IOs that you consume as part of the transactions.

Topics

- Requirements

- Best practices
- Limitations
- Monitoring and diagnostics
- Transaction isolation level
- Use cases
- Supported commands
- Unsupported capabilities
- Sessions
- Transaction errors

Requirements

To use the transactions feature, you need to meet the following requirements:

- You must use Amazon DocumentDB 4.0 or later.
- You must use a driver compatible with MongoDB 4.0 or later.

Best practices

Here are some best practices so that you can get the most using transactions with Amazon DocumentDB.

- Always commit or abort the transaction after it is complete. Leaving a transaction in an incomplete state ties up database resources and can cause write conflicts.
- It is recommended to keep transactions to the smallest number of commands needed. If you have transactions with multiple statements that can be divided up into multiple smaller transactions, it is advisable to do so to reduce the likelihood of a timeout. Always aim to create short transactions, not long-running reads.

Limitations

- Amazon DocumentDB does not support cursors within a transaction.
- Amazon DocumentDB cannot create new collections in a transaction and cannot query/update against non-existing collections.

- Document-level write locks are subject to a 1 minute timeout, which is not configurable by the user.
- On engine versions earlier than 8.0.2, retryable writes are not supported. Starting with engine version 8.0.2, retryable writes are supported outside of transactions for write operations in which each individual write affects at most one document, and transaction commit and abort operations are retryable. For more information, see Retryable writes in Amazon DocumentDB.
- Each Amazon DocumentDB instance has an upper bound on the number of concurrent transactions that can be open on the instance. For more information, see Instance quotas.
- For a given transaction, the transaction log size must be less than 32MB.
- Amazon DocumentDB does support `count()` within a transactions, but not all drivers support this capability. An alternative is to use the `countDocuments()` API, which translates the count query into an aggregation query on the client side.
- Transactions have a one minute execution limit and sessions have a 30-minute timeout. If a transaction times out, it will be aborted, and any subsequent commands issued within the session for the existing transaction will yield the following error:

```
WriteCommandError({
"ok" : 0,
"operationTime" : Timestamp(1603491424, 627726),
"code" : 251,
"errmsg" : "Given transaction number 0 does not match any in-progress transactions."
}
```

Monitoring and diagnostics

With the support for transactions in Amazon DocumentDB 4.0, additional CloudWatch metrics were added to help you monitor your transactions.

New CloudWatch Metrics

- `DatabaseTransactions`: The number of open transactions taken at a one-minute period.
- `DatabaseTransactionsAborted`: The number of aborted transactions taken at a one-minute period.
- `DatabaseTransactionsMax`: The maximum number of open transactions in a one-minute period.

- `TransactionsAborted`: The number of transactions aborted on an instance in a one-minute period.
- `TransactionsCommitted`: The number of transactions committed on an instance in a one-minute period.
- `TransactionsOpen`: The number of transactions open on an instance taken at a one-minute period.
- `TransactionsOpenMax`: The maximum number of transactions open on an instance in a one-minute period.
- `TransactionsStarted`: The number of transactions started on an instance in a one-minute period.

> **Note**
>
> For more CloudWatch metrics for Amazon DocumentDB, go to Monitoring Amazon DocumentDB with CloudWatch.

Additionally, new fields were added to both `currentOp` `lsid`, `transactionThreadId`, and a new state for "`idle transaction`" and `serverStatus` transactions: `currentActive`, `currentInactive`, `currentOpen`, `totalAborted`, `totalCommitted`, and `totalStarted`.

Transaction isolation level

When starting a transaction, you have the ability to specify both the `readConcern` and `writeConcern` as shown in the following example:

```
mySession.startTransaction({readConcern: {level: 'snapshot'}, writeConcern:
{w: 'majority'}});
```

For `readConcern`, Amazon DocumentDB supports snapshot isolation by default. If a `readConcern` of local, available, or majority are specified, Amazon DocumentDB will upgrade the `readConcern` level to snapshot. Amazon DocumentDB does not support the linearizable `readConcern` and specifying such a read concern will result in an error.

For `writeConcern`, Amazon DocumentDB supports majority by default and a write quorum is achieved when four copies of the data are persisted across three AZs. If a lower `writeConcern` is

specified, Amazon DocumentDB will upgrade the `writeConcern` to majority. Further, all Amazon DocumentDB writes are journaled and journaling cannot be disabled.

Use cases

In this section, we will walk through two use cases for transactions: multi-statement and multi-collection.

Multi-Statement Transactions

Amazon DocumentDB transactions are multi-statement, which means you can write a transaction that spans multiple statements with an explicit commit or rollback. You can group `insert`, `update`, `delete`, and `findAndModify` actions as a single atomic operation.

A common use case for multi-statement transactions is a debit-credit transaction. For example: you owe a friend money for clothes. Thus, you need to debit (withdraw) $500 from your account and credit $500 (deposit) to your friend's account. To perform that operation, you perform both the debit and credit operations within a single transaction to ensure atomicity. Doing so prevents scenarios where $500 is debited from your account, but not credited to your friend’s account. Here's what this use case would look like:

```
// *** Transfer $500 from Alice to Bob inside a transaction: Success Scenario***
// Setup bank account for Alice and Bob. Each have $1000 in their account

var databaseName = "bank";
var collectionName = "account";
var amountToTransfer = 500;

var session = db.getMongo().startSession({causalConsistency: false});
var bankDB = session.getDatabase(databaseName);
var accountColl = bankDB[collectionName];
accountColl.drop();

accountColl.insert({name: "Alice", balance: 1000});
accountColl.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountColl.find({"name": "Alice"}).next().balance;
var newAliceBalance = aliceBalance - amountToTransfer;
```

```
accountColl.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountColl.find({"name": "Alice"}).next().balance;

// add $500 to Bob's account
var bobBalance = accountColl.find({"name": "Bob"}).next().balance;
var newBobBalance = bobBalance + amountToTransfer;
accountColl.update({"name": "Bob"},{"$set": {"balance": newBobBalance}});
var findBobBalance = accountColl.find({"name": "Bob"}).next().balance;

session.commitTransaction();

accountColl.find();

// *** Transfer $500 from Alice to Bob inside a transaction: Failure Scenario***

// Setup bank account for Alice and Bob. Each have $1000 in their account
var databaseName = "bank";
var collectionName = "account";
var amountToTransfer = 500;

var session = db.getMongo().startSession({causalConsistency: false});
var bankDB = session.getDatabase(databaseName);
var accountColl = bankDB[collectionName];
accountColl.drop();

accountColl.insert({name: "Alice", balance: 1000});
accountColl.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountColl.find({"name": "Alice"}).next().balance;
var newAliceBalance = aliceBalance - amountToTransfer;
accountColl.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountColl.find({"name": "Alice"}).next().balance;

session.abortTransaction();
```

Multi-collection transactions

Our transactions are also multi-collection, which means they can be used to perform multiple operations within a single transaction and across multiple collections. This provides a consistent

view of data and maintains your data's integrity. When you commit the commands as a single <>, the transactions are all-or-nothing executions—in that, they will either all succeed or all fail.

Here is an example of multi-collection transactions, using the same scenario and data from the example for multi-statement transactions.

```
// *** Transfer $500 from Alice to Bob inside a transaction: Success Scenario***

// Setup bank account for Alice and Bob. Each have $1000 in their account
var amountToTransfer = 500;
var collectionName = "account";

var session = db.getMongo().startSession({causalConsistency: false});
var accountCollInBankA = session.getDatabase("bankA")[collectionName];
var accountCollInBankB = session.getDatabase("bankB")[collectionName];

accountCollInBankA.drop();
accountCollInBankB.drop();

accountCollInBankA.insert({name: "Alice", balance: 1000});
accountCollInBankB.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountCollInBankA.find({"name": "Alice"}).next().balance;
var newAliceBalance = aliceBalance - amountToTransfer;
accountCollInBankA.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountCollInBankA.find({"name": "Alice"}).next().balance;

// add $500 to Bob's account
var bobBalance = accountCollInBankB.find({"name": "Bob"}).next().balance;
var newBobBalance = bobBalance + amountToTransfer;
accountCollInBankB.update({"name": "Bob"},{"$set": {"balance": newBobBalance}});
var findBobBalance = accountCollInBankB.find({"name": "Bob"}).next().balance;

session.commitTransaction();

accountCollInBankA.find(); // Alice holds $500 in bankA
accountCollInBankB.find(); // Bob holds $1500 in bankB

// *** Transfer $500 from Alice to Bob inside a transaction: Failure Scenario***
```

```
// Setup bank account for Alice and Bob. Each have $1000 in their account
var collectionName = "account";
var amountToTransfer = 500;

var session = db.getMongo().startSession({causalConsistency: false});
var accountCollInBankA = session.getDatabase("bankA")[collectionName];
var accountCollInBankB = session.getDatabase("bankB")[collectionName];

accountCollInBankA.drop();
accountCollInBankB.drop();

accountCollInBankA.insert({name: "Alice", balance: 1000});
accountCollInBankB.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountCollInBankA.find({"name": "Alice"}).next().balance;
var newAliceBalance = aliceBalance - amountToTransfer;
accountCollInBankA.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountCollInBankA.find({"name": "Alice"}).next().balance;

// add $500 to Bob's account
var bobBalance = accountCollInBankB.find({"name": "Bob"}).next().balance;
var newBobBalance = bobBalance + amountToTransfer;
accountCollInBankB.update({"name": "Bob"},{"$set": {"balance": newBobBalance}});
var findBobBalance = accountCollInBankB.find({"name": "Bob"}).next().balance;

session.abortTransaction();

accountCollInBankA.find(); // Alice holds $1000 in bankA
accountCollInBankB.find(); // Bob holds $1000 in bankB
```

Transaction API examples for callback API

The callback API is only available for 4.2+ drivers.

Javascript

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Javascript.

```
// *** Transfer $500 from Alice to Bob inside a transaction: Success ***
// Setup bank account for Alice and Bob. Each have $1000 in their account
var databaseName = "bank";
var collectionName = "account";
var amountToTransfer = 500;

var session = db.getMongo().startSession({causalConsistency: false});
var bankDB = session.getDatabase(databaseName);
var accountColl = bankDB[collectionName];
accountColl.drop();

accountColl.insert({name: "Alice", balance: 1000});
accountColl.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountColl.find({"name": "Alice"}).next().balance;
assert(aliceBalance >= amountToTransfer);
var newAliceBalance = aliceBalance - amountToTransfer;
accountColl.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountColl.find({"name": "Alice"}).next().balance;
assert.eq(newAliceBalance, findAliceBalance);

// add $500 to Bob's account
var bobBalance = accountColl.find({"name": "Bob"}).next().balance;
var newBobBalance = bobBalance + amountToTransfer;
accountColl.update({"name": "Bob"},{"$set": {"balance": newBobBalance}});
var findBobBalance = accountColl.find({"name": "Bob"}).next().balance;
assert.eq(newBobBalance, findBobBalance);

session.commitTransaction();

accountColl.find();
```

Node.js

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Node.js.

```
// Node.js callback API:

const bankDB = await mongoclient.db("bank");
```

```
var accountColl = await bankDB.createCollection("account");
var amountToTransfer = 500;

const session = mongoclient.startSession({causalConsistency: false});
await accountColl.drop();

await accountColl.insertOne({name: "Alice", balance: 1000}, { session });
await accountColl.insertOne({name: "Bob", balance: 1000}, { session });

const transactionOptions = {
    readConcern: { level: 'snapshot' },
    writeConcern: { w: 'majority' }
    };

// deduct $500 from Alice's account
var aliceBalance = await accountColl.findOne({name: "Alice"}, {session});
assert(aliceBalance.balance >= amountToTransfer);
var newAliceBalance = aliceBalance - amountToTransfer;
session.startTransaction(transactionOptions);
await accountColl.updateOne({name: "Alice"}, {$set: {balance: newAliceBalance}},
 {session });
await session.commitTransaction();
aliceBalance = await accountColl.findOne({name: "Alice"}, {session});
assert(newAliceBalance == aliceBalance.balance);

// add $500 to Bob's account
var bobBalance = await accountColl.findOne({name: "Bob"}, {session});
var newBobBalance = bobBalance.balance + amountToTransfer;
session.startTransaction(transactionOptions);
await accountColl.updateOne({name: "Bob"}, {$set: {balance: newBobBalance}},
 {session });
await session.commitTransaction();
bobBalance = await accountColl.findOne({name: "Bob"}, {session});
assert(newBobBalance == bobBalance.balance);
```

C#

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with C#.

```
// C# Callback API

var dbName = "bank";
```

```
var collName = "account";
var amountToTransfer = 500;

using (var session = client.StartSession(new ClientSessionOptions{CausalConsistency
 = false}))
{
    var bankDB = client.GetDatabase(dbName);
    var accountColl = bankDB.GetCollection<BsonDocument>(collName);
    bankDB.DropCollection(collName);
    accountColl.InsertOne(session, new BsonDocument { {"name", "Alice"}, {"balance",
 1000 } });
    accountColl.InsertOne(session, new BsonDocument { {"name", "Bob"}, {"balance",
 1000 } });

    // start transaction
    var transactionOptions = new TransactionOptions(
            readConcern: ReadConcern.Snapshot,
            writeConcern: WriteConcern.WMajority);
    var result = session.WithTransaction(
        (sess, cancellationtoken) =>
        {
            // deduct $500 from Alice's account
            var aliceBalance = accountColl.Find(sess,
 Builders<BsonDocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
            Debug.Assert(aliceBalance >= amountToTransfer);
            var newAliceBalance = aliceBalance.AsInt32 - amountToTransfer;
            accountColl.UpdateOne(sess, Builders<BsonDocument>.Filter.Eq("name",
 "Alice"),
                                      Builders<BsonDocument>.Update.Set("balance",
 newAliceBalance));
            aliceBalance = accountColl.Find(sess,
 Builders<BsonDocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
            Debug.Assert(aliceBalance == newAliceBalance);

            // add $500 from Bob's account
            var bobBalance = accountColl.Find(sess,
 Builders<BsonDocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
            var newBobBalance = bobBalance.AsInt32 + amountToTransfer;
            accountColl.UpdateOne(sess, Builders<BsonDocument>.Filter.Eq("name",
 "Bob"),
```

```
                                    Builders<BsonDocument>.Update.Set("balance",
 newBobBalance));
            bobBalance = accountColl.Find(sess,
 Builders<BsonDocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
            Debug.Assert(bobBalance == newBobBalance);

            return "Transaction committed";
        }, transactionOptions);
   // check values outside of transaction
    var aliceNewBalance = accountColl.Find(Builders<BsonDocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
    var bobNewBalance = accountColl.Find(Builders<BsonDocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
    Debug.Assert(aliceNewBalance ==  500);
    Debug.Assert(bobNewBalance ==  1500);
}
```

Ruby

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Ruby.

```
// Ruby Callback API

dbName = "bank"
collName = "account"
amountToTransfer = 500

session = client.start_session(:causal_consistency=> false)
bankDB = Mongo::Database.new(client, dbName)
accountColl = bankDB[collName]
accountColl.drop()

accountColl.insert_one({"name"=>"Alice", "balance"=>1000})
accountColl.insert_one({"name"=>"Bob", "balance"=>1000})

    # start transaction
    session.with_transaction(read_concern: {level: :snapshot}, write_concern:
 {w: :majority}) do
        # deduct $500 from Alice's account
        aliceBalance = accountColl.find({"name"=>"Alice"}, :session=>
 session).first['balance']
```

```
        assert aliceBalance >= amountToTransfer
        newAliceBalance = aliceBalance - amountToTransfer
        accountColl.update_one({"name"=>"Alice"}, { "$set" =>
 {"balance"=>newAliceBalance} }, :session=> session)
        aliceBalance = accountColl.find({"name"=>>"Alice"}, :session=>
 session).first['balance']
        assert_equal(newAliceBalance, aliceBalance)

        # add $500 from Bob's account
        bobBalance = accountColl.find({"name"=>"Bob"}, :session=>
 session).first['balance']
        newBobBalance = bobBalance + amountToTransfer
        accountColl.update_one({"name"=>"Bob"}, { "$set" =>
 {"balance"=>newBobBalance} }, :session=> session)
        bobBalance = accountColl.find({"name"=>"Bob"}, :session=>
 session).first['balance']
        assert_equal(newBobBalance, bobBalance)
    end

   # check results outside of transaction
    aliceBalance = accountColl.find({"name"=>"Alice"}).first['balance']
    bobBalance = accountColl.find({"name"=>"Bob"}).first['balance']
    assert_equal(aliceBalance, 500)
    assert_equal(bobBalance, 1500)

session.end_session
```

Go

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Go.

```
// Go - Callback API
type Account struct {
    Name string
    Balance  int
}

ctx := context.TODO()

dbName := "bank"
collName := "account"
amountToTransfer := 500
```

```
session, err := client.StartSession(options.Session().SetCausalConsistency(false))
assert.NilError(t, err)
defer session.EndSession(ctx)

bankDB := client.Database(dbName)
accountColl := bankDB.Collection(collName)
accountColl.Drop(ctx)

_, err = accountColl.InsertOne(ctx, bson.M{"name" : "Alice", "balance":1000})
_, err = accountColl.InsertOne(ctx, bson.M{"name" : "Bob", "balance":1000})

transactionOptions := options.Transaction().SetReadConcern(readconcern.Snapshot()).

 SetWriteConcern(writeconcern.New(writeconcern.WMajority()))
_, err = session.WithTransaction(ctx, func(sessionCtx mongo.SessionContext)
 (interface{}, error) {
    var result Account
    // deduct $500 from Alice's account
    err = accountColl.FindOne(sessionCtx, bson.M{"name": "Alice"}).Decode(&result)
    aliceBalance := result.Balance
    newAliceBalance := aliceBalance - amountToTransfer
    _, err = accountColl.UpdateOne(sessionCtx, bson.M{"name": "Alice"},
 bson.M{"$set": bson.M{"balance": newAliceBalance}})
    err = accountColl.FindOne(sessionCtx, bson.M{"name": "Alice"}).Decode(&result)
    aliceBalance = result.Balance
    assert.Equal(t, aliceBalance, newAliceBalance)

    // add $500 to Bob's account
    err = accountColl.FindOne(sessionCtx, bson.M{"name": "Bob"}).Decode(&result)
    bobBalance := result.Balance
    newBobBalance := bobBalance + amountToTransfer
    _, err = accountColl.UpdateOne(sessionCtx, bson.M{"name": "Bob"}, bson.M{"$set":
 bson.M{"balance": newBobBalance}})
    err = accountColl.FindOne(sessionCtx, bson.M{"name": "Bob"}).Decode(&result)
    bobBalance = result.Balance
    assert.Equal(t, bobBalance, newBobBalance)

    if err != nil {
        return nil, err
    }
    return "transaction committed", err
}, transactionOptions)
```

```
// check results outside of transaction
var result Account
err = accountColl.FindOne(ctx, bson.M{"name": "Alice"}).Decode(&result)
aliceNewBalance := result.Balance
err = accountColl.FindOne(ctx, bson.M{"name": "Bob"}).Decode(&result)
bobNewBalance := result.Balance
assert.Equal(t, aliceNewBalance, 500)
assert.Equal(t, bobNewBalance, 1500)
```

Java

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Java.

```
// Java (sync) - Callback API
MongoDatabase bankDB = mongoClient.getDatabase("bank");
MongoCollection accountColl = bankDB.getCollection("account");
accountColl.drop();
int amountToTransfer = 500;

// add sample data
accountColl.insertOne(new Document("name", "Alice").append("balance", 1000));
accountColl.insertOne(new Document("name", "Bob").append("balance", 1000));

TransactionOptions txnOptions = TransactionOptions.builder()
        .readConcern(ReadConcern.SNAPSHOT)
        .writeConcern(WriteConcern.MAJORITY)
        .build();
ClientSessionOptions sessionOptions =
 ClientSessionOptions.builder().causallyConsistent(false).build();
try ( ClientSession clientSession = mongoClient.startSession(sessionOptions) ) {
    clientSession.withTransaction(new TransactionBody<Void>() {
        @Override
        public Void execute() {
            // deduct $500 from Alice's account
            List<Document> documentList = new ArrayList<>();
            accountColl.find(clientSession, new Document("name",
 "Alice")).into(documentList);
            int aliceBalance = (int) documentList.get(0).get("balance");
            int newAliceBalance = aliceBalance - amountToTransfer;

            accountColl.updateOne(clientSession, new Document("name", "Alice"), new
 Document("$set", new Document("balance", newAliceBalance)));
```

```
            // check Alice's new balance
            documentList = new ArrayList<>();
            accountColl.find(clientSession, new Document("name",
 "Alice")).into(documentList);
            int updatedBalance = (int) documentList.get(0).get("balance");
            Assert.assertEquals(updatedBalance, newAliceBalance);

            // add $500 to Bob's account
            documentList = new ArrayList<>();
            accountColl.find(clientSession, new Document("name",
 "Bob")).into(documentList);
            int bobBalance = (int) documentList.get(0).get("balance");
            int newBobBalance = bobBalance + amountToTransfer;

            accountColl.updateOne(clientSession, new Document("name", "Bob"), new
 Document("$set", new Document("balance", newBobBalance)));

            // check Bob's new balance
            documentList = new ArrayList<>();
            accountColl.find(clientSession, new Document("name",
 "Bob")).into(documentList);
            updatedBalance = (int) documentList.get(0).get("balance");
            Assert.assertEquals(updatedBalance, newBobBalance);

            return null;
        }
    }, txnOptions);
}
```

C

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with C.

```
// Sample Code for C with Callback

#include <bson.h>
#include <mongoc.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
```

```
typedef struct {
    int64_t balance;
    bson_t *account;
    bson_t *opts;
    mongoc_collection_t *collection;
} ctx_t;

bool callback_session (mongoc_client_session_t *session, void *ctx, bson_t **reply,
 bson_error_t *error)
{
    bool r = true;
    ctx_t *data = (ctx_t *) ctx;
    bson_t local_reply;
    bson_t *selector = data->account;
    bson_t *update = BCON_NEW ("$set", "{", "balance", BCON_INT64 (data->balance),
 "}");

    mongoc_collection_update_one (data->collection, selector, update, data->opts,
 &local_reply, error);

    *reply = bson_copy (&local_reply);
    bson_destroy (&local_reply);
    bson_destroy (update);
    return r;
}

void test_callback_money_transfer(mongoc_client_t* client, mongoc_collection_t*
 collection, int amount_to_transfer){

    bson_t reply;
    bool r = true;
    const bson_t *doc;
    bson_iter_t iter;
    ctx_t alice_ctx;
    ctx_t bob_ctx;
    bson_error_t error;

    // find query
    bson_t *alice_query = bson_new ();
    BSON_APPEND_UTF8(alice_query, "name", "Alice");

    bson_t *bob_query = bson_new ();
    BSON_APPEND_UTF8(bob_query, "name", "Bob");
```

```
   // create session
   // set causal consistency to false
   mongoc_session_opt_t *session_opts = mongoc_session_opts_new ();
   mongoc_session_opts_set_causal_consistency (session_opts, false);
   // start the session
   mongoc_client_session_t *client_session = mongoc_client_start_session (client,
session_opts, &error);

   // add session to options
   bson_t *opts = bson_new();
   mongoc_client_session_append (client_session, opts, &error);

   // deduct 500 from Alice
   // find account balance of Alice
   mongoc_cursor_t *cursor = mongoc_collection_find_with_opts (collection,
alice_query, NULL, NULL);
   mongoc_cursor_next (cursor, &doc);
   bson_iter_init (&iter, doc);
   bson_iter_find (&iter, "balance");
   int64_t alice_balance = (bson_iter_value (&iter))->value.v_int64;
   assert(alice_balance >= amount_to_transfer);
   int64_t new_alice_balance = alice_balance - amount_to_transfer;

   // set variables which will be used by callback function
   alice_ctx.collection = collection;
   alice_ctx.opts = opts;
   alice_ctx.balance = new_alice_balance;
   alice_ctx.account = alice_query;

   // callback
   r = mongoc_client_session_with_transaction (client_session, &callback_session,
NULL, &alice_ctx, &reply, &error);
   assert(r);

   // find account balance of Alice after transaction
   cursor = mongoc_collection_find_with_opts (collection, alice_query, NULL, NULL);
   mongoc_cursor_next (cursor, &doc);
   bson_iter_init (&iter, doc);
   bson_iter_find (&iter, "balance");
   alice_balance = (bson_iter_value (&iter))->value.v_int64;
   assert(alice_balance == new_alice_balance);
   assert(alice_balance == 500);

       // add 500 to bob's balance
```

```
   // find account balance of Bob
   cursor = mongoc_collection_find_with_opts (collection, bob_query, NULL, NULL);
   mongoc_cursor_next (cursor, &doc);
   bson_iter_init (&iter, doc);
   bson_iter_find (&iter, "balance");
   int64_t bob_balance = (bson_iter_value (&iter))->value.v_int64;
   int64_t new_bob_balance = bob_balance + amount_to_transfer;

   bob_ctx.collection = collection;
   bob_ctx.opts = opts;
   bob_ctx.balance = new_bob_balance;
   bob_ctx.account = bob_query;

   // set read & write concern
   mongoc_read_concern_t *read_concern = mongoc_read_concern_new ();
   mongoc_write_concern_t *write_concern = mongoc_write_concern_new ();
   mongoc_transaction_opt_t *txn_opts = mongoc_transaction_opts_new ();

   mongoc_write_concern_set_w(write_concern, MONGOC_WRITE_CONCERN_W_MAJORITY);
   mongoc_read_concern_set_level(read_concern, MONGOC_READ_CONCERN_LEVEL_SNAPSHOT);
   mongoc_transaction_opts_set_write_concern (txn_opts, write_concern);
   mongoc_transaction_opts_set_read_concern (txn_opts, read_concern);

   // callback
   r = mongoc_client_session_with_transaction (client_session, &callback_session,
txn_opts, &bob_ctx, &reply, &error);
   assert(r);

// find account balance of Bob after transaction
   cursor = mongoc_collection_find_with_opts (collection, bob_query, NULL, NULL);
   mongoc_cursor_next (cursor, &doc);
   bson_iter_init (&iter, doc);
   bson_iter_find (&iter, "balance");
   bob_balance = (bson_iter_value (&iter))->value.v_int64;
   assert(bob_balance == new_bob_balance);
   assert(bob_balance == 1500);

   // cleanup
   bson_destroy(alice_query);
   bson_destroy(bob_query);
   mongoc_client_session_destroy(client_session);
   bson_destroy(opts);
   mongoc_transaction_opts_destroy(txn_opts);
   mongoc_read_concern_destroy(read_concern);
```

```
    mongoc_write_concern_destroy(write_concern);
    mongoc_cursor_destroy(cursor);
    bson_destroy(doc);
}
int main(int argc, char* argv[]) {
    mongoc_init ();
    mongoc_client_t* client = mongoc_client_new (<connection uri>);
    bson_error_t error;

    // connect to bank db
    mongoc_database_t *database = mongoc_client_get_database (client, "bank");
    // access account collection
    mongoc_collection_t* collection = mongoc_client_get_collection(client, "bank",
 "account");
    // set amount to transfer
    int64_t amount_to_transfer = 500;
    // delete the collection if already existing
    mongoc_collection_drop(collection, &error);

    // open Alice account
    bson_t *alice_account = bson_new ();
    BSON_APPEND_UTF8(alice_account, "name", "Alice");
    BSON_APPEND_INT64(alice_account, "balance", 1000);

    // open Bob account
    bson_t *bob_account = bson_new ();
    BSON_APPEND_UTF8(bob_account, "name", "Bob");
    BSON_APPEND_INT64(bob_account, "balance", 1000);

    bool r = true;

    r = mongoc_collection_insert_one(collection, alice_account, NULL, NULL, &error);
    if (!r) {printf("Error encountered:%s", error.message);}
    r = mongoc_collection_insert_one(collection, bob_account, NULL, NULL, &error);
    if (!r) {printf("Error encountered:%s", error.message);}

    test_callback_money_transfer(client, collection, amount_to_transfer);

}
```

Python

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Python.

```
// Sample Python code with callback api

import pymongo

def callback(session, balance, query):
    collection.update_one(query, {'$set': {"balance": balance}}, session=session)

client = pymongo.MongoClient(<connection uri>)
rc_snapshot = pymongo.read_concern.ReadConcern('snapshot')
wc_majority = pymongo.write_concern.WriteConcern('majority')

# To start, drop and create an account collection and insert balances for both Alice
 and Bob
collection = client.get_database("bank").get_collection("account")
collection.drop()
collection.insert_one({"_id": 1, "name": "Alice", "balance": 1000})
collection.insert_one({"_id": 2, "name": "Bob", "balance": 1000})

amount_to_transfer = 500

# deduct 500 from Alice's account
alice_balance = collection.find_one({"name": "Alice"}).get("balance")
assert alice_balance >= amount_to_transfer
new_alice_balance = alice_balance - amount_to_transfer

with client.start_session({'causalConsistency':False}) as session:
    session.with_transaction(lambda s: callback(s, new_alice_balance, {"name":
 "Alice"}), read_concern=rc_snapshot, write_concern=wc_majority)

updated_alice_balance = collection.find_one({"name": "Alice"}).get("balance")
assert updated_alice_balance == new_alice_balance

# add 500 to Bob's account
bob_balance = collection.find_one({"name": "Bob"}).get("balance")
assert bob_balance >= amount_to_transfer
new_bob_balance = bob_balance + amount_to_transfer

with client.start_session({'causalConsistency':False}) as session:
    session.with_transaction(lambda s: callback(s, new_bob_balance, {"name":
 "Bob"}), read_concern=rc_snapshot, write_concern=wc_majority)

updated_bob_balance = collection.find_one({"name": "Bob"}).get("balance")
assert updated_bob_balance == new_bob_balance
```

Transaction API examples for core API

Javascript

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Javascript.

```
// *** Transfer $500 from Alice to Bob inside a transaction: Success ***
// Setup bank account for Alice and Bob. Each have $1000 in their account
var databaseName = "bank";
var collectionName = "account";
var amountToTransfer = 500;

var session = db.getMongo().startSession({causalConsistency: false});
var bankDB = session.getDatabase(databaseName);
var accountColl = bankDB[collectionName];
accountColl.drop();

accountColl.insert({name: "Alice", balance: 1000});
accountColl.insert({name: "Bob", balance: 1000});

session.startTransaction();

// deduct $500 from Alice's account
var aliceBalance = accountColl.find({"name": "Alice"}).next().balance;
assert(aliceBalance >= amountToTransfer);
var newAliceBalance = aliceBalance - amountToTransfer;
accountColl.update({"name": "Alice"},{"$set": {"balance": newAliceBalance}});
var findAliceBalance = accountColl.find({"name": "Alice"}).next().balance;
assert.eq(newAliceBalance, findAliceBalance);

// add $500 to Bob's account
var bobBalance = accountColl.find({"name": "Bob"}).next().balance;
var newBobBalance = bobBalance + amountToTransfer;
accountColl.update({"name": "Bob"},{"$set": {"balance": newBobBalance}});
var findBobBalance = accountColl.find({"name": "Bob"}).next().balance;
assert.eq(newBobBalance, findBobBalance);

session.commitTransaction();

accountColl.find();
```

C#

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with C#.

```
// C# Core API

public void TransferMoneyWithRetry(IMongoCollection<bSondocument> accountColl,
 IClientSessionHandle session)
{
    var amountToTransfer = 500;

    // start transaction
   var transactionOptions = new TransactionOptions(
                readConcern: ReadConcern.Snapshot,
                writeConcern: WriteConcern.WMajority);
    session.StartTransaction(transactionOptions);
   try
    {
        // deduct $500 from Alice's account
        var aliceBalance = accountColl.Find(session,
 Builders<bSondocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
       Debug.Assert(aliceBalance >= amountToTransfer);
       var newAliceBalance = aliceBalance.AsInt32 - amountToTransfer;
       accountColl.UpdateOne(session, Builders<bSondocument>.Filter.Eq("name",
 "Alice"),
                                Builders<bSondocument>.Update.Set("balance",
 newAliceBalance));
       aliceBalance = accountColl.Find(session,
 Builders<bSondocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
       Debug.Assert(aliceBalance == newAliceBalance);

       // add $500 from Bob's account
       var bobBalance = accountColl.Find(session,
 Builders<bSondocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
       var newBobBalance = bobBalance.AsInt32 + amountToTransfer;
       accountColl.UpdateOne(session, Builders<bSondocument>.Filter.Eq("name",
 "Bob"),
                                Builders<bSondocument>.Update.Set("balance",
 newBobBalance));
```

```
        bobBalance = accountColl.Find(session,
 Builders<bSondocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
        Debug.Assert(bobBalance == newBobBalance);

    }
    catch (Exception e)
    {
        session.AbortTransaction();
        throw;
    }

    session.CommitTransaction();
 }

}
public void DoTransactionWithRetry(MongoClient client)
{
    var dbName = "bank";
    var collName = "account";
    using (var session = client.StartSession(new
 ClientSessionOptions{CausalConsistency = false}))
    {
        try
        {
            var bankDB = client.GetDatabase(dbName);
            var accountColl = bankDB.GetCollection<bSondocument>(collName);
            bankDB.DropCollection(collName);
            accountColl.InsertOne(session, new BsonDocument { {"name", "Alice"},
 {"balance", 1000 } });
            accountColl.InsertOne(session, new BsonDocument { {"name", "Bob"},
 {"balance", 1000 } });

            while(true) {
                try
                {
                        TransferMoneyWithRetry(accountColl, session);
                        break;
                }
                catch (MongoException e)
                {
                    if(e.HasErrorLabel("TransientTransactionError"))
                    {
                        continue;
```

```
                    }
                    else
                    {
                        throw;
                    }
                }
            }

            // check values outside of transaction
            var aliceNewBalance =
 accountColl.Find(Builders<bSondocument>.Filter.Eq("name",
 "Alice")).FirstOrDefault().GetValue("balance");
            var bobNewBalance =
 accountColl.Find(Builders<bSondocument>.Filter.Eq("name",
 "Bob")).FirstOrDefault().GetValue("balance");
            Debug.Assert(aliceNewBalance ==  500);
            Debug.Assert(bobNewBalance ==  1500);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error running transaction: " + e.Message);
        }
    }
}
```

Ruby

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Ruby.

```
# Ruby Core API

def transfer_money_w_retry(session, accountColl)
    amountToTransfer = 500

    session.start_transaction(read_concern: {level: :snapshot}, write_concern:
 {w: :majority})
    # deduct $500 from Alice's account
    aliceBalance = accountColl.find({"name"=>"Alice"}, :session=>
 session).first['balance']
    assert aliceBalance >= amountToTransfer
    newAliceBalance = aliceBalance - amountToTransfer
```

```
    accountColl.update_one({"name"=>"Alice"}, { "$set" =>
 {"balance"=>newAliceBalance} }, :session=> session)
    aliceBalance = accountColl.find({"name"=>"Alice"}, :session=>
 session).first['balance']
    assert_equal(newAliceBalance, aliceBalance)

    # add $500 to Bob's account
    bobBalance = accountColl.find({"name"=>"Bob"}, :session=>
 session).first['balance']
    newBobBalance = bobBalance + amountToTransfer
    accountColl.update_one({"name"=>"Bob"}, { "$set" =>
 {"balance"=>newBobBalance} }, :session=> session)
    bobBalance = accountColl.find({"name"=>"Bob"}, :session=>
 session).first['balance']
    assert_equal(newBobBalance, bobBalance)

    session.commit_transaction

end

def do_txn_w_retry(client)
     dbName = "bank"
    collName = "account"

    session = client.start_session(:causal_consistency=> false)
    bankDB = Mongo::Database.new(client, dbName)
    accountColl = bankDB[collName]
    accountColl.drop()

    accountColl.insert_one({"name"=>"Alice", "balance"=>1000})
    accountColl.insert_one({"name"=>"Bob", "balance"=>1000})

    begin
        transferMoneyWithRetry(session, accountColl)
        puts "transaction committed"
    rescue Mongo::Error => e
        if e.label?('TransientTransactionError')
            retry
        else
            puts "transaction failed"
            raise
        end
    end
```

```
    # check results outside of transaction
    aliceBalance = accountColl.find({"name"=>"Alice"}).first['balance']
    bobBalance = accountColl.find({"name"=>"Bob"}).first['balance']
    assert_equal(aliceBalance, 500)
    assert_equal(bobBalance, 1500)

end
```

Go

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Go.

```
// Go - Core API
type Account struct {
    Name string
    Balance  int
}

func transferMoneyWithRetry(sessionContext mongo.SessionContext, accountColl
 *mongo.Collection, t *testing.T) error {
    amountToTransfer := 500

    transactionOptions :=
 options.Transaction().SetReadConcern(readconcern.Snapshot()).

 SetWriteConcern(writeconcern.New(writeconcern.WMajority()))
    if err := sessionContext.StartTransaction(transactionOptions); err != nil {
        panic(err)
    }

    var result Account
    // deduct $500 from Alice's account
    err := accountColl.FindOne(sessionContext, bson.M{"name":
 "Alice"}).Decode(&result)
    aliceBalance := result.Balance
    newAliceBalance := aliceBalance - amountToTransfer
    _, err = accountColl.UpdateOne(sessionContext, bson.M{"name": "Alice"},
 bson.M{"$set": bson.M{"balance": newAliceBalance}})
    if err != nil {
        sessionContext.AbortTransaction(sessionContext)
    }
```

```
    err = accountColl.FindOne(sessionContext, bson.M{"name":
 "Alice"}).Decode(&result)
    aliceBalance = result.Balance
    assert.Equal(t, aliceBalance, newAliceBalance)

    // add $500 to Bob's account
    err = accountColl.FindOne(sessionContext, bson.M{"name": "Bob"}).Decode(&result)
    bobBalance := result.Balance
    newBobBalance := bobBalance + amountToTransfer
    _, err = accountColl.UpdateOne(sessionContext, bson.M{"name": "Bob"},
 bson.M{"$set": bson.M{"balance": newBobBalance}})
    if err != nil {
        sessionContext.AbortTransaction(sessionContext)
    }
    err = accountColl.FindOne(sessionContext, bson.M{"name": "Bob"}).Decode(&result)
    bobBalance = result.Balance
    assert.Equal(t, bobBalance, newBobBalance)

    err = sessionContext.CommitTransaction(sessionContext)
    return err
}

func doTransactionWithRetry(t *testing.T) {
    ctx := context.TODO()

    dbName := "bank"
    collName := "account"
    bankDB := client.Database(dbName)
    accountColl := bankDB.Collection(collName)

    client.UseSessionWithOptions(ctx, options.Session().SetCausalConsistency(false),
 func(sessionContext mongo.SessionContext) error {
        accountColl.Drop(ctx)
        accountColl.InsertOne(sessionContext, bson.M{"name" : "Alice",
 "balance":1000})
        accountColl.InsertOne(sessionContext, bson.M{"name" : "Bob",
 "balance":1000})
        for {
            err := transferMoneyWithRetry(sessionContext, accountColl, t)
            if err == nil {
                println("transaction committed")
                return nil
            }
```

```
            if mongoErr := err.(mongo.CommandError);
 mongoErr.HasErrorLabel("TransientTransactionError") {
                continue
            }
            println("transaction failed")
            return err
        }
    })

    // check results outside of transaction
    var result Account
    accountColl.FindOne(ctx, bson.M{"name": "Alice"}).Decode(&result)
    aliceBalance := result.Balance
    assert.Equal(t, aliceBalance, 500)
    accountColl.FindOne(ctx, bson.M{"name": "Bob"}).Decode(&result)
    bobBalance := result.Balance
    assert.Equal(t, bobBalance, 1500)
}
```

Java

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Java.

```
// Java (sync) - Core API

public void transferMoneyWithRetry() {
   // connect to server
    MongoClientURI mongoURI = new MongoClientURI(uri);
    MongoClient mongoClient = new MongoClient(mongoURI);

    MongoDatabase bankDB = mongoClient.getDatabase("bank");
    MongoCollection accountColl = bankDB.getCollection("account");
    accountColl.drop();

   // insert some sample data
    accountColl.insertOne(new Document("name", "Alice").append("balance", 1000));
    accountColl.insertOne(new Document("name", "Bob").append("balance", 1000));

    while (true) {
        try {
            doTransferMoneyWithRetry(accountColl, mongoClient);
            break;
```

```
        } catch (MongoException e) {
            if (e.hasErrorLabel(MongoException.TRANSIENT_TRANSACTION_ERROR_LABEL)) {
                continue;
            } else {
                throw e;
            }
        }
    }
}

public void doTransferMoneyWithRetry(MongoCollection accountColl, MongoClient
 mongoClient) {
    int amountToTransfer = 500;

   TransactionOptions txnOptions = TransactionOptions.builder()
      .readConcern(ReadConcern.SNAPSHOT)
      .writeConcern(WriteConcern.MAJORITY)
      .build();
    ClientSessionOptions sessionOptions =
 ClientSessionOptions.builder().causallyConsistent(false).build();
    try ( ClientSession clientSession = mongoClient.startSession(sessionOptions) ) {
        clientSession.startTransaction(txnOptions);

        // deduct $500 from Alice's account
        List<Document> documentList = new ArrayList<>();
        accountColl.find(clientSession, new Document("name",
 "Alice")).into(documentList);
        int aliceBalance = (int) documentList.get(0).get("balance");
        Assert.assertTrue(aliceBalance >= amountToTransfer);
        int newAliceBalance = aliceBalance - amountToTransfer;
        accountColl.updateOne(clientSession, new Document("name", "Alice"), new
 Document("$set", new Document("balance", newAliceBalance)));

        // check Alice's new balance
        documentList = new ArrayList<>();
        accountColl.find(clientSession, new Document("name",
 "Alice")).into(documentList);
        int updatedBalance = (int) documentList.get(0).get("balance");
        Assert.assertEquals(updatedBalance, newAliceBalance);

        // add $500 to Bob's account
        documentList = new ArrayList<>();
        accountColl.find(clientSession, new Document("name",
 "Bob")).into(documentList);
```

```
        int bobBalance = (int) documentList.get(0).get("balance");
        int newBobBalance = bobBalance + amountToTransfer;
        accountColl.updateOne(clientSession, new Document("name", "Bob"), new
 Document("$set", new Document("balance", newBobBalance)));

        // check Bob's new balance
        documentList = new ArrayList<>();
        accountColl.find(clientSession, new Document("name",
 "Bob")).into(documentList);
        updatedBalance = (int) documentList.get(0).get("balance");
        Assert.assertEquals(updatedBalance, newBobBalance);

        // commit transaction
        clientSession.commitTransaction();
    }
}
// Java (async) -- Core API
public void transferMoneyWithRetry() {
    // connect to the server
    MongoClient mongoClient = MongoClients.create(uri);

    MongoDatabase bankDB = mongoClient.getDatabase("bank");
    MongoCollection accountColl = bankDB.getCollection("account");
    SubscriberLatchWrapper<Void> dropCallback = new SubscriberLatchWrapper<>();
    mongoClient.getDatabase("bank").drop().subscribe(dropCallback);
    dropCallback.await();

    // insert some sample data
    SubscriberLatchWrapper<InsertOneResult> insertionCallback = new
 SubscriberLatchWrapper<>();
    accountColl.insertOne(new Document("name", "Alice").append("balance",
 1000)).subscribe(insertionCallback);
    insertionCallback.await();

    insertionCallback = new SubscriberLatchWrapper<>();
    accountColl.insertOne(new Document("name", "Bob").append("balance",
 1000)).subscribe(insertionCallback);;
    insertionCallback.await();

    while (true) {
        try {
            doTransferMoneyWithRetry(accountColl, mongoClient);
            break;
        } catch (MongoException e) {
```

```
            if (e.hasErrorLabel(MongoException.TRANSIENT_TRANSACTION_ERROR_LABEL)) {
                continue;
            } else {
                throw e;
            }
        }
    }
}

public void doTransferMoneyWithRetry(MongoCollection accountColl, MongoClient
 mongoClient) {
    int amountToTransfer = 500;

    // start the transaction
    TransactionOptions txnOptions = TransactionOptions.builder()
            .readConcern(ReadConcern.SNAPSHOT)
            .writeConcern(WriteConcern.MAJORITY)
            .build();
    ClientSessionOptions sessionOptions =
 ClientSessionOptions.builder().causallyConsistent(false).build();

    SubscriberLatchWrapper<ClientSession> sessionCallback = new
 SubscriberLatchWrapper<>();
    mongoClient.startSession(sessionOptions).subscribe(sessionCallback);
    ClientSession session = sessionCallback.get().get(0);
    session.startTransaction(txnOptions);

    // deduct $500 from Alice's account
    SubscriberLatchWrapper<Document> findCallback = new SubscriberLatchWrapper<>();
    accountColl.find(session, new Document("name",
 "Alice")).first().subscribe(findCallback);
    Document documentFound = findCallback.get().get(0);
    int aliceBalance = (int) documentFound.get("balance");
    int newAliceBalance = aliceBalance - amountToTransfer;

    SubscriberLatchWrapper<UpdateResult> updateCallback = new
 SubscriberLatchWrapper<>();
    accountColl.updateOne(session, new Document("name",
 "Alice"), new Document("$set", new Document("balance",
 newAliceBalance))).subscribe(updateCallback);
    updateCallback.await();

    // check Alice's new balance
    findCallback = new SubscriberLatchWrapper<>();
```

```
    accountColl.find(session, new Document("name",
 "Alice")).first().subscribe(findCallback);
    documentFound = findCallback.get().get(0);
    int updatedBalance = (int) documentFound.get("balance");
    Assert.assertEquals(updatedBalance, newAliceBalance);

    // add $500 to Bob's account
    findCallback = new SubscriberLatchWrapper<>();
    accountColl.find(session, new Document("name",
 "Bob")).first().subscribe(findCallback);
    documentFound = findCallback.get().get(0);
    int bobBalance = (int) documentFound.get("balance");
    int newBobBalance = bobBalance + amountToTransfer;

    updateCallback = new SubscriberLatchWrapper<>();
    accountColl.updateOne(session, new Document("name", "Bob"), new Document("$set",
 new Document("balance", newBobBalance))).subscribe(updateCallback);
    updateCallback.await();

    // check Bob's new balance
    findCallback = new SubscriberLatchWrapper<>();
    accountColl.find(session, new Document("name",
 "Bob")).first().subscribe(findCallback);
    documentFound = findCallback.get().get(0);
    updatedBalance = (int) documentFound.get("balance");
    Assert.assertEquals(updatedBalance, newBobBalance);

    // commit the transaction
    SubscriberLatchWrapper<Void> transactionCallback = new
 SubscriberLatchWrapper<>();
    session.commitTransaction().subscribe(transactionCallback);
    transactionCallback.await();
}

public class SubscriberLatchWrapper<T> implements Subscriber<T> {

    /**
     * A Subscriber that stores the publishers results and provides a latch so can
 block on completion.
     *
     * @param <T> The publishers result type
     */
    private final List<T> received;
    private final List<RuntimeException> errors;
```

```
    private final CountDownLatch latch;
    private volatile Subscription subscription;
    private volatile boolean completed;

    /**
     * Construct an instance
     */
    public SubscriberLatchWrapper() {
        this.received = new ArrayList<>();
        this.errors = new ArrayList<>();
        this.latch = new CountDownLatch(1);
    }

    @Override
    public void onSubscribe(final Subscription s) {
        subscription = s;
        subscription.request(Integer.MAX_VALUE);
    }

    @Override
    public void onNext(final T t) {
        received.add(t);
    }

    @Override
    public void onError(final Throwable t) {
        if (t instanceof RuntimeException) {
            errors.add((RuntimeException) t);
        } else {
            errors.add(new RuntimeException("Unexpected exception", t));
        }
        onComplete();
    }

    @Override
    public void onComplete() {
        completed = true;
        subscription.cancel();
        latch.countDown();
    }

    /**
     * Get received elements
     *
```

```
     * @return the list of received elements
     */
    public List<T> getReceived() {
        return received;
    }

    /**
     * Get received elements.
     *
     * @return the list of receive elements
     */
    public List<T> get() {
        return await().getReceived();
    }

    /**
     * Await completion or error
     *
     * @return this
     */
    public SubscriberLatchWrapper<T> await() {
        subscription.request(Integer.MAX_VALUE);
        try {
            if (!latch.await(300, TimeUnit.SECONDS)) {
                throw new MongoTimeoutException("Publisher onComplete timed out for
300 seconds");
            }
        } catch (InterruptedException e) {
            throw new MongoInterruptedException("Interrupted waiting for
observeration", e);
        }
        if (!errors.isEmpty()) {
            throw errors.get(0);
        }
        return this;
    }

    public boolean getCompleted() {
        return this.completed;
    }

    public void close() {
        subscription.cancel();
        received.clear();
```

```
    }
}
```

C

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with C.

```
// Sample C code with core session

bool core_session(mongoc_client_session_t *client_session, mongoc_collection_t*
 collection, bson_t *selector, int64_t balance){
    bool r = true;
    bson_error_t error;
    bson_t *opts = bson_new();
    bson_t *update = BCON_NEW ("$set", "{", "balance", BCON_INT64 (balance), "}");

    // set read & write concern
    mongoc_read_concern_t *read_concern = mongoc_read_concern_new ();
    mongoc_write_concern_t *write_concern = mongoc_write_concern_new ();
    mongoc_transaction_opt_t *txn_opts = mongoc_transaction_opts_new ();

    mongoc_write_concern_set_w(write_concern, MONGOC_WRITE_CONCERN_W_MAJORITY);
    mongoc_read_concern_set_level(read_concern, MONGOC_READ_CONCERN_LEVEL_SNAPSHOT);
    mongoc_transaction_opts_set_write_concern (txn_opts, write_concern);
    mongoc_transaction_opts_set_read_concern (txn_opts, read_concern);

    mongoc_client_session_start_transaction (client_session, txn_opts, &error);
    mongoc_client_session_append (client_session, opts, &error);

    r = mongoc_collection_update_one (collection, selector, update, opts, NULL,
 &error);

    mongoc_client_session_commit_transaction (client_session, NULL, &error);
    bson_destroy (opts);
    mongoc_transaction_opts_destroy(txn_opts);
    mongoc_read_concern_destroy(read_concern);
    mongoc_write_concern_destroy(write_concern);
    bson_destroy (update);
    return r;
}
```

```
void test_core_money_transfer(mongoc_client_t* client, mongoc_collection_t*
 collection, int amount_to_transfer){

    bson_t reply;
    bool r = true;
    const bson_t *doc;
    bson_iter_t iter;
    bson_error_t error;

    // find query
    bson_t *alice_query = bson_new ();
    BSON_APPEND_UTF8(alice_query, "name", "Alice");

    bson_t *bob_query = bson_new ();
    BSON_APPEND_UTF8(bob_query, "name", "Bob");

    // create session
    // set causal consistency to false
    mongoc_session_opt_t *session_opts = mongoc_session_opts_new ();
    mongoc_session_opts_set_causal_consistency (session_opts, false);
    // start the session
    mongoc_client_session_t *client_session = mongoc_client_start_session (client,
 session_opts, &error);

    // add session to options
    bson_t *opts = bson_new();
    mongoc_client_session_append (client_session, opts, &error);

    // deduct 500 from Alice
    // find account balance of Alice
    mongoc_cursor_t *cursor = mongoc_collection_find_with_opts (collection,
 alice_query, NULL, NULL);
    mongoc_cursor_next (cursor, &doc);
    bson_iter_init (&iter, doc);
    bson_iter_find (&iter, "balance");
    int64_t alice_balance = (bson_iter_value (&iter))->value.v_int64;
    assert(alice_balance >= amount_to_transfer);
    int64_t new_alice_balance = alice_balance - amount_to_transfer;

    // core
    r = core_session (client_session, collection, alice_query, new_alice_balance);
    assert(r);

    // find account balance of Alice after transaction
```

```
    cursor = mongoc_collection_find_with_opts (collection, alice_query, NULL, NULL);
    mongoc_cursor_next (cursor, &doc);
    bson_iter_init (&iter, doc);
    bson_iter_find (&iter, "balance");
    alice_balance = (bson_iter_value (&iter))->value.v_int64;
    assert(alice_balance == new_alice_balance);
    assert(alice_balance == 500);

    // add 500 to Bob's balance
    // find account balance of Bob
    cursor = mongoc_collection_find_with_opts (collection, bob_query, NULL, NULL);
    mongoc_cursor_next (cursor, &doc);
    bson_iter_init (&iter, doc);
    bson_iter_find (&iter, "balance");
    int64_t bob_balance = (bson_iter_value (&iter))->value.v_int64;
    int64_t new_bob_balance = bob_balance + amount_to_transfer;

    //core
    r = core_session (client_session, collection, bob_query, new_bob_balance);
    assert(r);

    // find account balance of Bob after transaction
    cursor = mongoc_collection_find_with_opts (collection, bob_query, NULL, NULL);
    mongoc_cursor_next (cursor, &doc);
    bson_iter_init (&iter, doc);
    bson_iter_find (&iter, "balance");
    bob_balance = (bson_iter_value (&iter))->value.v_int64;
    assert(bob_balance == new_bob_balance);
    assert(bob_balance == 1500);

    // cleanup
    bson_destroy(alice_query);
    bson_destroy(bob_query);
    mongoc_client_session_destroy(client_session);
    bson_destroy(opts);
    mongoc_cursor_destroy(cursor);
    bson_destroy(doc);
}

int main(int argc, char* argv[]) {
    mongoc_init ();
    mongoc_client_t* client = mongoc_client_new (<connection uri>);
    bson_error_t error;
```

```
    // connect to bank db
    mongoc_database_t *database = mongoc_client_get_database (client, "bank");
    // access account collection
    mongoc_collection_t* collection = mongoc_client_get_collection(client, "bank",
 "account");
    // set amount to transfer
    int64_t amount_to_transfer = 500;
    // delete the collection if already existing
    mongoc_collection_drop(collection, &error);

    // open Alice account
    bson_t *alice_account = bson_new ();
    BSON_APPEND_UTF8(alice_account, "name", "Alice");
    BSON_APPEND_INT64(alice_account, "balance", 1000);

    // open Bob account
    bson_t *bob_account = bson_new ();
    BSON_APPEND_UTF8(bob_account, "name", "Bob");
    BSON_APPEND_INT64(bob_account, "balance", 1000);

    bool r = true;

    r = mongoc_collection_insert_one(collection, alice_account, NULL, NULL, &error);
    if (!r) {printf("Error encountered:%s", error.message);}
    r = mongoc_collection_insert_one(collection, bob_account, NULL, NULL, &error);
    if (!r) {printf("Error encountered:%s", error.message);}

    test_core_money_transfer(client, collection, amount_to_transfer);

}
```

Scala

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Scala.

```
// Scala Core API
def transferMoneyWithRetry(sessionObservable: SingleObservable[ClientSession] ,
 database: MongoDatabase ): Unit = {
    val accountColl = database.getCollection("account")
    var amountToTransfer = 500
```

```
    var transactionObservable: Observable[ClientSession] =
 sessionObservable.map(clientSession => {
    clientSession.startTransaction()

    // deduct $500 from Alice's account
    var aliceBalance = accountColl.find(clientSession, Document("name" ->
 "Alice")).await().head.getInteger("balance")
    assert(aliceBalance >= amountToTransfer)
    var newAliceBalance = aliceBalance - amountToTransfer
    accountColl.updateOne(clientSession, Document("name" -> "Alice"),
 Document("$set" -> Document("balance" -> newAliceBalance))).await()
    aliceBalance = accountColl.find(clientSession, Document("name" ->
 "Alice")).await().head.getInteger("balance")
    assert(aliceBalance == newAliceBalance)

    // add $500 to Bob's account
    var bobBalance = accountColl.find(clientSession, Document("name" ->
 "Bob")).await().head.getInteger("balance")
    var newBobBalance = bobBalance + amountToTransfer
    accountColl.updateOne(clientSession, Document("name" -> "Bob"), Document("$set"
 -> Document("balance" -> newBobBalance))).await()
    bobBalance = accountColl.find(clientSession, Document("name" ->
 "Bob")).await().head.getInteger("balance")
    assert(bobBalance == newBobBalance)

    clientSession
    })

    transactionObservable.flatMap(clientSession =>
 clientSession.commitTransaction()).await()
}

def doTransactionWithRetry(): Unit = {
    val client: MongoClient = MongoClientWrapper.getMongoClient()
    val database: MongoDatabase = client.getDatabase("bank")
    val accountColl = database.getCollection("account")
    accountColl.drop().await()

    val sessionOptions =
 ClientSessionOptions.builder().causallyConsistent(false).build()
    var  sessionObservable: SingleObservable[ClientSession] =
 client.startSession(sessionOptions)
    accountColl.insertOne(Document("name" -> "Alice", "balance" -> 1000)).await()
    accountColl.insertOne(Document("name" -> "Bob", "balance" -> 1000)).await()
```

```
    var retry = true
    while (retry) {
        try {
        transferMoneyWithRetry(sessionObservable, database)
        println("transaction committed")
        retry = false
        }
        catch {
        case e: MongoException if
 e.hasErrorLabel(MongoException.TRANSIENT_TRANSACTION_ERROR_LABEL) => {
            println("retrying transaction")
        }
        case other: Throwable => {
            println("transaction failed")
            retry = false
            throw other

        }
        }
    }

    // check results outside of transaction
    assert(accountColl.find(Document("name" ->
 "Alice")).results().head.getInteger("balance") == 500)
    assert(accountColl.find(Document("name" ->
 "Bob")).results().head.getInteger("balance") == 1500)

    accountColl.drop().await()

}
```

Python

The following code demonstrates how to utilize the Amazon DocumentDB transaction API with Python.

```
// Sample Python code with Core api

import pymongo

client = pymongo.MongoClient(<connection_string>)
rc_snapshot = pymongo.read_concern.ReadConcern('snapshot')
```

```
wc_majority = pymongo.write_concern.WriteConcern('majority')

# To start, drop and create an account collection and insert balances for both Alice
 and Bob
collection = client.get_database("bank").get_collection("account")
collection.drop()
collection.insert_one({"_id": 1, "name": "Alice", "balance": 1000})
collection.insert_one({"_id": 2, "name": "Bob", "balance": 1000})

amount_to_transfer = 500

# deduct 500 from Alice's account
alice_balance = collection.find_one({"name": "Alice"}).get("balance")
assert alice_balance >= amount_to_transfer
new_alice_balance = alice_balance - amount_to_transfer

with client.start_session({'causalConsistency':False}) as session:
    session.start_transaction(read_concern=rc_snapshot, write_concern=wc_majority)
    collection.update_one({"name": "Alice"}, {'$set': {"balance":
 new_alice_balance}}, session=session)
    session.commit_transaction()

updated_alice_balance = collection.find_one({"name": "Alice"}).get("balance")
assert updated_alice_balance == new_alice_balance

# add 500 to Bob's account
bob_balance = collection.find_one({"name": "Bob"}).get("balance")
assert bob_balance >= amount_to_transfer
new_bob_balance = bob_balance + amount_to_transfer

with client.start_session({'causalConsistency':False}) as session:
    session.start_transaction(read_concern=rc_snapshot, write_concern=wc_majority)
    collection.update_one({"name": "Bob"}, {'$set': {"balance": new_bob_balance}},
 session=session)
    session.commit_transaction()

updated_bob_balance = collection.find_one({"name": "Bob"}).get("balance")
assert updated_bob_balance == new_bob_balance
```

Supported commands

Command	Supported
abortTransaction	Yes
commitTransaction	Yes
endSessions	Yes
killSession	Yes
killAllSession	Yes
killAllSessionsByPattern	No
refreshSessions	No
startSession	Yes

Unsupported capabilities

Methods	Stages or Commands
db.collection.aggregate()	$collStats $currentOp $indexStats $listSessions $out
db.collection.count() db.collection.countDocuments()	$where $near $nearSphere

Methods	Stages or Commands
`db.collection.insert()`	`insert` is not supported if it is not run against an existing collection. This method is supported if it targets a pre-existing collection.

Sessions

MongoDB sessions are a framework that is used to support retryable writes, causal consistency, transactions, and manage operations across databases. When a session is created, a logical session identifier (lsid) is generated by the client and is used to tag all operations within that session when sending commands to the server.

Amazon DocumentDB supports the use of sessions to enable transactions, but does not support causal consistency. Starting with engine version 8.0.2, Amazon DocumentDB supports retryable writes outside of transactions for write operations in which each individual write affects at most one document. On earlier engine versions, retryable writes are not supported. For more information, see Retryable writes in Amazon DocumentDB.

When utilizing transactions within Amazon DocumentDB, a transaction will be initiated from within a session using the `session.startTransaction()` API and a session supports a single transaction at a time. Similarly, transactions are completed using either the commit (`session.commitTransaction()`) or abort (`session.abortTransaction()`) APIs.

Causal consistency

Causal consistency guarantees that within a single client session the client will observe read-after-write consistency, monatomic reads/writes, and writes will follow reads and these guarantees apply across all instances in a cluster, not just the primary. Amazon DocumentDB does not support causal consistency and the following statement will result in an error.

```
var mySession = db.getMongo().startSession();
var mySessionObject = mySession.getDatabase('test').getCollection('account');

mySessionObject.updateOne({"_id": 2}, {"$inc": {"balance": 400}});
//Result:{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

```
mySessionObject.find()
//Error: error: {
//         "ok" : 0,
//         "code" : 303,
//         "errmsg" : "Feature not supported: 'causal consistency'",
//         "operationTime" : Timestamp(1603461817, 493214)
//}

mySession.endSession()
```

You can disable causal consistency within a session. Note that doing so will enable you to utilize the session framework, but will not provide causal consistency guarantees for reads. When using Amazon DocumentDB, reads from the primary will be read-after-write consistent and reads from the replica instances will be eventually consistent. Transactions are the primary use case for utilizing sessions.

```
var mySession = db.getMongo().startSession({causalConsistency: false});
var mySessionObject = mySession.getDatabase('test').getCollection('account');

mySessionObject.updateOne({"_id": 2}, {"$inc": {"balance": 400}});
//Result:{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

mySessionObject.find()
//{ "_id" : 1, "name" : "Bob", "balance" : 100 }
//{ "_id" : 2, "name" : "Alice", "balance" : 1700 }
```

Retryable writes

With retryable writes, your driver automatically retries a write operation one time if a network error occurs or the driver can't find the primary. Starting with engine version 8.0.2, Amazon DocumentDB supports retryable writes. On earlier engine versions, retryable writes must be disabled by setting `retryWrites=false` in the connection string. For more information, see Retryable writes in Amazon DocumentDB.

Transaction errors

When using transactions, there are scenarios that can yield an error that states that a transaction number does not match any in progress transaction.

The error can be generated in at least two different scenarios:

- After the one-minute transaction timeout.
- After an instance restart (due to patching, crash recovery, etc.), it is possible to receive this error even in cases where the transaction successfully committed. During an instance restart, the database can't tell the difference between a transaction that successfully completed versus a transaction that aborted. In other words, the transaction completion state is ambiguous.

The best way to handle this error is to make transactional updates idempotent -- for example, by using the `$set` mutator instead of an increment/decrement operation. See below:

```
{ "ok" : 0,
"operationTime" : Timestamp(1603938167, 1),
"code" : 251,
"errmsg" : "Given transaction number 1 does not match any in-progress transactions."
}
```

Retryable writes in Amazon DocumentDB

Starting with engine version 8.0.2, Amazon DocumentDB supports retryable writes. When a write operation fails due to a transient network error or primary election, the driver can automatically retry the operation exactly once. Amazon DocumentDB deduplicates the retried write so that the operation is applied at most once, preserving idempotency.

Retryable writes require a MongoDB-compatible driver that supports the retryable writes protocol. Most current MongoDB-compatible drivers enable retryable writes by default (`retryWrites=true` in the connection string). If you are upgrading to engine 8.0.2 from an earlier version, you can remove `retryWrites=false` from your connection string to enable this feature.

Requirements

To use retryable writes, you must meet the following requirements:

- Amazon DocumentDB engine version 8.0.2 or later.
- A MongoDB-compatible driver that supports retryable writes. For the minimum version, see the documentation for your driver.
- The connection string must include `retryWrites=true`, or the driver must default to retryable writes (most current drivers do).

Supported operations

Retryable writes apply to write operations in which each individual write affects at most one document. Batch operations qualify, because each write within the batch is deduplicated separately.

The following write operations are retryable:

- `insertOne`
- `insertMany`
- `updateOne`
- `deleteOne`
- `findOneAndUpdate`
- `findOneAndDelete`
- `findOneAndReplace`
- `bulkWrite` (when composed of `insertOne`, `updateOne`, `deleteOne`, or `replaceOne` operations)

For `insertMany` and `bulkWrite`, each document in the batch is deduplicated individually. If a retry is needed, only the documents that were not yet applied are inserted.

> **Note**
>
> `commitTransaction` and `abortTransaction` are also retryable, as transaction-control commands rather than as writes. The individual writes inside a transaction are not retryable. For more information, see Limitations.

Limitations

The following limitations apply to retryable writes in Amazon DocumentDB:

- `updateMany` and `deleteMany` are not retryable.
- Writes inside multi-statement transactions are not retryable. Transaction commit and abort operations are retryable separately.

- Documents must include an `_id` field for retryable inserts.
- Retryable writes are available only on engine version 8.0.2 and later. On earlier engine versions, set `retryWrites=false` in your connection string to avoid errors.
- Write operations are not retryable on clusters that use query planner version 1.0.
- For `findOneAndUpdate`, `findOneAndDelete`, and `findOneAndReplace` operations that return large documents, retryable writes may increase write latency because the full result document is cached for deduplication. If write performance is critical and the returned documents are large, consider setting `retryWrites=false` for those workloads.

Enabling retryable writes

To enable retryable writes, include `retryWrites=true` in your connection string:

```
mongodb://<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=true
```

If your driver defaults to `retryWrites=true`, you can remove any explicit `retryWrites=false` from your connection string.

For the full connection string walkthrough, see Connecting programmatically to Amazon DocumentDB.

How retryable writes work

When your application sends a write with `retryWrites=true`, the driver attaches a logical session ID (`lsid`) and a transaction number (`txnNumber`) to the write command. Amazon DocumentDB uses these identifiers to deduplicate writes:

- On the first attempt, Amazon DocumentDB executes the write and caches the result.
- If the driver retries the same write (same `lsid` and `txnNumber`), Amazon DocumentDB returns the cached result without re-executing the write.
- Amazon DocumentDB honors retries for 60 minutes after the original write. Drivers retry immediately, so this window is far longer than a driver needs.

If a retry arrives more than 60 minutes after the original write, Amazon DocumentDB no longer has a cached result for it and runs the write as a new operation.

Error handling

The following errors are specific to retryable writes in Amazon DocumentDB:

Error code	Name	Description
225	TransactionTooOld	A newer transaction number for the same session and statement was already committed, so this retry is too old to apply.
301	Retryable writes not supported	The write included retryable write fields, but the engine version does not support retryable writes, or the operation is not retryable. Set `retryWrites=false` when connecting to engine versions earlier than 8.0.2.

Migrating from retryWrites=false

If you are upgrading to engine version 8.0.2 and currently use `retryWrites=false` in your connection strings:

1. Upgrade your cluster to engine version 8.0.2 or later.
2. Remove `retryWrites=false` from your application's connection strings, or change it to `retryWrites=true`.
3. No application code changes are required. The driver handles retries automatically.

Using collation in Amazon DocumentDB 8.0

Amazon DocumentDB 8.0 now supports collation. Collation allows you to configure language-specific rules for string comparison. With collation, you can specify rules for case-sensitive comparisons, or specify a language locale. Collation can be configured at the collection-level or index-level in DocumentDB 8.0. When you use collation in DocumentDB, internally a collation document is created with the following parameters.

```
{
```

```
locale: string,
caseLevel: boolean,
caseFirst: string,
strength: int,
numericOrdering: boolean,
alternate: string,
maxVariable: string,
backwards: boolean,
normalization: boolean
}
```

Limitations

Collation has following limitations in Amazon DocumentDB:

- Collation is compatible with planner v3 available in Amazon DocumentDB 8.0. Switching to plannerv2 or plannerv1 may cause inconsistent behaviors, including “Index not found“ errors.
- Due to inherent library differences, if a mongodb collection with collation is exported, you will need to update the metadata.bson files and change its version from 57.1 to 60.2 prior to migration.
- In rare cases, your collation settings may breach the internal limit on character count, causing the following error. "Error: Collation document has non-default attributes more than supported. Please reduce the number of options." In this case, try to reduce the options you provide in the collation document, or alternatively you could try and use default values wherever possible.

Using views in Amazon DocumentDB 8.0

- Amazon DocumentDB 8.0 now supports views. Views function as virtual collections that present data based on specified aggregation operations. When you create a view, you define a query that transforms data from one or more source collections. Amazon DocumentDB 8.0 executes this query each time the view is accessed, without consuming additional storage resources. Unlike standard collections, views in Amazon DocumentDB 8.0 do not store documents on disk, making them an efficient solution for presenting transformed or filtered data to applications. To create a view in Amazon DocumentDB, you can use the createView command or the db.createView() helper method:

```
db.createView("viewName","sourceCollection",
[
    { $match: { status: "active" } },
    { $project: { _id: 1, name: 1, email: 1 } }
]
)
```

This creates a view named "viewName" based on "sourceCollection" that only includes active documents and projects only the _id, name, and email fields. Views in Amazon DocumentDB are read-only. Write operations on views will return an error. For optimal performance with large datasets, you can structure your view pipelines to maximize efficiency. For complex aggregation pipelines, it's recommended to use the $match stage as the first stage or early in the pipeline to reduce the number of documents subsequent stages need to process, thus improving query performance.

Best practices

Some best practices to follow with Views are listed below.

- Filter Early: Use $match stages early in view pipelines to reduce the amount of data processed.
- Avoid Complex Aggregations: For frequently accessed views with complex aggregations, consider creating a separate collection with precomputed results that are updated periodically.
- Index Planning: Ensure that fields used in view pipelines, especially in $match and $sort operations, are properly indexed on the source collection.
- Query Optimization: Use the explain command to understand how your view queries are executed and optimize accordingly.
- Alternatives for Views: Given the functional differences between Amazon DocumentDB and MongoDB views, consider using regular collections with scheduled updates as an alternative to views when encountering limitations.

Aggregator operator compatibility

Amazon DocumentDB supports many aggregation operators in view definitions while continuing to expand compatibility. When using views, focus on these supported operators:

- $match for filtering documents
- $project for field selection and transformation

- $addFields for adding computed fields
- $sort for ordering results
- $limit and $skip for pagination

Some specialized operators like $currentOp, $replaceRoot, and $geoNear currently work in direct aggregation queries rather than view definitions.

Leveraging indexes and views

Views in Amazon DocumentDB 8.0 use the indexes of the underlying collection. As a result, you cannot create, drop, or rebuild indexes on a view directly. However, well-designed indexes on the source collection can significantly improve view query performance Below are some steps to optimize query performance on views:

- Ensure appropriate indexes exist on the source collection fields used in the view's pipeline, especially in $match and $sort operations
- Use the explain() method to analyze query execution plans and verify index usage. E.g., `db.viewName.find({...}).explain()`

Amazon DocumentDB generative artificial intelligence

Amazon DocumentDB offers capabilities to enable machine learning (ML) and generative artificial intelligence (AI) models to work with data stored in Amazon DocumentDB in real time. Customers no longer have to spend time managing separate infrastructure, writing code to connect with another service, and duplicating data from their primary database.

For more information on artificial intelligence and how AWS can support your AI needs, see this "What-is" article.

Topics

- No-code machine learning with Amazon SageMaker AI Canvas
- Vector search for Amazon DocumentDB

No-code machine learning with Amazon SageMaker AI Canvas

Amazon SageMaker AI Canvas enables you to build your own AI/ML models without having to write a single line of code. You can build ML models for common use cases such as regression and forecasting and can access and evaluate foundation models (FMs) from Amazon Bedrock. You can also access public FMs from Amazon SageMaker AI JumpStart for content generation, text extraction, and text summarization to support generative AI solutions.

How to build no-code ML models with SageMaker AI Canvas

Amazon DocumentDB now integrates with Amazon SageMaker AI Canvas to enable no-code machine learning (ML) with data stored in Amazon DocumentDB. You can now build ML models for regression and forecasting needs and use foundation models for content summarization and generation using data stored in Amazon DocumentDB without writing a single line of code.

SageMaker AI Canvas provides a visual interface that allows Amazon DocumentDB customers to generate predictions without requiring any AI/ML expertise or write a single line of code. Customers can now launch the SageMaker AI Canvas workspace from the AWS Management Console, import and join Amazon DocumentDB data for data preparation and model training. Data in Amazon DocumentDB can now be used in SageMaker AI Canvas to build and augment models to predict customer churn, detect fraud, predict maintenance failures, forecast business metrics, and generate content. Customers can now publish and share ML-driven insights across teams using SageMaker AI Canvas's native integration with Quick. Data ingestion pipelines in SageMaker AI Canvas run on Amazon DocumentDB secondary instances by default, ensuring that the performance of application and SageMaker AI Canvas ingestion workloads are not hindered.

Amazon DocumentDB customers can get started with SageMaker AI Canvas by navigating to the new Amazon DocumentDB No-Code ML Console page and connecting to new or available SageMaker AI Canvas workspaces.

Configuring the SageMaker AI domain and user profile

You can connect to Amazon DocumentDB clusters from SageMaker AI domains that are running in VPC Only mode. By launching a SageMaker AI domain in your VPC, you can control the data flow from your SageMaker AI Studio and Canvas environments. This allows you to restrict internet access, monitor and inspect traffic using standard AWS networking and security capabilities, and connect to other AWS resources through VPC endpoints. Refer to Amazon SageMaker AI Canvas Getting started and Configure Amazon SageMaker AI Canvas in a VPC without internet access

located in the *Amazon SageMaker AI Developer Guide* to create your SageMaker AI domain to connect to your Amazon DocumentDB cluster.

Configuring IAM access permissions for Amazon DocumentDB and SageMaker AI Canvas

An Amazon DocumentDB user that has `AmazonDocDBConsoleFullAccess` attached to their associated role and identity can access the AWS Management Console. Add the following actions to the aforementioned role or identity to provide access to no-code machine learning with Amazon SageMaker AI Canvas.

```
"sagemaker:CreatePresignedDomainUrl",
"sagemaker:DescribeDomain",
"sagemaker:ListDomains",
"sagemaker:ListUserProfiles"
```

Creating database users and roles for SageMaker AI Canvas

You can restrict access to the actions that users can perform on databases using role-based access control (RBAC) in Amazon DocumentDB. RBAC works by granting one or more roles to a user. These roles determine the operations that a user can perform on database resources.

As a Canvas user, you connect to a Amazon DocumentDB database with username and password credentials. You can create a database user/role for a Canvas user that has read access to the specific databases using Amazon DocumentDB RBAC functionality.

For example, use the `createUser` operation:

```
db.createUser({
user: "canvas_user",
pwd: "<insert-password>",
roles: [{role: "read", db: "sample-database-1"}]
})
```

This creates a `canvas_user` which has read permissions to the `sample-database-1` database. Your Canvas analysts can use this credential to access data in your Amazon DocumentDB cluster. Refer to Database access using Role-Based Access Control to learn more.

Available Regions

The no-code integration is available in Regions where both Amazon DocumentDB and Amazon SageMaker AI Canvas are supported. The Regions include:

- us-east-1 (N. Virginia)
- us-east-2 (Ohio)
- us-west-2 (Oregon)
- ap-northeast-1 (Tokyo)
- ap-northeast-2 (Seoul)
- ap-south-1 (Mumbai)
- ap-southeast-1 (Singapore)
- ap-southeast-2 (Sydney)
- eu-central-1 (Frankfurt)
- eu-west-1 (Ireland)

Refer to Amazon SageMaker AI Canvas in the *Amazon SageMaker AI Developer Guide* for the latest Region availability.

Vector search for Amazon DocumentDB

Vector search is a method used in machine learning to find similar data points to a given data point by comparing their vector representations using distance or similarity metrics. The closer the two vectors are in the vector space, the more similar the underlying items are considered to be. This technique helps capture the semantic meaning of the data. This approach is useful in various applications, such as recommendation systems, natural language processing, and image recognition.

Vector search for Amazon DocumentDB combines the flexibility and rich querying capability of a JSON-based document database with the power of vector search. If you want to use your existing Amazon DocumentDB data or a flexible document data structure to build machine learning and generative AI use cases, such as semantic search experience, product recommendation, personalization, chatbots, fraud detection, and anomaly detection, then vector search for Amazon DocumentDB is an ideal choice for you. Vector search is available on Amazon DocumentDB 5.0 instance-based clusters.

Topics

- Inserting vectors
- Creating a vector index
- Getting an index definition
- Querying vectors
- Features and limitations
- Best practices

Inserting vectors

To insert vectors into your Amazon DocumentDB database, you can use existing insert methods:

Example

In the following example, a collection of five documents within a test database is created. Each document includes two fields: the product name and its corresponding vector embedding.

```
db.collection.insertMany([
  {"product_name": "Product A", "vectorEmbedding": [0.2, 0.5, 0.8]},
  {"product_name": "Product B", "vectorEmbedding": [0.7, 0.3, 0.9]},
  {"product_name": "Product C", "vectorEmbedding": [0.1, 0.2, 0.5]},
  {"product_name": "Product D", "vectorEmbedding": [0.9, 0.6, 0.4]},
  {"product_name": "Product E", "vectorEmbedding": [0.4, 0.7, 0.2]}
]);
```

Creating a vector index

Amazon DocumentDB supports both Hierarchical Navigable Small World (HNSW) indexing and Inverted File with Flat Compression (IVFFlat) indexing methods. An IVFFlat index segregates vectors into lists and subsequently searches a selected subset of those lists that are nearest to the query vector. On the other hand, an HNSW index organizes the vector data into a multi-layered graph. Although HNSW has slower build times compared to IVFFlat, it delivers better query performance and recall. Unlike IVFFlat, HNSW has no training step involved, allowing the index to be generated without any initial data load. For most use cases, use the HNSW index type for vector search.

If you do not create a vector index, Amazon DocumentDB performs an exact nearest neighbor search, ensuring perfect recall. However, in production scenarios, speed is crucial. Use vector

indexes, which might trade some recall for improved speed. It's important to note that adding a vector index can lead to different query results.

Templates

You can use the following `createIndex` or `runCommand` templates to build a vector index on a vector field:

Using createIndex

In certain drivers, such as mongosh and Java, using the `vectorOptions` parameters in `createIndex` might result in an error. In such cases, use `runCommand`:

```
db.collection.createIndex(
  { "<vectorField>": "vector" },
  { "name": "<indexName>",
    "vectorOptions": {
      "type": " <hnsw> | <ivfflat> ",
      "dimensions": <number_of_dimensions>,
      "similarity": " <euclidean> | <cosine> | <dotProduct> ",
      "lists": <number_of_lists> [applicable for IVFFlat],
      "m": <max number of connections> [applicable for HNSW],
      "efConstruction": <size of the dynamic list for index build> [applicable for
 HNSW]
    }
  }
);
```

Using runCommand

In certain drivers, such as mongosh and Java, using the `vectorOptions` parameters in `createIndex` might result in an error. In such cases, use `runCommand`:

```
db.runCommand(
  { "createIndexes": "<collection>",
  "indexes": [{
      key: { "<vectorField>": "vector" },
      vectorOptions: {
          type: " <hnsw> | <ivfflat> ",
          dimensions: <number of dimensions>,
          similarity: " <euclidean> | <cosine> | <dotProduct> ",
          lists: <number_of_lists> [applicable for IVFFlat],
          m: <max number of connections> [applicable for HNSW],
```

```
          efConstruction: <size of the dynamic list for index build> [applicable for
 HNSW]
          },
      name: "myIndex"
      }]
  }
);
```

Parameter	Requirement	Data type	Description	Value(s)
name	optional	string	Specifies the name of the index.	Alphanumeric
type	optional		Specifies the type of index.	Supported: hnsw or ivfflat Default: HNSW (engine patch 3.0.4574 onwards)
dimensions	required	integer	Specifies the number of dimensions in the vector data.	Maximum of 2,000 dimension s.
similarity	required	string	Specifies the distance metric used for the similarity calculation.	• **euclidean** • **cosine** • **dotProduct**
lists	required for IVFFlat	integer	Specifies the number of clusters that the IVFFlat index uses to group	Minimum: 1 Maximum: Refer to the lists per instance type

Parameter	Requirement	Data type	Description	Value(s)
			the vector data. The recommended setting is the # of documents /1000 for up to 1M documents and `sqrt(# of documents)` for over 1M documents.	table in Features and limitations below.
`m`	optional	integer	Specifies the max number of connections for an HNSW index	Default: 16 Range [2, 100]
`efConstruction`	optional	integer	Specifies the size of the dynamic candidate list for constructing the graph for HNSW index. `efConstruction` must be greater than or equal to (2 * m)	Default: 64 Range [4, 1000]

It is important that you set the value of sub-parameters such as `lists` for IVFFlat and `m` and `efConstruction` for HNSW appropriately as it will affect the accuracy/recall, build time, and performance of your search. A higher list value increases the speed of the query as it reduces the number of vectors in each list, resulting in smaller regions. However, a smaller region size may lead to more recall errors, resulting in lower accuracy. For HNSW, increasing the value of `m` and `efConstruction` increases the accuracy, but also increases index build time and size. See the following examples:

Examples

HNSW

```
db.collection.createIndex(
  { "vectorEmbedding": "vector" },
  { "name": "myIndex",
    "vectorOptions": {
      "type": "hnsw",
      "dimensions": 3,
      "similarity": "euclidean",
      "m": 16,
      "efConstruction": 64
    }
  }
);
```

IVFFlat

```
db.collection.createIndex(
  { "vectorEmbedding": "vector" },
  { "name": "myIndex",
    "vectorOptions": {
      "type": "ivfflat",
      "dimensions": 3,
      "similarity": "euclidean",
      "lists":1
    }
  }
)
```

Getting an index definition

You can view the details of your indexes, including vector indexes, using the `getIndexes` command:

Example

```
db.collection.getIndexes()
```

Example output

```
[
 {
  "v" : 4,
  "key" : {
   "_id" : 1
  },
  "name" : "_id_",
  "ns" : "test.collection"
 },
 {
  "v" : 4,
  "key" : {
   "vectorEmbedding" : "vector"
  },
  "name" : "myIndex",
  "vectorOptions" : {
   "type" : "ivfflat",
   "dimensions" : 3,
   "similarity" : "euclidean",
   "lists" : 1
  },
  "ns" : "test.collection"
 }
]
```

Querying vectors

Amazon DocumentDB supports two vector search operators for querying vectors:

Classic vector search operator

Use the following template to query a vector:

```
db.collection.aggregate([
  {
    $search: {
      "vectorSearch": {
        "vector": <query vector>,
        "path": "<vectorField>",
        "similarity": "<distance metric>",
        "k": <number of results>,
        "probes":<number of probes> [applicable for IVFFlat],
        "efSearch":<size of the dynamic list during search> [applicable for HNSW]
```

```
      }
    }
  }
]);
```

Parameter	Requirement	Type	Description	Value(s)
vectorSearch	required	operator	Used inside $search command to query the vectors.	
vector	required	array	Indicates the query vector that will be used to find similar vectors.	
path	required	string	Defines the name of the vector field.	
k	required	integer	Specifies the number of results that the search returns.	
similarity	required	string	Specifies the distance metric used for the similarity calculation.	• **euclidean** • **cosine** • **dotProduct**
probes	optional	integer	The number of clusters you want vector search to	Default: 1

Parameter	Requirement	Type	Description	Value(s)
			inspect. A higher value provides better recall at the cost of speed. It can be set to the number of lists for exact nearest neighbor search (at which point the planner won't use the index). The recommended setting to start fine-tuning is `sqrt(# of lists)`.	
`efSearch`	optional	integer	Specifies the size of the dynamic candidate list that HNSW index uses during search. A higher value of `efSearch` provides better recall at cost of speed.	Default: 40 Range [1, 1000]

It is important to fine tune the value of `efSearch` (HNSW) or `probes` (IVFFlat) to achieve your desired performance and accuracy. See the following example operations:

HNSW

```
db.collection.aggregate([
  {
    $search: {
      "vectorSearch": {
        "vector": [0.2, 0.5, 0.8],
        "path": "vectorEmbedding",
        "similarity": "euclidean",
        "k": 2,
        "efSearch": 40
      }
    }
  }
]);
```

IVFFlat

```
db.collection.aggregate([
  {
    $search: {
      "vectorSearch": {
        "vector": [0.2, 0.5, 0.8],
        "path": "vectorEmbedding",
        "similarity": "euclidean",
        "k": 2,
        "probes": 1
      }
    }
  }
]);
```

Example output

Output from this operation looks something like the following:

```
{ "_id" : ObjectId("653d835ff96bee02cad7323c"), "product_name" : "Product A",
 "vectorEmbedding" : [ 0.2, 0.5, 0.8 ] }
{ "_id" : ObjectId("653d835ff96bee02cad7323e"), "product_name" : "Product C",
 "vectorEmbedding" : [ 0.1, 0.2, 0.5 ] }
```

`$vectorSearch` operator (available in Amazon DocumentDB 8.0 onwards)

Use the following template to query a vector:

```
db.collection.aggregate([
{
  "$vectorSearch": {
    "exact": true | false,
    "index": "<index-name>" [supports only HNSW index],
    "limit": <number-of-results> [same as k],
    "path": "<vector field-to-search>",
    "queryVector": <array-of-numbers>,
    "numCandidates": <number-of-candidates> [same as efSearch],
  }
}])
```

Features and limitations

Version compatibility

- Vector search for Amazon DocumentDB is only available on Amazon DocumentDB 5.0+ instance-based clusters.

Vectors

- Amazon DocumentDB can index vectors of up to 2,000 dimensions. However, up to 16,000 dimensions can be stored without an index.

Indexes

- For IVFFlat index creation, the recommended setting for lists parameter is the number of documents/1000 for up to 1M documents and `sqrt(# of documents)` for over 1M documents. Due to a working memory limit, Amazon DocumentDB supports a certain maximum value of the lists parameter depending on the number of dimensions. For your reference, the following table provides the maximum values of lists parameter for vectors of 500, 1000, and 2,000 dimensions:

Instance type	Lists with 500 dimensions	Lists with 1000 dimensions	Lists with 2000 dimensions
t3.med	372	257	150
r5.l	915	741	511
r5.xl	1,393	1,196	901
r5.2xl	5,460	5,230	4,788
r5.4xl	7,842	7,599	7,138
r5.8xl	11,220	10,974	10,498
r5.12xl	13,774	13,526	13,044
r5.16xl	15,943	15,694	15,208
r5.24xl	19,585	19,335	18,845

- No other index options such as `compound`, `sparse` or `partial` are supported with vector indexes.
- Parallel index build is not supported for HNSW index in Amazon DocumentDB 5.0.

Vector query

- For vector search query, it is important to fine tune the parameters such as `probes` or `efSearch` for optimum results. The higher the value of `probes` or `efSearch` parameter, the higher the recall and lower the speed. The recommended setting to start fine tuning the probes parameter is `sqrt(# of lists)`.

Best practices

Learn best practices for working with vector search in Amazon DocumentDB. This section is continually updated as new best practices are identified.

- Inverted File with Flat Compression (IVFFlat) index creation involves clustering and organizing the data points based on similarities. For an index to be more effective, load some data before creating the index.
- For vector search queries, it is important to fine tune the parameters such as `probes` or `efSearch` for optimum results. The higher the value of the `probes` or `efSearch` parameter, the higher is the recall and lower is the speed. The recommended setting to start fine tuning the `probes` parameter is `sqrt(lists)`.

Resources

- Vector search for Amazon DocumentDB is now generally available on the AWS Blog
- Semantic search code sample
- Amazon DocumentDB vector search code samples

Using AWS Lambda with change streams

Amazon DocumentDB is integrated with AWS Lambda so you can use Lambda functions to process records in a change stream. Lambda event source mapping is a resource that can be used to invoke Lambda functions to process Amazon DocumentDB events that don't directly invoke Lambda. With Amazon DocumentDB change stream as an event source, you can build event-driven applications that respond to changes in your data. For example, you can use Lambda functions to process new documents, track updates to existing documents, or log deleted documents.

You can configure an event source mapping to send records from your Amazon DocumentDB change stream to a Lambda function. Events can be sent one at a time or batched for improved efficiency and will processed in-order. You can configure your event source mapping's batching behavior based on a specific time window duration (0 - 300 sec) or batch record count (max limit of 10,000 records). You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process distinct items from multiple streams with a single function.

If your function returns an error, Lambda retries the batch until it processes successfully. In case the events in the change stream have expired, Lambda will disable the event source mapping. In this case, you can create a new event source mapping and configure it with a starting position of your choice. Lambda event source mappings process events at least once due to the distributed nature of its pollers. As a result, your Lambda function may receive duplicate events in rare situations.

Follow best practices for working with AWS Lambda functions and build idempotent functions to avoid issues related to duplicate events. For more information, see Using AWS Lambda console with Amazon DocumentDB in the *AWS Lambda Developer Guide*.

As performance best practices, the Lambda function needs to be short lived. To avoid introducing unnecessary processing delays, it also should not execute complex logic. For a high velocity stream in particular, it is better to trigger an asynchronous post-processing step function workflows than synchronous long running Lambdas. For more information about AWS Lambda, see the AWS Lambda Developer Guide.

Limitations

The following are limitations to consider when working with Amazon DocumentDB and AWS Lambda:

- AWS Lambda event source mapping is not supported on Amazon DocumentDB 3.6.
- AWS Lambda event source mapping is not supported on elastic clusters.
- AWS Lambda payload sizes cannot exceed 6MB. For more information about Lambda batch sizes, see “Batching behavior” in Lambda event source mappings section in the *AWS Lambda Developer Guide*.

Amazon DocumentDB compatibility with MongoDB

Amazon DocumentDB supports MongoDB compatibility including MongoDB 4.0, MongoDB 5.0, and MongoDB 8.0. MongoDB compatibility means that a vast majority of the applications, drivers, and tools you already use today with your MongoDB databases can be used with Amazon DocumentDB with little or no change. This section describes everything you need to know about Amazon DocumentDB compatibility with MongoDB including new capabilities and features, getting started, migrations paths, and functional differences.

Topics

- MongoDB 8.0 compatibility
- MongoDB 5.0 compatibility
- MongoDB 4.0 compatibility
- Functional differences: Amazon DocumentDB and MongoDB
- Supported MongoDB APIs, operations, and data types in Amazon DocumentDB

MongoDB 8.0 compatibility

Topics

- What's new in Amazon DocumentDB 8.0
- Get started with Amazon DocumentDB 8.0
- Upgrade or migrate to Amazon DocumentDB 5.0 or 8.0
- Functional differences

What's new in Amazon DocumentDB 8.0

Amazon DocumentDB 8.0 improves query performance by up to 7x, and improves compression ratio by up to 5x, enabling you to build high-performance applications at a lower cost. Amazon DocumentDB 8.0 has full wire protocol compatibility with MongoDB 8.0. The summary below introduces some of major features that were introduced in Amazon DocumentDB 8.0. To see a full list of the new capabilities, see the Release notes.

- Provides compatibility with MongoDB 8.0 by adding support for MongoDB 8.0 API drivers. Also supports applications that are built using MongoDB API versions 6.0 and 7.0.

- New query planner in Amazon DocumentDB 8.0 extends performance improvements to aggregation stage operators, along with supporting aggregation pipeline optimizations and distinct commands.
- Amazon DocumentDB 8.0 supports collation and views.
- Offers 6 new aggregation stages: $replaceWith, $vectorSearch, $merge, $set, $unset, $bucket, and 3 new aggregation operators $pow, $rand, $dateTrunc.
- A new version of text index: Text index v2 in Amazon DocumentDB 8.0 introduces additional tokens, enhancing text search capabilities.
- Through parallel vector index build, Amazon DocumentDB 8.0 reduces index build time by up to 30x.
- Increased storage limit to 256 TiB.

Get started with Amazon DocumentDB 8.0

To get started with Amazon DocumentDB 8.0, see the Get started with Amazon DocumentDB. You can create a new Amazon DocumentDB 8.0 cluster using the AWS Management Console or the AWS SDK, AWS CLI, or CloudFormation. When connecting to Amazon DocumentDB, it is required that you use a MongoDB driver or utility that is compatible with MongoDB 5.0 or higher.

> **Note**
>
> When using the AWS SDK, AWS CLI, or CloudFormation, the engine version will default to the newest 8.0 minor version available in the AWS Region. To use a specific minor version, specify it with `engineVersion`. For more information, see Amazon DocumentDB cluster settings.
>
> For a given Amazon DocumentDB cluster, you can determine the cluster version using the AWS CLI to call `describe-db-clusters` or use the Amazon DocumentDB management console to view the engine version number for a particular cluster.

Amazon DocumentDB 5.0 and 8.0 supports the Amazon EC2 `r8g` Graviton4 processor and Graviton2 processors such as `r6g` and `t4.medium` instance types for your clusters and is available in all supported AWS Regions (see Instances). For more information on pricing, see Amazon DocumentDB (with MongoDB compatibility) Pricing.

Upgrade or migrate to Amazon DocumentDB 5.0 or 8.0

You can migrate from MongoDB 3.6 or MongoDB 4.0 to Amazon DocumentDB 5.0 or Amazon DocumentDB 8.0 using the AWS DMS or utilities like mongodump, mongorestore, mongoimport, and mongoexport. For instructions on migrating, see Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service.

Functional differences

Functional differences between Amazon DocumentDB 5.0 and 8.0

With the release of Amazon DocumentDB 8.0, there are functional differences between Amazon DocumentDB 5.0 and Amazon DocumentDB 8.0:

- Planner v1 is the default query planner in Amazon DocumentDB 5.0, whereas the more performant Planner v3 is the default in Amazon DocumentDB 8.0.
- New features in Amazon DocumentDB 8.0 including Views, Collation and operators like $merge are compatible only with Planner v3.
- Compression is turned ON by default in Amazon DocumentDB 8.0 and is set to use Zstandard algorithm. Furthermore, in Amazon DocumentDB 8.0, `enabled` is no longer a valid value for `default_collection_compression`. The valid values are `zstd`, `lz4`, and `disabled`.
- Amazon DocumentDB 8.0 only supports TLS 1.2 and higher for database connections.

> **Important**
>
> Amazon DocumentDB 8.0 requires TLS 1.2 or higher. TLS 1.0 and TLS 1.1 are no longer supported.

MongoDB 5.0 compatibility

Topics

- What's new in Amazon DocumentDB 5.0
- What's new in Amazon DocumentDB 5.0.1
- Get started with Amazon DocumentDB 5.0

- Upgrade or migrate to Amazon DocumentDB 5.0
- Functional differences

What's new in Amazon DocumentDB 5.0

Amazon DocumentDB 5.0 introduces new features and capabilities that include storage limits and client-side field level encryption. The summary below introduces some of major features that were introduced in Amazon DocumentDB 5.0. To see a full list of the new capabilities, see the Release notes.

- Increased storage limit to 128 TiB for all instance-based Amazon DocumentDB clusters and shard-based elastic clusters.
- Introduced Amazon DocumentDB 5.0 Engine Version 3.0.775)
  - Support for MongoDB 5.0 API drivers
  - Support for Client-side Field Level Encryption (FLE). You can now encrypt fields at the client-side before writing the data to Amazon DocumentDB cluster. For more information, see Client-side field level encryption.
  - New aggregation operators: `$dateAdd`, `$dateSubtract`
  - Supports for indexes with `$elemMatch` operator. As a result, queries having `$elemMatch` will result in index scans.

Amazon DocumentDB does not support every MongoDB 5.0 feature. When we built Amazon DocumentDB 5.0, we worked backwards from the feature and capabilities that our customers asked us to build the most. We will continue to add additional MongoDB 5.0 capabilities based on what customers ask us to build. For the latest list of supported APIs, see Supported MongoDB APIs, operations, and data types in Amazon DocumentDB.

What's new in Amazon DocumentDB 5.0.1

Amazon DocumentDB 5.0.1 is the first minor version release for the Amazon DocumentDB 5.0 major version. Minor versions include improvements such as new features, security fixes, and bug fixes. You can upgrade to 5.0.1 from 5.0.0 using a minor version upgrade. For more information, see Amazon DocumentDB minor version upgrade.

- Enhanced aggregation capabilities with new operators (`$rand`, `$pow`, `$dateToParts`, `$dateFromParts`).

- Active connections metric to monitor instances. For more information, see Instance quotas.
- Granular command-level performance metrics in CloudWatch for CRUD operations (`find`, `insert`, `findAndModify`, `update`, `delete`, `aggregate`, `count`, `distinct`, `getMore`, `abortTransaction`, `commitTransaction`), exposing `AvgDuration`, `P100Duration`, and `MaxConcurrent` dimensions for improved operational monitoring.
- Amazon DocumentDB 5.0.1 only supports TLS 1.2 and higher for database connections.
- Added `gcRuntimeStats`, `documentFragmentStats`, and breakup of collection level metrics into `storageSegmentBase` and `storageSegmentExtended` in `collstats` to surface more information into the overall health of the garbage collection process.

Get started with Amazon DocumentDB 5.0

To get started with Amazon DocumentDB 5.0, see the Get started with Amazon DocumentDB. You can create a new Amazon DocumentDB 5.0 cluster using the AWS Management Console or the AWS SDK, AWS CLI, or CloudFormation. When connecting to Amazon DocumentDB, it is required that you use a MongoDB driver or utility that is compatible with MongoDB 5.0 or higher.

> **Note**
>
> When using the AWS SDK, AWS CLI, or CloudFormation, the engine version will default to the newest 8.0 minor version available in the AWS Region. To create a 5.0 cluster, specify `engineVersion` explicitly. For more information, see Amazon DocumentDB cluster settings.
>
> For a given Amazon DocumentDB cluster, you can determine the cluster version using the AWS CLI to call `describe-db-clusters` or use the Amazon DocumentDB management console to view the engine version number for a particular cluster.

Amazon DocumentDB 5.0 and 8.0 supports the Amazon EC2 `r8g` Graviton4 processor and Graviton2 processors such as `r6g` and `t4.medium` instance types for your clusters and is available in all supported AWS Regions (see Instances). For more information on pricing, see Amazon DocumentDB (with MongoDB compatibility) Pricing.

Upgrade or migrate to Amazon DocumentDB 5.0

You can migrate from MongoDB 3.6 or MongoDB 4.0 to Amazon DocumentDB 5.0 using the AWS DMS or utilities like mongodump, mongorestore, mongoimport, and mongoexport. For instructions

on migrating, see Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service.

Functional differences

Functional differences between Amazon DocumentDB 4.0 and 5.0

With the release of Amazon DocumentDB 5.0, there are functional differences between Amazon DocumentDB 4.0 and Amazon DocumentDB 5.0:

- The backup built-in role now supports `serverStatus`. Action - Developers and applications with backup role can collect statistics about the state of the Amazon DocumentDB cluster.
- The `SecondaryDelaySecs` field replaces `slaveDelay` in `replSetGetConfig` output.
- The **hello** command replaces `isMaster` - **hello** returns a document that describes the role of a Amazon DocumentDB cluster.
- Amazon DocumentDB 5.0 now supports index scans with the `$elemMatch` operator in the first nesting level. Index scans are supported when the query only filter has one level of the `$elemMatch` filter but are not supported if a nested `$elemMatch` query is included.

  For example, in Amazon DocumentDB 5.0, if you include the `$elemMatch` operator in the nested level, it will not return a value as it does in Amazon DocumentDB 4.0:

```
db.foo.insert(
[
    {a: {b: 5}},
    {a: {b: [5]}},
    {a: {b: [3, 7]}},
    {a: [{b: 5}]},
    {a: [{b: 3}, {b: 7}]},
    {a: [{b: [5]}]},
    {a: [{b: [3, 7]}]},
    {a: [[{b: 5}]]},
    {a: [[{b: 3}, {b: 7}]]},
    {a: [[{b: [5]}]]},
    {a: [[{b: [3, 7]}]]}
]);

// DocumentDB 5.0
> db.foo.find({a: {$elemMatch: {b: {$elemMatch: {$lt: 6, $gt: 4}}}}}, {_id: 0})
{ "a" : [ { "b" : [ 5 ] } ] }
```

```
// DocumentDB 4.0
> db.foo.find({a: {$elemMatch: {b: {$elemMatch: {$lt: 6, $gt: 4}}}}}, {_id: 0})
{ "a" : [ { "b" : [ 5 ] } ] }
{ "a" : [ [ { "b" : [ 5 ] } ] ] }
```

- The "$" projection in Amazon DocumentDB 4.0 returns all documents with all fields. With Amazon DocumentDB 5.0, the **find** command with a "$" projection returns documents that match the query parameter containing only the field that matched the "$" projection.
- In Amazon DocumentDB 5.0, the **find** commands with `$regex` and `$options` query parameters return an error: "Cannot set options in both `$regex` and `$options`".
- With Amazon DocumentDB 5.0, `$indexOfCP` now returns "-1" when:
  - the substring is not found in the string expression, or
  - start is a number greater than end, or
  - start is a number greater than the byte length of the string.
- In Amazon DocumentDB 4.0, `$indexOfCP` returns "0" when the start position is a number greater than end or the byte length of the string.
- With Amazon DocumentDB 5.0, projection operations in `_id fields`, for example `{"_id.nestedField" : 1}`, return documents that only include the projected field. Whereas in Amazon DocumentDB 4.0, nested field projection commands do not filter out any document.

MongoDB 4.0 compatibility

Topics

- Amazon DocumentDB 4.0 features
- Get started with Amazon DocumentDB 4.0
- Upgrade or migrate to Amazon DocumentDB 4.0
- Functional differences

Amazon DocumentDB 4.0 features

Amazon DocumentDB 4.0 introduced many new features and capabilities that included ACID transactions and improvements to change streams. The summary below shows some of the major features that were introduced in Amazon DocumentDB 4.0. To see a full list of the capabilities, see the Release notes.

- **ACID Transactions**: Amazon DocumentDB now supports the ability to perform transactions across multiple documents, statements, collections, and databases. Transactions simplify application development by enabling you to perform atomic, consistent, isolated, and durable (ACID) operations across one or more documents within an Amazon DocumentDB cluster. For more information, see Transactions in Amazon DocumentDB.
- **Change streams**: You now have the ability to open a change stream at the cluster level (`client.watch()` or `mongo.watch()`) and the database (`db.watch()`), you can specify a `startAtOperationTime` to open a change stream cursor, and lastly you can now extend your change stream retention period to 7 days (previously 24 hours). For more information, see Using change streams with Amazon DocumentDB.
- **AWS Database Migration Service** (AWS DMS): You can now use AWS DMS to migrate your MongoDB 4.0 workloads to Amazon DocumentDB. AWS DMS now supports a MongoDB 4.0 source, Amazon DocumentDB 4.0 target, and an Amazon DocumentDB 3.6 source for performing upgrades between Amazon DocumentDB 3.6 and 4.0. For more information, see AWS DMS Documentation.
- **Performance and indexing**: You can now utilize an index with `$lookup`, find queries with a projection that contain one field or one field and the `_id` field can be served direct from the index and without needing to read from the collection (covered query), the ability to `hint()` with `findAndModify`, performance optimizations for `$addToSet`, and improvements to reduce overall index sizes. For more information, see Release notes.
- **Operators**: Amazon DocumentDB 4.0 now supports a number of new aggregation operators: `$ifNull`, `$replaceRoot`, `$setIsSubset`, `$setIntersection`, `$setUnion`, `$setEquals`. You can see all the MongoDB APIs, Operations, and Data Types that we support at Supported MongoDB APIs, operations, and data types in Amazon DocumentDB.
- **Role based access control** (RBAC): With both `ListCollection` and `ListDatabase` commands you can now optionally use the `authorizedCollections` and `authorizedDatabases` parameters to allow users to list the collections and databases that they have permission to access without requiring the `listCollections` and `listDatabase` roles, respectively. You also have the ability to kill your own cursors without requiring the `KillCursor` role.

Amazon DocumentDB does not support every MongoDB 4.0 feature. When we built Amazon DocumentDB 4.0, we worked backwards from the feature and capabilities that our customers asked us to build the most. We will continue to add additional MongoDB 4.0 capabilities based on what customers ask us to build. For example, Amazon DocumentDB 4.0 does not currently support the type conversion operators or the string operators that were introduced in MongoDB 4.0. For the

latest list of supported APIs, see Supported MongoDB APIs, operations, and data types in Amazon DocumentDB.

Get started with Amazon DocumentDB 4.0

To get started with Amazon DocumentDB 4.0, see the Get started with Amazon DocumentDB. You can create a new Amazon DocumentDB 4.0 cluster using the AWS Management Console or the AWS SDK, AWS CLI, or CloudFormation. When connecting to Amazon DocumentDB, it is required that you use a MongoDB driver or utility that is compatible with MongoDB 4.0 or higher.

> **Note**
>
> When using the AWS SDK, AWS CLI, or CloudFormation, the engine version will default to the newest 8.0 minor version available in the AWS Region. To create a 4.0 cluster, specify `engineVersion` explicitly. For more information, see Amazon DocumentDB cluster settings.
>
> For a given Amazon DocumentDB cluster, you can determine the cluster version using the AWS CLI to call `describe-db-clusters` or use the Amazon DocumentDB management console to view the engine version number for a particular cluster.

Amazon DocumentDB 4.0 supports `r5`, `r6g`, `t3.medium`, and `t4g.medium` instance types for your clusters and is available in all supported AWS Regions. There is no additional cost for using Amazon DocumentDB 4.0. For more information on pricing, see Amazon DocumentDB (with MongoDB compatibility) Pricing.

Upgrade or migrate to Amazon DocumentDB 4.0

You can migrate from MongoDB 3.6 or MongoDB 4.0 to Amazon DocumentDB 4.0 using the AWS DMS or utilities like mongodump, mongorestore, mongoimport, and mongoexport. Similarly, you can use the same tools to upgrade from Amazon DocumentDB 3.6 to Amazon DocumentDB 4.0. For instructions on migrating, see Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service.

Functional differences

Functional differences between Amazon DocumentDB 3.6 and 4.0

With the release of Amazon DocumentDB 4.0, there are functional differences between Amazon DocumentDB 3.6 and Amazon DocumentDB 4.0:

- **Projection for nested documents**: Amazon DocumentDB 3.6 considers the first field in a nested document when applying a projection. However, Amazon DocumentDB 4.0 will parse subdocuments and apply the projection to each sub document as well. For example: if the projection is `"a.b.c": 1`, then the behavior in both versions is identical. However, if the projection is `{a:{b:{c:1}}}` then Amazon DocumentDB 3.6 will only apply the projection to 'a' and not 'b' or 'c'.
- **Behavior for `minKey`, `maxKey`**: In Amazon DocumentDB 4.0, the behavior for `{x: {$gt:MaxKey}}` returns nothing, and for `{x:{$lt:MaxKey}}` returns everything.
- **Document comparison differences**: Comparing numerical values of different types (double, int, long) in subdocuments (e.g., b in `{"_id" :1, "a" :{"b":1}}`) now provides a consistent output across numerical data types and for each level of a document.

Functional differences between Amazon DocumentDB 4.0 and MongoDB 4.0

Below are functional differences between Amazon DocumentDB 4.0 and MongoDB 4.0.

- **Lookup with empty key in path**: When a collection contains a document with empty key inside the array (e.g. `{"x" : [ { "" : 10 }, { "b" : 20 } ]}`), and when the key used in the query ends in an empty string (e.g. `x.`), then Amazon DocumentDB will return that document since it traverses all the documents in the array whereas MongoDB will not return that document.
- **`$setOnInsert` along with `$` in the path**: The field operator `$setOnInsert` will not work in combination with $ in the path in Amazon DocumentDB, which is also consistent with MongoDB 4.0.

Functional differences: Amazon DocumentDB and MongoDB

The following are the functional differences between Amazon DocumentDB (with MongoDB compatibility) and MongoDB.

Topics

- Functional benefits of Amazon DocumentDB
- Updated functional differences
- Functional differences with MongoDB

Functional benefits of Amazon DocumentDB

Implicit transactions

In Amazon DocumentDB, all CRUD statements (`findAndModify`, `update`, `insert`, `delete`) guarantee atomicity and consistency, even for operations that modify multiple documents. With the launch of Amazon DocumentDB 4.0, explicit transactions that provide ACID properties for multi-statement and multi-collection operations are now supported. For more on using transactions in Amazon DocumentDB, see Transactions in Amazon DocumentDB.

The following are examples of operations in Amazon DocumentDB that modify multiple documents that satisfy both atomic and consistent behaviors.

```
db.miles.update(
    { "credit_card": { $eq: true } },
    { $mul: { "flight_miles.$[]": NumberInt(2) } },
    { multi: true }
)
```

```
db.miles.updateMany(
    { "credit_card": { $eq: true } },
    { $mul: { "flight_miles.$[]": NumberInt(2) } }
)
```

```
db.runCommand({
  update: "miles",
  updates: [
    {
      q: { "credit_card": { $eq: true } },
      u: { $mul: { "flight_miles.$[]": NumberInt(2) } },
      multi: true
```

```
    }
  ]
})
```

```
db.products.deleteMany({
  "cost": { $gt: 30.00 }
})
```

```
db.runCommand({
  delete: "products",
  deletes: [{ q: { "cost": { $gt: 30.00 } }, limit: 0 }]
})
```

The individual operations that compose bulk operations such as `updateMany` and `deleteMany` are atomic but the entirety of the bulk operation is not atomic. For example, the entirety of the `insertMany` operation is atomic if the individual insert operations execute successfully without error. If an error is encountered with an `insertMany` operation, each individual insert statement within the `insertMany` operation will execute as an atomic operation. If you require ACID properties for `insertMany`, `updateMany`, and `deleteMany` operations, it is recommended to use a transaction.

Updated functional differences

Amazon DocumentDB continues to improve compatibility with MongoDB by working backwards from the capabilities our customers ask us to build. This section contains the functional differences that we have removed in Amazon DocumentDB to make migrations and building applications easier for our customers.

Topics

- Array indexing
- Multi-key indexes
- Null characters in strings
- Role-based access control
- $regex indexing

- Projection for nested documents

Array indexing

As of April 23, 2020, Amazon DocumentDB now supports the ability to index arrays that are greater than 2,048 bytes. The limit for an individual item in an array still remains as 2,048 bytes, which is consistent with MongoDB.

If you are creating a new index, no action is needed to take advantage of the improved functionality. If you have an existing index, you can take advantage of the improved functionality by dropping the index and then recreating it. The current index version with the improved capabilities is `"v" : 3`.

> **Note**
>
> For production clusters, dropping the index might have an impact on your application performance. Test first and proceed with caution when making changes to a production system. In addition, the time it takes to recreate the index depends on the overall data size of the collection.

You can query for the version of your indexes using the following command.

```
db.collection.getIndexes()
```

Output from this operation looks something like the following. In this output, the version of the index is `"v" : 3`, which is the most current index version.

```
[
    {
        "v" : 3,
        "key" : {
        "_id" : 1
        },
        "name" : "_id_",
        "ns" : "test.test"
    }
]
```

Multi-key indexes

As of April 23, 2020, Amazon DocumentDB now supports the ability to create a compound index with multiple keys in the same array.

If you are creating a new index, no action is needed to take advantage of the improved functionality. If you have an existing index, you can take advantage of the improved functionality by dropping the index and then recreating it. The current index version with the improved capabilities is `"v" : 3`.

> **Note**
>
> For production clusters, dropping the index might have an impact on your application performance. Test first and proceed with caution when making changes to a production system. In addition, the time it takes to recreate the index depends on the overall data size of the collection.

You can query for the version of your indexes using the following command.

```
db.collection.getIndexes()
```

Output from this operation looks something like the following. In this output, the version of the index is `"v" : 3`, which is the most current index version.

```
[
    {
        "v" : 3,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "test.test"
    }
]
```

Null characters in strings

As of June 22, 2020, Amazon DocumentDB now supports null characters (`'\0'`) in strings.

Role-based access control

As of March 26, 2020, Amazon DocumentDB supports role-based access control (RBAC) for built-in roles. To learn more, see Role-Based Access Control.

$regex indexing

As of June 22, 2020, Amazon DocumentDB now supports the ability for `$regex` operators to utilize an index.

To utilize an index with the `$regex` operator, you must use the `hint()` command. When using `hint()`, you must specify the name of the field you are applying the `$regex` on. For example, if you have an index on field `product` with the index name as `p_1`, `db.foo.find({product: /^x.*/}).hint({product:1})` will utilize the `p_1` index, but `db.foo.find({product: /^x.*/}).hint("p_1")` will not utilize the index. You can verify if an index is chosen by utilizing the `explain()` command or using the profiler for logging slow queries. For example, `db.foo.find({product: /^x.*/}).hint("p_1").explain()`.

> **Note**
>
> The `hint()` method can only be used with one index at a time.

The use of an index for a `$regex` query is optimized for regex queries that utilize a prefix and do not specify the `i`, `m`, or o regex options.

When using an index with `$regex`, it is recommended that you create an index on highly selective fields where the number of duplicate values is less than 1% of the total number of documents in the collection. As an example, if your collection contains 100,000 documents, only create indexes on fields where the same value occurs 1000 times or fewer.

Projection for nested documents

There is a functional difference with `$project` operator between Amazon DocumentDB and MongoDB in version 3.6 that has been resolved in Amazon DocumentDB 4.0 but will remain unsupported in Amazon DocumentDB 3.6.

Amazon DocumentDB 3.6 only considers the first field in a nested document when applying a projection whereas MongoDB 3.6 will parse subdocuments and apply the projection to each sub document as well.

For example: if the projection is `"a.b.c": 1`, then the behavior works as expect in both Amazon DocumentDB and MongoDB. However, if the projection is `{a:{b:{c:1}}}` then Amazon DocumentDB 3.6 will only apply the projection to `a` and not `b` or `c`. In Amazon DocumentDB 4.0, the projection `{a:{b:{c:1}}}` will be applied to `a`, `b`, and `c`.

Functional differences with MongoDB

Topics

- $vectorSearch operator
- OpCountersCommand
- Admin databases and collections
- cursormaxTimeMS
- explain()
- Index builds
- Lookup with empty key in path
- MongoDB APIs, operations, and data types
- mongodump and mongorestore utilities
- Result ordering
- Retryable writes
- Sparse index
- Using $elemMatch within an $all expression
- Dollar($) and dot(.) in field names
- $lookup
- $natural and reverse sorting

`$vectorSearch` operator

Amazon DocumentDB does not support `$vectorSearch` as an independent operator. Instead we support, `vectorSearch` inside the `$search` operator. For more information, see Vector search for Amazon DocumentDB.

`OpCountersCommand`

Amazon DocumentDB's `OpCountersCommand` behavior deviates from MongoDB's `opcounters.command` as follows:

- MongoDB's `opcounters.command` counts all commands except insert, update, and delete while Amazon DocumentDB's `OpCountersCommand` also excludes the `find` command.
- Amazon DocumentDB counts some internal commands toward the `OpCountersCommand`.

Admin databases and collections

Amazon DocumentDB does not support the admin or local database nor MongoDB `system.*` or `startup_log` collections respectively.

`cursormaxTimeMS`

In Amazon DocumentDB, `cursor.maxTimeMS` resets the counter for each `getMore` request. Thus, if a 3000MS `maxTimeMS` is specified, the query takes 2800MS, and each subsequent `getMore` request takes 300MS, then the cursor will not timeout. The cursor will only timeout when a single operations, either the query or an individual `getMore` request, takes more than the specified `maxTimeMS`. Further, the sweeper that checks cursor execution time runs at a five (5) minute granularity.

explain()

Amazon DocumentDB emulates the MongoDB 3.6, 4.0, 5.0, and 8.0 APIs on a purpose-built database engine that utilizes a distributed, fault-tolerant, self-healing storage system. As a result, query plans and the output of `explain()` may differ between Amazon DocumentDB and MongoDB. Customers who want control over their query plan can use the `$hint` operator to enforce selection of a preferred index.

Index builds

Amazon DocumentDB allows only one index build to occur on a collection at any given time. Either in the foreground or the background. If operations such as `createIndex()` or `dropIndex()` occur on the same collection when an index build is currently in progress, the newly attempted operation will fail.

By default, index builds in Amazon DocumentDB and MongoDB version 4.0 occur in the foreground. MongoDB version 4.2, and later ignores the background index build option if specified to createIndexes or its shell helpers `createIndex()` and `createIndexes()`.

A Time to Live (TTL) index starts expiring documents after the index build is completed.

Lookup with empty key in path

When you look up with a key that includes empty string as part of the path (e.g. `x.`, `x..b`), and the object has an empty string key path (e.g. `{"x" : [ { "" : 10 }, { "b" : 20 } ]}`) inside an array, Amazon DocumentDB will return different results than if you were to run the same look up in MongoDB.

In MongoDB, the empty key path look up within array works as expected when the empty string key is not at the end of path look up. However, when the empty string key is at the end of path look up, it does not look into the array.

However in Amazon DocumentDB, only the first element within the array is read, because `getArrayIndexFromKeyString` converts empty string to `0`, so string key look up is treated as array index look up.

MongoDB APIs, operations, and data types

Amazon DocumentDB is compatible with the MongoDB 3.6, 4.0, 5.0, and 8.0 APIs. For an up-to-date list of supported functionality, see Supported MongoDB APIs, operations, and data types in Amazon DocumentDB.

`mongodump` and `mongorestore` utilities

Amazon DocumentDB does not support an admin database and thus does not dump or restore the admin database when using the `mongodump` or `mongorestore` utilities. When you create a new database in Amazon DocumentDB using `mongorestore`, you need to re-create the user roles in addition to the restore operation.

> **MongoDB Database Tools version requirement**
>
> Use MongoDB Database Tools up to and including version 100.11.0 for Amazon DocumentDB. To download, see MongoDB Database Tools releases on the MongoDB website.

Result ordering

Amazon DocumentDB does not guarantee implicit result sort ordering of result sets. To ensure the ordering of a result set, explicitly specify a sort order using `sort()`.

The following example sorts the items in the inventory collection in descending order based on the stock field.

```
db.inventory.find().sort({ stock: -1 })
```

When using the `$sort` aggregation stage, the sort order is not preserved unless the `$sort` stage is the last stage in the aggregation pipeline. When using the `$sort` aggregation stage in combination with the `$group` aggregation stage, the `$sort` aggregation stage is only applied to the `$first` and `$last` accumulators. In Amazon DocumentDB 4.0, support was added for `$push` to respect sort order from the previous `$sort` stage.

Retryable writes

Most current MongoDB-compatible drivers enable retryable writes by default. Amazon DocumentDB supports retryable writes starting with engine version 8.0.2. On earlier engine versions, retryable writes are not supported and you must disable them to avoid errors.

On engine versions earlier than 8.0.2, enabling retryable writes results in an error message similar to the following.

```
{"ok":0,"errmsg":"Retryable writes are not supported","code":301,"name":"MongoError"}
```

Retryable writes can be disabled via the connection string (for example, `MongoClient("mongodb://my.mongodb.cluster/db?retryWrites=false")`) or the MongoClient constructor's keyword argument (for example, `MongoClient("mongodb://my.mongodb.cluster/db", retryWrites=False)`).

For more information about retryable writes on engine version 8.0.2 and later, see Retryable writes in Amazon DocumentDB.

Sparse index

To use a sparse index that you have created in a query, you must use the `$exists` clause on the fields that cover the index. If you omit `$exists`, Amazon DocumentDB will not use the sparse index.

The following is an example.

```
db.inventory.count({ "stock": { $exists: true }})
```

For sparse, multi-key indexes, Amazon DocumentDB does not support a unique key constraint if the look up of a document results in a set of values and only a subset of the indexed fields is missing. For example, `createIndex({"a.b" : 1 }, { unique : true, sparse :true })` is not supported, given the input of `"a" : [ { "b" : 2 }, { "c" : 1 } ]`, as `"a.c"` is stored in the index.

Using `$elemMatch` within an `$all` expression

Amazon DocumentDB does not currently support the use of the `$elemMatch` operator within an `$all` expression. As a workaround, you can use the `$and` operator with `$elemMatch` as follows.

Original operation:

```
db.col.find({
  qty: {
    $all: [
      { "$elemMatch": { part: "xyz", qty: { $lt: 11 } } },
      { "$elemMatch": { num: 40, size: "XL" } }
    ]
  }
})
```

Updated operation:

```
db.col.find({
  $and: [
    { qty: { "$elemMatch": { part: "xyz", qty: { $lt: 11 } } } },
    { qty: { "$elemMatch": { qty: 40, size: "XL" } } }
  ]
```

```
})
```

Dollar($) and dot(.) in field names

Amazon DocumentDB does not support querying Dollar($) prefixed fields in $in, $nin and $all in nested objects. For example, the following query is not valid in Amazon DocumentDB:

```
coll.find({"field": {"$all": [{ "$a": 1 }]}})
```

$lookup

Amazon DocumentDB supports the ability to do equality matches (for example, left outer join) and also supports uncorrelated subqueries. Starting with version 8.0.2, Amazon DocumentDB also supports correlated subqueries, including the concise correlated subquery syntax. In a correlated subquery, only a subset of aggregation stages that evaluate expressions can reference the `let` variables. For the list of supported stages and related restrictions, see Supported stages in correlated subqueries. In versions earlier than 8.0.2, correlated subqueries are not supported.

Utilizing an index with $lookup

You can now utilize an index with the `$lookup` stage operator. Based on your use case, there are multiple indexing algorithms that you can use to optimize for performance. This section will explain the different indexing algorithms for `$lookup` and help you choose the best one for your workload.

By default, Amazon DocumentDB will utilize the hash algorithm when `allowDiskUse:false` is used and sort merge when `allowDiskUse:true` is used.

> **Note**
>
> The `allowDiskUse` option is currently not supported for the `find` command. The option is only supported as part of aggregation. Use the aggregation framework with `allowDiskUse:true` to handle large queries that might exceed memory limits.

For some use cases, it may be desirable to force the query optimizer to use a different algorithm. Below are the different indexing algorithms that the `$lookup` aggregation operator can use:

- **Nested loop**: A nested loop plan is typically beneficial for a workload if the foreign collection is <1 GB and the field in the foreign collection has an index. If the nested loop algorithm is being used, the explain plan will show the stage as `NESTED_LOOP_LOOKUP`.
- **Sort merge**: A sort merge plan is typically beneficial for a workload if the foreign collection does not have an index on the field used in lookup and the working dataset doesn't fit in memory. If the sort merge algorithm is being used, the explain plan will show the stage as `SORT_LOOKUP`.
- **Hash**: A hash plan is typically beneficial for a workload if the foreign collection is < 1 GB and the working dataset fits in memory. If the hash algorithm is being used, the explain plan will show the stage as `HASH_LOOKUP`.

You can identify the indexing algorithm that is being used for the `$lookup` operator by using `explain` on the query. Below is an example:

```
db.localCollection.explain().aggregate(
   [
      {
         $lookup:
            {
               from: "foreignCollection",
               localField: "a",
               foreignField: "b",
               as: "joined"
            }
      }
   ]
)

output
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "test.localCollection",
        "winningPlan" : {
            "stage" : "SUBSCAN",
            "inputStage" : {
                "stage" : "SORT_AGGREGATE",
                "inputStage" : {
                    "stage" : "SORT",
                    "inputStage" : {
```

```
                        "stage" : "NESTED_LOOP_LOOKUP",
                        "inputStages" : [
                            {
                                "stage" : "COLLSCAN"
                            },
                            {
                                "stage" : "FETCH",
                                "inputStage" : {
                                    "stage" : "COLLSCAN"
                                }
                            }
                        ]
                    }
                }
            }
        }
    },
    "serverInfo" : {
        "host" : "devbox-test",
        "port" : 27317,
        "version" : "3.6.0"
    },
    "ok" : 1
}
```

As an alternative to using the `explain()` method, you can use the profiler to review the algorithm that is being utilized with your use of the `$lookup` operator. For more information on the profiler, see Profiling Amazon DocumentDB operations.

Using a `planHint`

If you wish to force the query optimizer to use a different indexing algorithm with `$lookup`, you can use a `planHint`. To do that, use the comment in the aggregation stage options to force a different plan. Below is an example of the syntax for the comment:

```
comment : {
    comment :  "<string>",
    lookupStage : { planHint : "SORT" | "HASH" | "NESTED_LOOP" }
}
```

Below is an example of using the `planHint` to force the query optimizer to use the HASH indexing algorithm:

```
db.foo.aggregate(
   [
      {
         $lookup:
            {
               from: "foo",
               localField: "_id",
               foreignField: "_id",
               as: "joined"
            },
      }
   ]
),
{
   comment : "{ \"lookupStage\" : { \"planHint\": \"HASH\" }}"
```

To test which algorithm is best for your workload, you can use the `executionStats` parameter of the `explain` method to measure the execution time of the `$lookup` stage while modifying the indexing algorithm (i.e., HASH/SORT/NESTED_LOOP).

The following example shows how to use `executionStats` to measure the execution time of the `$lookup` stage using the SORT algorithm.

```
db.foo.explain("executionStats").aggregate(
   [
      {
         $lookup:
            {
               from: "foo",
               localField: "_id",
               foreignField: "_id",
               as: "joined"
            },
      }
   ]
),
{
```

```
    comment : "{ \"lookupStage\" : { \"planHint\": \"SORT\" }}"
```

`$natural` and reverse sorting

Amazon DocumentDB supports `$natural` for forward collection scans only. Reverse collection scans (`{$natural: -1}`) will lead to a `MongoServerError`.

Supported MongoDB APIs, operations, and data types in Amazon DocumentDB

Amazon DocumentDB (with MongoDB compatibility) is a fast, scalable, highly-available, and fully managed document database service that supports MongoDB workloads. Amazon DocumentDB is compatible with the MongoDB 3.6, 4.0, 5.0, and 8.0 APIs. This section lists the supported functionality. For support using MongoDB APIs and drivers, consult the MongoDB Community Forums. For support using the Amazon DocumentDB service, contact the appropriate AWS support team. For functional differences between Amazon DocumentDB and MongoDB, see Functional differences: Amazon DocumentDB and MongoDB.

MongoDB commands and operators that are internal-only or not applicable to a fully-managed service are not supported and are not included in the list of supported functionality.

We have added over 50+ additional capabilities since launch, and will continue to work backwards from our customers to deliver the capabilities that they need. For information on the most recent launches, see Amazon DocumentDB Announcements.

If there is a feature that isn't supported that you'd like us to build, let us know by sending an email with your accountID, the requested features, and use case to the Amazon DocumentDB service team.

Topics

- Database commands
- Query and projection operators
- Update operators
- Geospatial
- Cursor methods

- Aggregation pipeline operators
- Data types
- Indexes

Database commands

Topics

- Administrative commands
- Aggregation
- Authentication
- Diagnostic commands
- Query and write operations
- Role management commands
- Sessions commands
- User management
- Sharding commands

Administrative commands

Command	3.6	4.0	5.0	8.0	Elastic cluster
Capped Collections	No	No	No	No	No
cloneCollectionAsCapped	No	No	No	No	No
collMod	Partial	Partial	Partial	Partial	Partial
collMod: expireAfterSeconds	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
collMod: hidden	No	No	Yes (5.0.2+)	Yes (8.0.1+)	No
convertToCapped	No	No	No	No	No
copydb	No	No	No	No	No
create	Yes	Yes	Yes	Yes	Yes
createView	No	No	No	Yes	No
createIndexes	Yes	Yes	Yes	Yes	Yes
currentOp	Yes	Yes	Yes	Yes	Yes
drop	Yes	Yes	Yes	Yes	Yes
dropDatabase	Yes	Yes	Yes	Yes	Yes
dropIndexes	Yes	Yes	Yes	Yes	Yes
filemd5	No	No	No	No	No
getAuditConfig	No	Yes	Yes	Yes	No
killCursors	Yes	Yes	Yes	Yes	Yes
killOp	Yes	Yes	Yes	Yes	Yes
listCollections*	Yes	Yes	Yes	Yes	Yes
listDatabases	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
listIndexes	Yes	Yes	Yes	Yes	Yes
reIndex	No	No	Yes	Yes	No
renameCollection	Yes	Yes	Yes	Yes	No
setAuditConfig	No	Yes	Yes	Yes	No

* The `type` key in the filter option is not supported.

Aggregation

Command	3.6	4.0	5.0	8.0	Elastic cluster
aggregate	Yes	Yes	Yes	Yes	Yes
count	Yes	Yes	Yes	Yes	Yes
distinct	Yes	Yes	Yes	Yes	Yes
mapReduce	No	No	No	Yes	No

Authentication

Command	3.6	4.0	5.0	8.0	Elastic cluster
authenticate	Yes	Yes	Yes	Yes	Yes
logout	Yes	Yes	Yes	Yes	Yes

Diagnostic commands

Command	3.6	4.0	5.0	8.0	Elastic cluster
buildInfo	Yes	Yes	Yes	Yes	Yes
collStats	Yes	Yes	Yes	Yes	Yes
connPoolStats	No	No	No	No	No
connectionStatus	Yes	Yes	Yes	Yes	Yes
dataSize	Yes	Yes	Yes	Yes	Yes
dbHash	No	No	No	No	No
dbStats	Yes	Yes	Yes	Yes	Yes
explain	Yes	Yes	Yes	Yes	Yes
explain: executionStats	Yes	Yes	Yes	Yes	Yes
features	No	No	No	No	No
hostInfo	Yes	Yes	Yes	Yes	Yes
listCommands	Yes	Yes	Yes	Yes	Yes
planCacheClearFilters	No	No	Yes	Yes	No
planCacheListFilters	No	No	Yes	Yes	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
planCache SetFilter	No	No	Yes	Yes	No
profiler	Yes	Yes	Yes	Yes	No
serverStatus	Yes	Yes	Yes	Yes	Yes
top	Yes	Yes	Yes	Yes	Yes

Query and write operations

Command	3.6	4.0	5.0	8.0	Elastic cluster
Change streams	Yes	Yes	Yes	Yes	No
delete	Yes	Yes	Yes	Yes	Yes
find	Yes	Yes	Yes	Yes	Yes
findAndModify	Yes	Yes	Yes	Yes	Yes
getLastError	No	No	No	No	No
getMore	Yes	Yes	Yes	Yes	Yes
getPrevError	No	No	No	No	No
GridFS	Yes	Yes	Yes	Yes	No
insert	Yes	Yes	Yes	Yes	Yes
parallelCollectionScan	No	No	No	No	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
resetError	No	No	No	No	No
update	Yes	Yes	Yes	Yes	Yes
ReplaceOne	Yes	Yes	Yes	Yes	Yes

Role management commands

Command	3.6	4.0	5.0	8.0	Elastic cluster
c	Ye	Ye	Ye	Ye	No
e					
c	Ye	Ye	Ye	Ye	No
l					
t					
c	Ye	Ye	Ye	Ye	No
g	Ye	Ye	Ye	Ye	No
s					
r	Ye	Ye	Ye	Ye	No
e					
e					
r	Ye	Ye	Ye	Ye	No
v					
c					
r	Ye	Ye	Ye	Ye	No
u	Ye	Ye	Ye	Ye	No
e					

Sessions commands

Command	3.6	4.0	5.0	8.0	Elastic cluster
abortTransaction	No	Yes	Yes	Yes	No
commitTransaction	No	Yes	Yes	Yes	No
endSessions	No	No	No	No	No
killAllSessions	No	Yes	Yes	Yes	No
killAllSessionsByPattern	No	No	No	No	No
killSessions	No	Yes	Yes	Yes	No
refreshSessions	No	No	No	No	No
startSession	No	Yes	Yes	Yes	No

User management

Command	3.6	4.0	5.0	8.0	Elastic cluster
createUser	Yes	Yes	Yes	Yes	Yes
dropAllUsersFromDatabase	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
dropUser	Yes	Yes	Yes	Yes	Yes
grantRolesToUser	Yes	Yes	Yes	Yes	Yes
revokeRolesFromUser	Yes	Yes	Yes	Yes	Yes
updateUser	Yes	Yes	Yes	Yes	Yes
usersInfo	Yes	Yes	Yes	Yes	Yes

Sharding commands

Command	Elastic cluster
abortReshardCollection	No
addShard	No
addShardToZone	No
balancerCollectionStatus	No
balancerStart	No
balancerStatus	No
balancerStop	No
checkShardingIndex	No
clearJumboFlag	No
cleanupOrphaned	No
cleanupReshardCollection	No

Command	Elastic cluster
commitReshardCollection	No
enableSharding	Yes
flushRouterConfig	No
getShardMap	No
getShardVersion	No
isdbgrid	No
listShards	No
medianKey	No
moveChunk	No
movePrimary	No
mergeChunks	No
refineCollectionShardKey	No
removeShard	No
removeShardFromZone	No
reshardCollection	No
setAllowMigrations	No
setShardVersion	No
shardCollection	Yes
shardingState	No
split	No

Command	Elastic cluster
splitVector	No
unsetSharding	No
updateZoneKeyRange	No

Query and projection operators

Topics

- Array operators
- Bitwise operators
- Comment operator
- Comparison operators
- Element operators
- Evaluation query operators
- Logical operators
- Projection operators

Array operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$all	Yes	Yes	Yes	Yes	Yes
$elemMatch	Yes	Yes	Yes	Yes	Yes
$size	Yes	Yes	Yes	Yes	Yes

Bitwise operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$bitsAllSet	Yes	Yes	Yes	Yes	Yes
$bitsAnySet	Yes	Yes	Yes	Yes	Yes
$bitsAllClear	Yes	Yes	Yes	Yes	Yes
$bitsAnyClear	Yes	Yes	Yes	Yes	Yes

Comment operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$comment	Yes	Yes	Yes	Yes	Yes

Comparison operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$eq	Yes	Yes	Yes	Yes	Yes
$gt	Yes	Yes	Yes	Yes	Yes
$gte	Yes	Yes	Yes	Yes	Yes
$in	Yes	Yes	Yes	Yes	Yes
$lt	Yes	Yes	Yes	Yes	Yes
$lte	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$ne	Yes	Yes	Yes	Yes	Yes
$nin	Yes	Yes	Yes	Yes	Yes

Element operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$exists	Yes	Yes	Yes	Yes	Yes
$type	Yes	Yes	Yes	Yes	Yes

Evaluation query operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$expr	No	Yes	Yes	Yes	No
$jsonSchema	No	Yes	Yes	Yes	No
$mod	Yes	Yes	Yes	Yes	Yes
$regex	Yes	Yes	Yes	Yes	Yes
$text	No	No	Yes	Yes	No
$where	No	No	No	No	No

Logical operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$and	Yes	Yes	Yes	Yes	Yes
$nor	Yes	Yes	Yes	Yes	Yes
$not	Yes	Yes	Yes	Yes	Yes
$or	Yes	Yes	Yes	Yes	Yes

Projection operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$	Yes	Yes	Yes	Yes	Yes
$elemMatch	Yes	Yes	Yes	Yes	Yes
$meta	No	No	Yes	Yes	No
$slice	Yes	Yes	Yes	Yes	Yes

Update operators

Topics

- Array operators
- Bitwise operators
- Field operators
- Update modifiers

Array operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$	Yes	Yes	Yes	Yes	Yes
$[]	Yes	Yes	Yes	Yes	Yes
$[<identifier>]	Yes	Yes	Yes	Yes	Yes
$addToSet	Yes	Yes	Yes	Yes	Yes
$pop	Yes	Yes	Yes	Yes	Yes
$pullAll	Yes	Yes	Yes	Yes	Yes
$pull	Yes	Yes	Yes	Yes	Yes
$push	Yes	Yes	Yes	Yes	Yes

Bitwise operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$bit	Yes	Yes	Yes	Yes	Yes

Field operators

Operator	3.6	4.0	5.0	8.0	Elastic cluster
$currentDate	Yes	Yes	Yes	Yes	Yes
$inc	Yes	Yes	Yes	Yes	Yes

Operator	3.6	4.0	5.0	8.0	Elastic cluster
$max	Yes	Yes	Yes	Yes	Yes
$min	Yes	Yes	Yes	Yes	Yes
$mul	Yes	Yes	Yes	Yes	Yes
$rename	Yes	Yes	Yes	Yes	Yes
$set	Yes	Yes	Yes	Yes	Yes
$setOnInsert	Yes	Yes	Yes	Yes	Yes
$unset	Yes	Yes	Yes	Yes	Yes

Update modifiers

Operator	3.6	4.0	5.0	8.0	Elastic cluster
$each	Yes	Yes	Yes	Yes	Yes
$position	Yes	Yes	Yes	Yes	Yes
$slice	Yes	Yes	Yes	Yes	Yes
$sort	Yes	Yes	Yes	Yes	Yes

Geospatial

Geometry specifiers

Query Selectors	3.6	4.0	5.0	8.0	Elastic cluster
$box	No	No	No	No	No

Query Selectors	3.6	4.0	5.0	8.0	Elastic cluster
$center	No	No	No	No	No
$centerSphere	No	No	No	No	No
$geometry	Yes	Yes	Yes	Yes	Yes
$maxDistance	Yes	Yes	Yes	Yes	Yes
$minDistance	Yes	Yes	Yes	Yes	Yes
$nearSphere	Yes	Yes	Yes	Yes	Yes
$polygon	No	No	No	No	No
$uniqueDocs	No	No	No	No	No

Query selectors

Command	3.6	4.0	5.0	8.0	Elastic cluster
$geoIntersects	Yes	Yes	Yes	Yes	Yes
$geoWithin	Yes	Yes	Yes	Yes	Yes
$near	Yes	Yes	Yes	Yes	Yes
$nearSphere	Yes	Yes	Yes	Yes	Yes
$polygon	No	No	No	No	No
$uniqueDocs	No	No	No	No	No

Cursor methods

Command	3.6	4.0	5.0	8.0	Elastic cluster
cursor.batchSize()	Yes	Yes	Yes	Yes	Yes
cursor.close()	Yes	Yes	Yes	Yes	Yes
cursor.collation()	No	No	No	Yes	No
cursor.comment()	Yes	Yes	Yes	Yes	Yes
cursor.count()	Yes	Yes	Yes	Yes	Yes
cursor.explain()	Yes	Yes	Yes	Yes	No
cursor.forEach()	Yes	Yes	Yes	Yes	Yes
cursor.hasNext()	Yes	Yes	Yes	Yes	Yes
cursor.hint()	Yes	Yes	Yes	Yes	Yes*
cursor.isClosed()	Yes	Yes	Yes	Yes	Yes
cursor.isExhausted()	Yes	Yes	Yes	Yes	No
cursor.itcount()	Yes	Yes	Yes	Yes	No
cursor.limit()	Yes	Yes	Yes	Yes	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
cursor.map()	Yes	Yes	Yes	Yes	No
cursor.max()	No	No	Yes (5.0.2+)	Yes (8.0.1+)	No
cursor.maxScan()	Yes	Yes	Yes	Yes	No
cursor.maxTimeMS()	Yes	Yes	Yes	Yes	No
cursor.min()	No	No	Yes (5.0.2+)	Yes (8.0.1+)	No
cursor.next()	Yes	Yes	Yes	Yes	Yes
cursor.noCursorTimeout()	No	No	No	No	No
cursor.objsLeftInBatch()	Yes	Yes	Yes	Yes	No
cursor.pretty()	Yes	Yes	Yes	Yes	No
cursor.readConcern()	Yes	Yes	Yes	Yes	No
cursor.readPref()	Yes	Yes	Yes	Yes	No
cursor.returnKey()	No	No	No	No	No
cursor.showRecordId()	No	No	No	No	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
cursor.size()	Yes	Yes	Yes	Yes	No
cursor.skip()	Yes	Yes	Yes	Yes	No
cursor.sort()	Yes	Yes	Yes	Yes	No
cursor.tailable()	No	No	No	No	No
cursor.toArray()	Yes	Yes	Yes	Yes	No

* Index `hint` is supported with index expressions. For example, `db.foo.find().hint({x:1})`.

The `cursor.min()` and `cursor.max()` methods require a `hint` that names the index to use. Both methods must specify the same fields, and you can't combine them with a `$natural` sort. They're supported on the `find` command only.

Aggregation pipeline operators

Topics

- Accumulator expressions
- Arithmetic operators
- Array operators
- Boolean operators
- Comparison operators
- Conditional expression operators
- Data type operator
- Data size operator
- Date operators
- Literal operator
- Merge operator

- Natural operator
- Set operators
- Stage operators
- String operators
- System variables
- Text search operator
- Type conversion operators
- Variable operators
- Trigonometry operators
- Bitwise operators (aggregation)
- Timestamp operators
- Miscellaneous operators

Accumulator expressions

Expression	3.6	4.0	5.0	8.0	Elastic cluster
$accumulator	-	-	No	No	No
$addToSet	Yes	Yes	Yes	Yes	Yes
$avg	Yes	Yes	Yes	Yes	Yes
$bottom	-	-	-	Yes (8.0.1+)	No
$bottomN	-	-	-	Yes (8.0.1+)	No
$count	-	-	No	Yes (8.0.1+)	No
$covariancePop	No	No	No	Yes (8.0.2+)	No
$covarianceSamp	No	No	No	Yes (8.0.2+)	No

Expression	3.6	4.0	5.0	8.0	Elastic cluster
$denseRank	No	No	No	Yes (8.0.2+)	No
$derivative	No	No	No	No	No
$documentNumber	No	No	No	Yes (8.0.2+)	No
$expMovingAvg	No	No	No	No	No
$first	Yes	Yes	Yes	Yes	Yes
$firstN	-	-	-	Yes (8.0.1+)	No
$integral	No	No	No	No	No
$last	Yes	Yes	Yes	Yes	Yes
$lastN	-	-	-	Yes (8.0.1+)	No
$linearFill	No	No	No	No	No
$locf	No	No	No	No	No
$max	Yes	Yes	Yes	Yes	Yes
$maxN	-	-	-	Yes (8.0.1+)	No
$median	-	-	-	Yes (8.0.1+)	No
$min	Yes	Yes	Yes	Yes	Yes
$minN	-	-	-	Yes (8.0.1+)	No
$percentile	-	-	-	Yes (8.0.1+)	No
$push	Yes	Yes	Yes	Yes	Yes
$rank	No	No	No	Yes (8.0.2+)	No

Expression	3.6	4.0	5.0	8.0	Elastic cluster
$shift	No	No	No	No	No
$stdDevPop	No	No	No	Yes (8.0.1+)	No
$stdDevSamp	No	No	No	Yes (8.0.1+)	No
$sum	Yes	Yes	Yes	Yes	Yes
$top	-	-	-	Yes (8.0.1+)	No
$topN	-	-	-	Yes (8.0.1+)	No

Arithmetic operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$abs	Yes	Yes	Yes	Yes	Yes
$add	Yes	Yes	Yes	Yes	Yes
$ceil	No	Yes	Yes	Yes	Yes
$divide	Yes	Yes	Yes	Yes	Yes
$exp	No	Yes	Yes	Yes	Yes
$floor	No	Yes	Yes	Yes	Yes
$ln	No	Yes	Yes	Yes	Yes
$log	No	Yes	Yes	Yes	Yes
$log10	No	Yes	Yes	Yes	Yes
$mod	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$multiply	Yes	Yes	Yes	Yes	Yes
$pow	No	No	Yes (5.0.1+)	Yes	No
$round	-	-	No	Yes (8.0.1+)	No
$sqrt	No	Yes	Yes	Yes	Yes
$subtract	Yes	Yes	Yes	Yes	Yes
$trunc	No	No	No	Yes (8.0.1+)	No

Array operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$arrayElemAt	Yes	Yes	Yes	Yes	Yes
$arrayToObject	Yes	Yes	Yes	Yes	Yes
$concatArrays	Yes	Yes	Yes	Yes	Yes
$filter	Yes	Yes	Yes	Yes	Yes
$first	-	-	Yes	Yes	No
$in	Yes	Yes	Yes	Yes	Yes
$indexOfArray	Yes	Yes	Yes	Yes	Yes
$isArray	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$last	-	-	Yes	Yes	No
$objectToArray	Yes	Yes	Yes	Yes	Yes
$range	Yes	Yes	Yes	Yes	Yes
$reverseArray	Yes	Yes	Yes	Yes	Yes
$sortArray	No	No	No	Yes	No
$reduce	Yes	Yes	Yes	Yes	Yes
$size	Yes	Yes	Yes	Yes	Yes
$slice	Yes	Yes	Yes	Yes	Yes
$zip	Yes	Yes	Yes	Yes	Yes

Boolean operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$and	Yes	Yes	Yes	Yes	Yes
$not	Yes	Yes	Yes	Yes	Yes
$or	Yes	Yes	Yes	Yes	Yes

Comparison operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$cmp	Yes	Yes	Yes	Yes	Yes
$eq	Yes	Yes	Yes	Yes	Yes
$gt	Yes	Yes	Yes	Yes	Yes
$gte	Yes	Yes	Yes	Yes	Yes
$lt	Yes	Yes	Yes	Yes	Yes
$lte	Yes	Yes	Yes	Yes	Yes
$ne	Yes	Yes	Yes	Yes	Yes

Conditional expression operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$cond	Yes	Yes	Yes	Yes	Yes
$ifNull	Yes	Yes	Yes	Yes	Yes
$switch	No	Yes	Yes	Yes	No

Data type operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$type	Yes	Yes	Yes	Yes	Yes

Data size operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$binarySize	-	-	No	Yes (8.0.1+)	No
$bsonSize	-	-	No	Yes (8.0.1+)	No

Date operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$dateAdd	No	No	Yes	Yes	Yes
$dateDiff	-	-	Yes	Yes	No
$dateFromParts	No	No	Yes (5.0.1+)	No	No
$dateFromString	Yes	Yes	Yes	Yes	Yes
$dateSubtract	No	No	Yes	Yes	Yes
$dateToParts	No	No	Yes (5.0.1+)	No	No
$dateToString	Yes	Yes	Yes	Yes	Yes
$dateTrunc	-	-	No	Yes	No
$dayOfMonth	Yes	Yes	Yes	Yes	Yes
$dayOfWeek	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$dayOfYear	Yes	Yes	Yes	Yes	Yes
$hour	Yes	Yes	Yes	Yes	Yes
$isoDayOfWeek	Yes	Yes	Yes	Yes	Yes
$isoWeek	Yes	Yes	Yes	Yes	Yes
$isoWeekYear	Yes	Yes	Yes	Yes	Yes
$millisecond	Yes	Yes	Yes	Yes	Yes
$minute	Yes	Yes	Yes	Yes	Yes
$month	Yes	Yes	Yes	Yes	Yes
$second	Yes	Yes	Yes	Yes	Yes
$week	Yes	Yes	Yes	Yes	Yes
$year	Yes	Yes	Yes	Yes	Yes

Literal operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$literal	Yes	Yes	Yes	Yes	Yes

Merge operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$mergeObjects	Yes	Yes	Yes	Yes	Yes

Natural operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$natural	Yes	Yes	Yes	Yes	Yes

Set operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$allElementsTrue	No	Yes	Yes	Yes	Yes
$anyElementTrue	No	Yes	Yes	Yes	Yes
$setDifference	No	Yes	Yes	Yes	Yes
$setEquals	Yes	Yes	Yes	Yes	Yes
$setIntersection	Yes	Yes	Yes	Yes	Yes
$setIsSubset	Yes	Yes	Yes	Yes	Yes
$setUnion	Yes	Yes	Yes	Yes	Yes

Stage operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$addFields	Yes	Yes	Yes	Yes	Yes
$bucket	No	No	No	Yes	No
$bucketAuto	No	No	No	Yes (8.0.2+)	No
$changeStream	Yes	Yes	Yes	Yes	No
$changeStreamSplitLargeEvent	No	No	No	Yes (8.0.2+)	No
$collStats	No	Yes	Yes	Yes	No
$count	Yes	Yes	Yes	Yes	Yes
$currentOp	Yes	Yes	Yes	Yes	Yes
$densify	No	No	No	No	No
$facet	No	No	No	Yes (8.0.2+)	No
$fill	No	No	No	No	No
$geoNear	Yes	Yes	Yes	Yes	Yes
$graphLookup	No	No	No	Yes	No
$group	Yes	Yes	Yes	Yes	Yes
$indexStats	Yes	Yes	Yes	Yes	No
$limit	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$listLocalSessions	No	No	No	No	No
$listSessions	No	No	No	No	No
$lookup	Yes	Yes	Yes	Yes	Yes
$match	Yes	Yes	Yes	Yes	Yes
$merge	-	-	No	Yes	No
$out	Yes	Yes	Yes	Yes	No
$planCacheStats	-	-	No	No	No
$project	Yes	Yes	Yes	Yes	Yes
$redact	Yes	Yes	Yes	Yes	Yes
$replaceRoot	Yes	Yes	Yes	Yes	Yes
$sample	Yes	Yes	Yes	Yes	Yes
$set	-	-	No	Yes	No
$setWindowFields	-	-	No	Yes (8.0.2+)	No
$skip	Yes	Yes	Yes	Yes	Yes
$sort	Yes	Yes	Yes	Yes	Yes
$sortByCount	No	No	No	Yes (8.0.1+)	No
$unionWith	-	-	No	No	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
$unset	-	-	No	Yes	No
$unwind	Yes	Yes	Yes	Yes	Yes
$replaceWith	No	No	No	Yes	No
$vectorSearch	No	No	No	Yes	No
$listSearchIndexes	-	-	-	Yes (8.0.1+)	No

String operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$concat	Yes	Yes	Yes	Yes	Yes
$indexOfBytes	Yes	Yes	Yes	Yes	Yes
$indexOfCP	Yes	Yes	Yes	Yes	Yes
$ltrim	No	Yes	Yes	Yes	No
$regexFind	-	-	Yes	Yes	No
$regexFindAll	-	-	Yes	Yes	No
$regexMatch	-	-	Yes	Yes	No
$replaceAll	-	-	Yes	Yes	No
$replaceOne	-	-	Yes	Yes	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
$rtrim	No	Yes	Yes	Yes	No
$split	Yes	Yes	Yes	Yes	Yes
$strcasecmp	Yes	Yes	Yes	Yes	Yes
$strLenBytes	Yes	Yes	Yes	Yes	Yes
$strLenCP	Yes	Yes	Yes	Yes	Yes
$substr	Yes	Yes	Yes	Yes	Yes
$substrBytes	Yes	Yes	Yes	Yes	Yes
$substrCP	Yes	Yes	Yes	Yes	Yes
$toLower	Yes	Yes	Yes	Yes	Yes
$toUpper	Yes	Yes	Yes	Yes	Yes
$trim	No	Yes	Yes	Yes	No

System variables

Command	3.6	4.0	5.0	8.0	Elastic cluster
$$CURRENT	No	No	No	No	No
$$DESCEND	Yes	Yes	Yes	Yes	Yes
$$KEEP	Yes	Yes	Yes	Yes	Yes
$$PRUNE	Yes	Yes	Yes	Yes	Yes
$$REMOVE	No	No	No	No	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
$ROOT	Yes	Yes	Yes	Yes	Yes

Text search operator

Command	3.6	4.0	5.0	8.0	Elastic cluster
$meta	No	No	Yes	Yes	No
$search	No	No	Yes	Yes	No

Type conversion operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$convert	No	Yes	Yes	Yes	Yes
$isNumber	-	-	No	Yes (8.0.1+)	No
$toBool	No	Yes	Yes	Yes	Yes
$toDate	No	Yes	Yes	Yes	Yes
$toDecimal	No	Yes	Yes	Yes	Yes
$toDouble	No	Yes	Yes	Yes	Yes
$toInt	No	Yes	Yes	Yes	Yes
$toLong	No	Yes	Yes	Yes	Yes
$toObjectId	No	Yes	Yes	Yes	Yes
$toString	No	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
$toUUID	-	-	-	Yes (8.0.1+)	No

Variable operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$let	Yes	Yes	Yes	Yes	Yes
$map	Yes	Yes	Yes	Yes	Yes

Trigonometry operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$acos	-	-	-	Yes (8.0.1+)	No
$acosh	-	-	-	Yes (8.0.1+)	No
$asin	-	-	-	Yes (8.0.1+)	No
$asinh	-	-	-	Yes (8.0.1+)	No
$atan	-	-	-	Yes (8.0.1+)	No
$atan2	-	-	-	Yes (8.0.1+)	No
$atanh	-	-	-	Yes (8.0.1+)	No
$cos	-	-	-	Yes (8.0.1+)	No
$cosh	-	-	-	Yes (8.0.1+)	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
$degreesToRadians	-	-	-	Yes (8.0.1+)	No
$radiansToDegrees	-	-	-	Yes (8.0.1+)	No
$sin	-	-	-	Yes (8.0.1+)	No
$sinh	-	-	-	Yes (8.0.1+)	No
$tan	-	-	-	Yes (8.0.1+)	No
$tanh	-	-	-	Yes (8.0.1+)	No

Bitwise operators (aggregation)

Command	3.6	4.0	5.0	8.0	Elastic cluster
$bitAnd	-	-	-	Yes (8.0.1+)	No
$bitNot	-	-	-	Yes (8.0.1+)	No
$bitOr	-	-	-	Yes (8.0.1+)	No
$bitXor	-	-	-	Yes (8.0.1+)	No

Timestamp operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$tsIncrement	-	-	-	Yes (8.0.1+)	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
$tsSecond	-	-	-	Yes (8.0.1+)	No

Miscellaneous operators

Command	3.6	4.0	5.0	8.0	Elastic cluster
$getField	-	-	Yes	Yes	No
$rand	-	-	Yes (5.0.1+)	Yes	No
$sampleRate	-	-	No	Yes (8.0.1+)	No
$sigmoid	-	-	-	Yes (8.0.1+)	No

Data types

Command	3.6	4.0	5.0	8.0	Elastic cluster
32-bit Integer (int)	Yes	Yes	Yes	Yes	Yes
64-bit Integer (lon	Yes	Yes	Yes	Yes	Yes
Array	Yes	Yes	Yes	Yes	Yes
Binary Data	Yes	Yes	Yes	Yes	Yes
Boolean	Yes	Yes	Yes	Yes	Yes
Date	Yes	Yes	Yes	Yes	Yes

Command	3.6	4.0	5.0	8.0	Elastic cluster
DBPointer	No	No	No	No	No
DBRefs	No	No	No	No	No
Decimal128	Yes	Yes	Yes	Yes	Yes
Double	Yes	Yes	Yes	Yes	Yes
JavaScript	No	No	No	No	No
JavaScript (with scope)	No	No	No	No	No
MaxKey	Yes	Yes	Yes	Yes	Yes
MinKey	Yes	Yes	Yes	Yes	Yes
Null	Yes	Yes	Yes	Yes	Yes
Object	Yes	Yes	Yes	Yes	Yes
ObjectId	Yes	Yes	Yes	Yes	Yes
Regular Expression	Yes	Yes	Yes	Yes	Yes
String	Yes	Yes	Yes	Yes	Yes
Symbol	No	No	No	No	No
Timestamp	Yes	Yes	Yes	Yes	Yes
Undefined	No	No	No	No	No

Indexes and index properties

Topics

- Indexes
- Index properties

Indexes

Command	3.6	4.0	5.0	8.0	Elastic cluster
2dsphere	Yes	Yes	Yes	Yes	Yes
2d Index	No	No	No	No	No
Compound Index	Yes	Yes	Yes	Yes	Yes
Hashed Index	No	No	No	No	No
Multikey Index	Yes	Yes	Yes	Yes	Yes
Single Field Index	Yes	Yes	Yes	Yes	Yes
Text Index	No	No	Yes	Yes	No
Wildcard	No	No	No	Yes (8.0.3+)	No

Index properties

Command	3.6	4.0	5.0	8.0	Elastic cluster
Background	Yes	Yes	Yes	Yes	Yes
Case Insensitive	No	No	No	Yes	No
Hidden	No	No	No	Yes (8.0.1+)	No

Command	3.6	4.0	5.0	8.0	Elastic cluster
Partial	No	No	Yes	Yes	No
Sparse	Yes	Yes	Yes	Yes	Yes
Text	No	No	Yes	Yes	No
TTL	Yes	Yes	Yes	Yes	Yes
Unique	Yes	Yes	Yes	Yes	Yes
Vector	No	No	Yes	Yes	No

For detailed information about specific MongoDB operators, see the following topics:

- Aggregation pipeline operators
- Geospatial
- Projection operators
- Update operators

Aggregation pipeline operators

This section provides detailed information about aggregation pipeline operators supported by Amazon DocumentDB.

Topics

- $$DESCEND
- $$KEEP
- $$PRUNE
- $ROOT
- $abs
- $acos
- $acosh
- $add

- $addToSet
- $addFields
- $allElementsTrue
- $and
- $anyElementTrue
- $arrayElemAt
- $arrayToObject
- $asin
- $asinh
- $atan
- $atan2
- $atanh
- $avg
- $binarySize
- $bitAnd
- $bitNot
- $bitOr
- $bitXor
- $bottom
- $bottomN
- $bsonSize
- $bucket
- $bucketAuto
- $ceil
- $changeStream
- $changeStreamSplitLargeEvent
- $cmp
- $collStats
- $concat
- $concatArrays

- $cond
- $convert
- $cos
- $cosh
- $count (stage)
- $count (accumulator)
- $covariancePop
- $covarianceSamp
- $currentOp
- $dateAdd
- $dateDiff
- $dateFromParts
- $dateFromString
- $dateSubtract
- $dateToParts
- $dateToString
- $dateTrunc
- $dayOfMonth
- $dayOfWeek
- $dayOfYear
- $degreesToRadians
- $denseRank
- $divide
- $documentNumber
- $eq
- $exp
- $facet
- $filter
- $first
- $firstN

- $floor
- $geoNear
- $graphLookup
- $group
- $gt
- $gte
- $hour
- $ifNull
- $in
- $indexOfArray
- $indexOfBytes
- $indexOfCP
- $indexStats
- $isArray
- $isNumber
- $isoDayOfWeek
- $isoWeek
- $isoWeekYear
- $last
- $lastN
- $let
- $limit
- $listSearchIndexes
- $literal
- $ln
- $log
- $log10
- $lt
- $lte
- $lookup

- $ltrim
- $map
- $match
- $max
- $maxN
- $median
- $meta
- $merge
- $mergeObjects
- $millisecond
- $min
- $minN
- $minute
- $mod
- $month
- $multiply
- $natural
- $ne
- $not
- $objectToArray
- $or
- $out
- $percentile
- $pow
- $push
- $project
- $radiansToDegrees
- $rand
- $range
- $rank

- $redact
- $reduce
- $regexFind
- $regexFindAll
- $regexMatch
- $replaceAll
- $replaceOne
- $replaceRoot
- $replaceWith
- $reverseArray
- $round
- $rtrim
- $sample
- $search
- $second
- $set
- $setDifference
- $setEquals
- $setIntersection
- $setIsSubset
- $setUnion
- $setWindowFields
- $sigmoid
- $sin
- $sinh
- $skip
- $slice
- $size
- $sort
- $sortArray

- $sortByCount
- $split
- $sqrt
- $stdDevPop
- $stdDevSamp
- $strLenBytes
- $strLenCP
- $strcasecmp
- $substr
- $substrBytes
- $substrCP
- $subtract
- $sum
- $switch
- $tan
- $tanh
- $top
- $topN
- $trunc
- $tsIncrement
- $tsSecond
- $toBool
- $toDate
- $toDecimal
- $toDouble
- $toInt
- $toLong
- $toLower
- $toObjectId
- $toString

- $toUpper
- $toUUID
- $trim
- $type
- $unionWith
- $unset
- $unwind
- $vectorSearch
- $week
- $year
- $zip

$$DESCEND

The $$DESCEND operator in Amazon DocumentDB is a special positional array operator used within the $redact pipeline stage. It instructs the aggregation pipeline to descend into the current document and process all fields, regardless of their nesting level.

When the $redact stage encounters the $$DESCEND operator, it will keep all the fields in the current document visible and process them further down the pipeline. This is useful when you want to selectively redact or prune certain fields based on a condition, while retaining the structure of the document.

Parameters

None.

Example (MongoDB Shell)

In this example, we'll use the $redact stage with the $$DESCEND operator to selectively display documents where the code field is equal to "Reg".

Create sample documents

```
db.patient.insertMany([
  { "_id": 1, "code": "Emp", "patient": "John Doe", "DOB": "1/1/1980", "Hospital":
 "Main" },
```

```
  { "_id": 2, "code": "Reg", "patient": "Jane Doe", "DOB": "3/27/1989", "Hospital":
 "Cherry Hill" },
  { "_id": 3, "code": "Emp", "patient": "Bob Smith", "DOB": "6/15/1975", "Hospital":
 "Downtown" }
]);
```

Query example

```
db.patient.aggregate([
  { $redact: {
    $cond: {
      if: { $eq: ["Reg", "$code"] },
      then: "$$DESCEND",
      else: "$$PRUNE"
    }
  }}
]);
```

Output

```
{
  "_id": 2,
  "code": "Reg",
  "patient": "Jane Doe",
  "DOB": "3/27/1989",
  "Hospital": "Cherry Hill"
}
```

Code examples

To view a code example for using the $$DESCEND command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function redactPatients() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const collection = db.collection('patient');

  const result = await collection.aggregate([
    { $redact: {
      $cond: {
        if: { $eq: ["Reg", "$code"] },
        then: "$$DESCEND",
        else: "$$PRUNE"
      }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

redactPatients();
```

Python

```
from pymongo import MongoClient

def redact_patients():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.patient

    result = list(collection.aggregate([
        { "$redact": {
            "$cond": {
                "if": { "$eq": ["Reg", "$code"] },
                "then": "$$DESCEND",
                "else": "$$PRUNE"
            }
        }}
    ]))

    print(result)
    client.close()

redact_patients()
```

$$KEEP

The $$KEEP system variable is used with the $redact stage in the aggregation pipeline to keep the current document or field unchanged and include it in the output.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates the usage of $$KEEP in an Amazon DocumentDB aggregation pipeline. Documents are only kept if access equals "public", otherwise they are removed.

Create sample documents

```
db.articles.insertMany([
  { title: "Article A", access: "public", content: "Visible content" },
  { title: "Article B", access: "private", content: "Hidden content" }
]);
```

Query example

```
db.articles.aggregate([
  {
    $redact: {
      $cond: [
        { $eq: ["$access", "public"] },
        "$$KEEP",
        "$$PRUNE"
      ]
    }
  }
]);
```

Output

```
[
  {
```

```
    "_id" : ObjectId("..."),
    "title" : "Article A",
    "access" : "public",
    "content" : "Visible content"
  }
]
```

Code examples

To view a code example for using the $$KEEP command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function run() {
  const client = new MongoClient(
    'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-bundle.pem&replicaSet=rs0'
  );

  try {
    await client.connect();
    const db = client.db('test');
    const articles = db.collection('articles');

    const pipeline = [
      {
        $redact: {
          $cond: [
            { $eq: ["$access", "public"] },
            "$$KEEP",
            "$$PRUNE"
          ]
        }
      }
    ];

    const results = await articles.aggregate(pipeline).toArray();
    console.log(results);
  } finally {
    await client.close();
```

```
  }
}

run().catch(console.error);
```

Python

```
from pymongo import MongoClient

client = MongoClient(
    "mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-bundle.pem&replicaSet=rs0"
)

db = client.test
articles = db.articles

pipeline = [
    {
        "$redact": {
            "$cond": [
                {"$eq": ["$access", "public"]},
                "$$KEEP",
                "$$PRUNE"
            ]
        }
    }
]

results = list(articles.aggregate(pipeline))
print(results)

client.close()
```

$$PRUNE

The $$PRUNE system variable is used with the $redact stage in the aggregation pipeline to exclude documents or embedded document levels from the results. When a condition evaluates to $$PRUNE, the current document or subdocument is removed from the output. It is typically used with $$DESCEND (to keep and traverse the document) or $$KEEP (to keep the document at all levels).

Parameters

None. The $$PRUNE system variable is used without any parameters and must be used with `$redact`.

Example (MongoDB Shell)

The following example demonstrates how to use $$PRUNE with `$redact` to exclude users over 30 years old from the results.

Create sample documents

```
db.users.insert([
  { _id:1, name: "Carlos Salazar", age: 35, address: { street: "123 Main St", city:
 "Anytown", state: "CA" } },
  { _id:2, name: "Saanvi Sarkar", age: 28, address: { street: "456 Oak Rd", city:
 "Someplace", state: "NY" } },
  { _id:3, name: "Li Juan", age: 42, address: { street: "789 Pine Ave", city:
 "Springfield", state: "TX" } }
])
```

Query example

```
db.users.aggregate([
  {
    $redact: {
      $cond: {
        if: { $gt: ["$age", 30] },
        then: "$$PRUNE",
        else: "$$DESCEND"
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 2,
    "name": "Saanvi Sarkar",
    "age": 28,
```

```
    "address": {
      "street": "456 Oak Rd",
      "city": "Someplace",
      "state": "NY"
    }
  }
]
```

Code examples

To view a code example for using the $$PRUNE command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

async function main() {
  await client.connect();
  const db = client.db('test');
  const users = db.collection('users');

  const result = await users.aggregate([
    {
      $redact: {
        $cond: {
          if: { $gt: ["$age", 30] },
          then: "$$PRUNE",
          else: "$$DESCEND"
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
users = db['users']

result = list(users.aggregate([
    {
        '$redact': {
            '$cond': {
                'if': { '$gt': ['$age', 30] },
                'then': '$$PRUNE',
                'else': '$$DESCEND'
            }
        }
    }
]))

print(result)
client.close()
```

$ROOT

The $ROOT operator in Amazon DocumentDB is used to reference the entire input document within an aggregation pipeline. It allows you to access and manipulate the complete document, including all its nested fields and structures.

Parameters

None

Example (MongoDB Shell)

This example demonstrates using $ROOT to create an audit log that captures the complete original document along with metadata about when it was processed.

Create sample documents

```
db.orders.insertMany([
  {
    _id: 1,
    orderId: "ORD-2024-001",
    customer: "María García",
    email: "maría@example.com",
    items: [
      { product: "Laptop", quantity: 1, price: 1299.99 }
    ],
    totalAmount: 1299.99
  },
  {
    _id: 2,
    orderId: "ORD-2024-002",
    customer: "Arnav Desai",
    email: "arnav@example.com",
    items: [
      { product: "Mouse", quantity: 2, price: 29.99 },
      { product: "Keyboard", quantity: 1, price: 89.99 }
    ],
    totalAmount: 149.97
  }
]);
```

Query example

```
db.orders.aggregate([
  {
    $project: {
      processedAt: new Date(),
      originalDocument: "$$ROOT",
      summary: {
        $concat: [
          "Order ",
          "$orderId",
          " for ",
          "$customer",
          " - Total: $",
          { $toString: "$totalAmount" }
        ]
      }
    }
  }
```

```
]);
```

Output

```
[
  {
    _id: 1,
    processedAt: ISODate('2025-11-24T20:43:51.492Z'),
    originalDocument: {
      _id: 1,
      orderId: 'ORD-2024-001',
      customer: 'María García',
      email: 'maría@example.com',
      items: [ { product: 'Laptop', quantity: 1, price: 1299.99 } ],
      totalAmount: 1299.99
    },
    summary: 'Order ORD-2024-001 for María García - Total: $1299.99'
  },
  {
    _id: 2,
    processedAt: ISODate('2025-11-24T20:43:51.492Z'),
    originalDocument: {
      _id: 2,
      orderId: 'ORD-2024-002',
      customer: 'Arnav Desai',
      email: 'arnav@example.com',
      items: [
        { product: 'Mouse', quantity: 2, price: 29.99 },
        { product: 'Keyboard', quantity: 1, price: 89.99 }
      ],
      totalAmount: 149.97
    },
    summary: 'Order ORD-2024-002 for Arnav Desai - Total: $149.97'
  }
]
```

Code examples

To view a code example for using the $ROOT command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function createAuditLog() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const orders = db.collection('orders');

  const result = await orders.aggregate([
    {
      $project: {
        processedAt: new Date(),
        originalDocument: "$$ROOT",
        summary: {
          $concat: [
            "Order ",
            "$orderId",
            " for ",
            "$customer",
            " - Total: $",
            { $toString: "$totalAmount" }
          ]
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

createAuditLog();
```

Python

```
from pymongo import MongoClient
from datetime import datetime

def create_audit_log():
```

```
    client = MongoClient('mongodb://username:password@docdb-
cluster.cluster-123456789.us-east-1.docdb.amazonaws.com:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['orders']

    result = list(collection.aggregate([
        {
            '$project': {
                'processedAt': datetime.now(),
                'originalDocument': '$$ROOT',
                'summary': {
                    '$concat': [
                        "Order ",
                        "$orderId",
                        " for ",
                        "$customer",
                        " - Total: $",
                        { '$toString': "$totalAmount" }
                    ]
                }
            }
        }
    ]))

    print(result)
    client.close()

create_audit_log()
```

$abs

The $abs operator in Amazon DocumentDB returns the absolute value of a number. It can be used in the aggregation pipeline to perform mathematical operations on numeric fields.

Parameters

- number: The numeric expression for which the absolute value will be returned.

Example (MongoDB Shell)

This example demonstrates the usage of the $abs operator to find the absolute value of a numeric field.

Create sample documents

```
db.numbers.insertMany([
  { "_id": 1, "value": -5 },
  { "_id": 2, "value": 10 },
  { "_id": 3, "value": -3.14 },
  { "_id": 4, "value": 0 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    "_id": 1,
    "absolute_value": { $abs: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "absolute_value": 5 },
  { "_id": 2, "absolute_value": 10 },
  { "_id": 3, "absolute_value": 3.14 },
  { "_id": 4, "absolute_value": 0 }
]
```

Code examples

To view a code example for using the $abs command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      "_id": 1,
      "absolute_value": { $abs: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['numbers']

    result = list(collection.aggregate([
        { '$project': {
            "_id": 1,
            "absolute_value": { "$abs": "$value" }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$acos

New from version 8.0.1.

The `$acos` operator in Amazon DocumentDB returns the arccosine (inverse cosine) of a value. The input value must be between -1 and 1.

Parameters

- `expression`: An expression that resolves to a number between -1 and 1.

The result is in radians. To obtain degrees, apply `$radiansToDegrees` to the output.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the `$acos` operator to calculate the arccosine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 1 },
  { "_id": 2, "value": 0 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "angle": { $acos: "$value" }
  }}
]);
```

Output

```
[
```

```
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 1.5707963267948966 },
  { "_id": 3, "angle": 3.141592653589793 }
]
```

Code examples

To view a code example for using the $acos operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "angle": { $acos: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']
```

```
    result = list(collection.aggregate([
        { '$project': {
            'angle': { '$acos': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$acosh

New from version 8.0.1.

The `$acosh` operator in Amazon DocumentDB returns the inverse hyperbolic cosine (hyperbolic arccosine) of a value. The input value must be greater than or equal to 1.

Parameters

- `expression`: An expression that resolves to a number greater than or equal to 1.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the `$acosh` operator to calculate the inverse hyperbolic cosine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 1 },
  { "_id": 2, "value": 2 },
  { "_id": 3, "value": 10 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $acosh: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "result": 0 },
  { "_id": 2, "result": 1.3169578969248166 },
  { "_id": 3, "result": 2.993222846126381 }
]
```

Code examples

To view a code example for using the $acosh operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "result": { $acosh: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$acosh': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$add

The $add operator in Amazon DocumentDB is used to add numbers or dates together. It can be used to perform arithmetic operations on numeric fields or to perform date arithmetic by adding a number of time units to a date field.

Parameters

- `expression1`: The first number or date to add.
- `expression2`: The second number or date to add.
- `expression3`: (optional) Additional numbers or dates to add.

Example (MongoDB Shell)

The following example demonstrates how to use the $add operator to add two numbers together.

Create sample documents

```
db.numbers.insertMany([
  { _id: 1, a: 5, b: 3 },
  { _id: 2, a: 10, b: 7 },
  { _id: 3, a: 2, b: 8 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    _id: 1,
    sum: { $add: ["$a", "$b"] }
  }}
])
```

Output

```
[
  { "_id" : 1, "sum" : 8 },
  { "_id" : 2, "sum" : 17 },
  { "_id" : 3, "sum" : 10 }
]
```

In this example, the $add operator is used to add the values of the a and b fields for each document and store the result in the sum field.

Code examples

To view a code example for using the $add command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');
```

```
  const result = await collection.aggregate([
    { $project: {
      _id: 1,
      sum: { $add: ['$a', '$b'] }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['numbers']

result = list(collection.aggregate([
    { '$project': {
        '_id': 1,
        'sum': { '$add': ['$a', '$b'] }
    }}
]))

print(result)
client.close()
```

$addToSet

The `$addToSet` aggregation operator returns an array of unique values from a specified expression for each group. It is used within the `$group` stage to accumulate distinct values, automatically eliminating duplicates.

Parameters

- `expression`: The expression to evaluate for each document in the group.

Example (MongoDB Shell)

The following example demonstrates using the $addToSet operator to collect unique cities where orders were placed for each customer.

Create sample documents

```
db.orders.insertMany([
  { _id: 1, customer: "Alice", city: "Seattle", amount: 100 },
  { _id: 2, customer: "Alice", city: "Portland", amount: 150 },
  { _id: 3, customer: "Bob", city: "Seattle", amount: 200 },
  { _id: 4, customer: "Alice", city: "Seattle", amount: 75 },
  { _id: 5, customer: "Bob", city: "Boston", amount: 300 }
]);
```

Query example

```
db.orders.aggregate([
  {
    $group: {
      _id: "$customer",
      cities: { $addToSet: "$city" }
    }
  }
]);
```

Output

```
[
  { _id: 'Bob', cities: [ 'Seattle', 'Boston' ] },
  { _id: 'Alice', cities: [ 'Seattle', 'Portland' ] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The $addToSet operator can also be used as a window operator in the $setWindowFields stage. In this context, it returns an array of the unique values of the specified expression for the documents in each window. You specify the operator under the output field, and optionally define the window boundaries with a window document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$addToSet` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.visits.insertMany([
  { _id: 1, customer: "Alice", day: 1, city: "Seattle" },
  { _id: 2, customer: "Alice", day: 2, city: "Portland" },
  { _id: 3, customer: "Alice", day: 3, city: "Seattle" },
  { _id: 4, customer: "Bob", day: 1, city: "Boston" },
  { _id: 5, customer: "Bob", day: 2, city: "Boston" }
]);
```

Query example

The following example partitions the documents by `customer`, sorts each partition by `day`, and returns the set of unique `city` values from the start of the partition through the current document.

```
db.visits.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$customer",
      sortBy: { day: 1 },
      output: {
        citiesSoFar: {
          $addToSet: "$city",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
```

```
  { "_id": 1, "customer": "Alice", "day": 1, "city": "Seattle", "citiesSoFar":
 [ "Seattle" ] },
  { "_id": 2, "customer": "Alice", "day": 2, "city": "Portland", "citiesSoFar":
 [ "Portland", "Seattle" ] },
  { "_id": 3, "customer": "Alice", "day": 3, "city": "Seattle", "citiesSoFar":
 [ "Portland", "Seattle" ] },
  { "_id": 4, "customer": "Bob", "day": 1, "city": "Boston", "citiesSoFar":
 [ "Boston" ] },
  { "_id": 5, "customer": "Bob", "day": 2, "city": "Boston", "citiesSoFar":
 [ "Boston" ] }
]
```

Each document is augmented with `citiesSoFar`, the set of unique `city` values within its partition from the start through the current document. As with the accumulator form, the order of values in the returned array is not guaranteed.

Code examples

To view a code example for using the `$addToSet` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  // Accumulator usage: collect unique values per group
  const orders = db.collection('orders');
  await orders.insertMany([
    { _id: 1, customer: "Alice", city: "Seattle", amount: 100 },
    { _id: 2, customer: "Alice", city: "Portland", amount: 150 },
    { _id: 3, customer: "Bob", city: "Seattle", amount: 200 },
    { _id: 4, customer: "Alice", city: "Seattle", amount: 75 },
    { _id: 5, customer: "Bob", city: "Boston", amount: 300 }
  ]);
  const accumulatorResult = await orders.aggregate([
    {
```

```
      $group: {
        _id: "$customer",
        cities: { $addToSet: "$city" }
      }
    }
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Window operator usage: running set of unique values within each partition
  const visits = db.collection('visits');
  await visits.insertMany([
    { _id: 1, customer: "Alice", day: 1, city: "Seattle" },
    { _id: 2, customer: "Alice", day: 2, city: "Portland" },
    { _id: 3, customer: "Alice", day: 3, city: "Seattle" },
    { _id: 4, customer: "Bob", day: 1, city: "Boston" },
    { _id: 5, customer: "Bob", day: 2, city: "Boston" }
  ]);
  const windowResult = await visits.aggregate([
    {
      $setWindowFields: {
        partitionBy: "$customer",
        sortBy: { day: 1 },
        output: {
          citiesSoFar: {
            $addToSet: "$city",
            window: { documents: ["unbounded", "current"] }
          }
        }
      }
    }
  ]).toArray();
  console.log('Window result:', windowResult);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']

    # Accumulator usage: collect unique values per group
    orders = db['orders']
    orders.insert_many([
        { '_id': 1, 'customer': 'Alice', 'city': 'Seattle', 'amount': 100 },
        { '_id': 2, 'customer': 'Alice', 'city': 'Portland', 'amount': 150 },
        { '_id': 3, 'customer': 'Bob', 'city': 'Seattle', 'amount': 200 },
        { '_id': 4, 'customer': 'Alice', 'city': 'Seattle', 'amount': 75 },
        { '_id': 5, 'customer': 'Bob', 'city': 'Boston', 'amount': 300 }
    ])
    accumulator_result = list(orders.aggregate([
        {
            '$group': {
                '_id': '$customer',
                'cities': { '$addToSet': '$city' }
            }
        }
    ]))
    print('Accumulator result:', accumulator_result)

    # Window operator usage: running set of unique values within each partition
    visits = db['visits']
    visits.insert_many([
        { '_id': 1, 'customer': 'Alice', 'day': 1, 'city': 'Seattle' },
        { '_id': 2, 'customer': 'Alice', 'day': 2, 'city': 'Portland' },
        { '_id': 3, 'customer': 'Alice', 'day': 3, 'city': 'Seattle' },
        { '_id': 4, 'customer': 'Bob', 'day': 1, 'city': 'Boston' },
        { '_id': 5, 'customer': 'Bob', 'day': 2, 'city': 'Boston' }
    ])
    window_result = list(visits.aggregate([
        {
            '$setWindowFields': {
                'partitionBy': '$customer',
                'sortBy': { 'day': 1 },
                'output': {
                    'citiesSoFar': {
                        '$addToSet': '$city',
                        'window': { 'documents': ['unbounded', 'current'] }
                    }
                }
```

```
            }
        }
    ]))
    print('Window result:', window_result)

    client.close()

example()
```

$addFields

The `$addFields` stage in the Amazon DocumentDB aggregation pipeline allows you to add new computed fields to documents. This can be useful for adding derived or transformed data to the documents.

Parameters

- `newField`: The name of the new field to add.
- `expression`: An expression that resolves to the value of the new field.

Example (MongoDB Shell)

The following example demonstrates how to use `$addFields` to add a new field `TotalInventory` that calculates the total inventory based on the `Inventory.OnHand` and `Inventory.OrderQnty` fields.

Create sample documents

```
db.example.insertMany([
  {
    "Item": "Spray Paint",
    "Colors": ["Black", "Red", "Green", "Blue"],
    "Inventory": {
      "OnHand": 47,
      "MinOnHand": 50,
      "OrderQnty": 36
    },
    "UnitPrice": 3.99
  },
  {
```

```
    "Item": "Ruler",
    "Colors": ["Red", "Green", "Blue", "Clear", "Yellow"],
    "Inventory": {
      "OnHand": 47,
      "MinOnHand": 40
    },
    "UnitPrice": 0.89
  }
]);
```

Query example

```
db.example.aggregate([
  {
    $addFields: {
      TotalInventory: { $add: ["$Inventory.OnHand", "$Inventory.OrderQnty"] }
    }
  }
])
```

Output

```
[
  {
    "_id" : ObjectId("5bedafbcf65ff161707de24f"),
    "Item" : "Ruler",
    "Colors" : [ "Red", "Green", "Blue", "Clear", "Yellow" ],
    "Inventory" : {
      "OnHand" : 47,
      "MinOnHand" : 40
    },
    "UnitPrice" : 0.89,
    "TotalInventory" : 47
  },
  {
    "_id" : ObjectId("5bedafbcf65ff161707de250"),
    "Item" : "Spray Paint",
    "Colors" : [ "Black", "Red", "Green", "Blue" ],
    "Inventory" : {
      "OnHand" : 47,
      "MinOnHand" : 50,
      "OrderQnty" : 36
    },
```

```
    "UnitPrice" : 3.99,
    "TotalInventory" : 83
  }
]
```

Code examples

To view a code example for using the $addFields command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
const db = client.db('test');
const collection = db.collection('example');

const result = await collection.aggregate([
  {
    $addFields: {
      TotalInventory: { $add: ['$Inventory.OnHand', '$Inventory.OrderQnty'] }
    }
  }
]).toArray();

console.log(result);

await client.close();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['example']

result = list(collection.aggregate([
```

```
    {
        '$addFields': {
            'TotalInventory': { '$add': ['$Inventory.OnHand',
 '$Inventory.OrderQnty'] }
        }
    }
]))

print(result)
client.close()
```

$allElementsTrue

New from version 4.0

The `$allElementsTrue` operator is used to check if all elements in an array evaluate to a true value.

Parameters

- `expression`: An expression that evaluates to an array.

Example (MongoDB Shell)

The following example demonstrates the usage of `$allElementsTrue` to check if all elements in an array are true.

Create sample documents

```
db.collection.insert([
  { "name": "John", "scores": [100, 90, 80] },
  { "name": "Jane", "scores": [80, 85, 0] },
  { "name": "Bob", "scores": [90, 95, null] }
])
```

Query example

```
db.collection.find({
  "scores": { "$allElementsTrue": [{ "$gt": 0 }] }
})
```

Output

```
[
  { "_id" : ObjectId("..."), "name" : "John", "scores" : [ 100, 90, 80 ] },
  { "_id" : ObjectId("..."), "name" : "Bob", "scores" : [ 90, 95, null ] }
]
```

In this example, the query checks if all elements in the `scores` array are greater than 0. The document with `"name": "Jane"` is excluded because the `scores` array contains a 0, which is a falsy value.

Code examples

To view a code example for using the `$allElementsTrue` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.find({
    "scores": { "$allElementsTrue": [{ "$gt": 0 }] }
  }).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['collection']

    result = list(collection.find({
        "scores": {"$allElementsTrue": [{"$gt": 0}]}
    }))

    print(result)

    client.close()

example()
```

$and

The $and aggregation operator evaluates multiple expressions and returns `true` only if all expressions evaluate to `true`. If any expression is `false`, it returns `false`.

Parameters

- `expressions`: An array of expressions to evaluate.

Example (MongoDB Shell)

The following example demonstrates using the $and operator to check if products meet multiple criteria.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Laptop", price: 1200, inStock: true },
  { _id: 2, name: "Mouse", price: 25, inStock: false },
  { _id: 3, name: "Keyboard", price: 75, inStock: true }
]);
```

Query example

```
db.products.aggregate([
  {
    $project: {
      name: 1,
      price: 1,
      inStock: 1,
      affordable: {
        $and: [
          { $lt: ["$price", 100] },
          { $eq: ["$inStock", true] }
        ]
      }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Laptop', price: 1200, inStock: true, affordable: false },
  { _id: 2, name: 'Mouse', price: 25, inStock: false, affordable: false },
  { _id: 3, name: 'Keyboard', price: 75, inStock: true, affordable: true }
]
```

Code examples

To view a code example for using the $and aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const result = await collection.aggregate([
    {
```

```
      $project: {
        name: 1,
        price: 1,
        inStock: 1,
        affordable: {
          $and: [
            { $lt: ["$price", 100] },
            { $eq: ["$inStock", true] }
          ]
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'price': 1,
                'inStock': 1,
                'affordable': {
                    '$and': [
                        { '$lt': ['$price', 100] },
                        { '$eq': ['$inStock', True] }
                    ]
                }
```

```
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$anyElementTrue

New from version 4.0

The `$anyElementTrue` operator is used to determine if any element in an array is true.

Parameters

- `field`: An array field to evaluate.

Example (MongoDB Shell)

The following example demonstrates the usage of `$anyElementTrue` to check if any element in an array is true.

Create sample documents

```
db.grades.insertMany([
  { _id: 1, student: 'Tim', scores: [false, false, null] },
  { _id: 2, student: 'Bob', scores: [false, 0, false] },
  { _id: 3, student: 'Ivy', scores: [false, true, 0] }
])
```

Query example

```
db.grades.aggregate([
  {
    $project: {
      student: 1,
      isAnyTrue: { $anyElementTrue: ["$scores"] },
      _id: 0
```

```
    }
  }
])
```

Output

```
[
  { student: 'Tim', isAnyTrue: false },
  { student: 'Bob', isAnyTrue: false },
  { student: 'Ivy', isAnyTrue: true }
]
```

Code examples

To view a code example for using the `$anyElementTrue` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('grades');

  const result = await collection.aggregate([
    {
      $project: {
        student: 1,
        isAnyTrue: { $anyElementTrue: ["$scores"] },
        _id: 0
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['grades']

    result = list(collection.aggregate([
      {
        "$project": {
          "student": 1,
          "isAnyTrue": { "$anyElementTrue": ["$scores"] },
          "_id": 0
        }
      }
    ]))

    print(result)

    client.close()

example()
```

$arrayElemAt

The `$arrayElemAt` operator in Amazon DocumentDB allows you to retrieve an element from an array by its index position. This is particularly useful when you need to access a specific element within an array field in your documents.

Parameters

- `array`: The input array from which an element is to be retrieved.
- `index`: The zero-based index position of the element to retrieve. This value must be a non-negative integer.

Example (MongoDB Shell)

In this example, we'll demonstrate how to use the `$arrayElemAt` operator to retrieve specific elements from the `flight_miles` array in the `miles` collection.

Create sample documents

```
db.miles.insertMany([
  { "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "flight_miles" : [ 1205, 2560, 880 ]},
  { "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 2560, 890, 2780 ]},
  { "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 880 ]}
]);
```

Query example

```
db.miles.aggregate([
  { $project: {
    "_id": 1,
    "first_mile": { $arrayElemAt: [ "$flight_miles", 0 ] },
    "last_mile": { $arrayElemAt: [ "$flight_miles", -1 ] }
  }}
]);
```

Output

```
{ "_id" : 1, "first_mile" : 1205, "last_mile" : 880 }
{ "_id" : 2, "first_mile" : 1205, "last_mile" : 2780 }
{ "_id" : 3, "first_mile" : 1205, "last_mile" : 880 }
```

In this example, we use the `$arrayElemAt` operator to retrieve the first and last elements of the `flight_miles` array for each document.

Code examples

To view a code example for using the `$arrayElemAt` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('miles');

  const result = await collection.aggregate([
    { $project: {
      "_id": 1,
      "first_mile": { $arrayElemAt: [ "$flight_miles", 0 ] },
      "last_mile": { $arrayElemAt: [ "$flight_miles", -1 ] }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.mydatabase
    collection = db.miles

    result = list(collection.aggregate([
        { "$project": {
            "_id": 1,
            "first_mile": { "$arrayElemAt": [ "$flight_miles", 0 ] },
            "last_mile": { "$arrayElemAt": [ "$flight_miles", -1 ] }
        }}
    ]))
```

```
    print(result)
    client.close()

example()
```

$arrayToObject

The `$arrayToObject` operator in Amazon DocumentDB is the reverse of the `$objectToArray` operator. It takes an array of key-value pair documents and converts it into a single document. This is particularly useful when you need to convert an array of key-value pairs back into an object or document structure.

Parameters

- `array expression`: An expression that resolves to an array. The array elements must be documents with two fields: k (the key) and v (the value).

Example (MongoDB Shell)

The following example demonstrates how to use `$arrayToObject` to convert an array of key-value pairs back into a document.

Create sample documents

```
db.videos.insertMany([
  { "_id": 1, "name": "Live Soft", "inventory": { "Des Moines": 1000, "Ames": 500 } },
  { "_id": 2, "name": "Top Pilot", "inventory": { "Mason City": 250, "Des Moines":
 1000 } },
  { "_id": 3, "name": "Romancing the Rock", "inventory": { "Mason City": 250, "Ames":
 500 } },
  { "_id": 4, "name": "Bravemind", "inventory": { "Mason City": 250, "Des Moines":
 1000, "Ames": 500 } }
]);
```

Query example

```
db.videos.aggregate([
  { $project: { name: 1, videos: { $objectToArray: "$inventory" } } },
  { $project: { name: 1, inventory: { $arrayToObject: "$videos" } } }
```

```
]);
```

Output

```
{ "_id" : 1, "name" : "Live Soft", "inventory" : { "Des Moines" : 1000, "Ames" :
 500 } }
{ "_id" : 2, "name" : "Top Pilot", "inventory" : { "Mason City" : 250, "Des Moines" :
 1000 } }
{ "_id" : 3, "name" : "Romancing the Rock", "inventory" : { "Mason City" : 250,
 "Ames" : 500 } }
{ "_id" : 4, "name" : "Bravemind", "inventory" : { "Mason City" : 250, "Des Moines" :
 1000, "Ames" : 500 } }
```

In this example, the `$objectToArray` operator is used to convert the `inventory` object into an array of key-value pairs. The `$arrayToObject` operator is then used to convert the array back into a document, restoring the original object structure.

Code examples

To view a code example for using the `$arrayToObject` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('videos');

  const result = await collection.aggregate([
    { $project: { name: 1, videos: { $objectToArray: '$inventory' } } },
    { $project: { name: 1, inventory: { $arrayToObject: '$videos' } } }
  ]).toArray();

  console.log(result);
  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['videos']

    result = list(collection.aggregate([
        { '$project': { 'name': 1, 'videos': { '$objectToArray': '$inventory' } } },
        { '$project': { 'name': 1, 'inventory': { '$arrayToObject': '$videos' } } }
    ]))

    print(result)
    client.close()

example()
```

$asin

New from version 8.0.1.

The `$asin` operator in Amazon DocumentDB returns the arcsine (inverse sine) of a value. The input value must be between -1 and 1.

Parameters

- `expression`: An expression that resolves to a number between -1 and 1.

The result is in radians. To obtain degrees, apply `$radiansToDegrees` to the output.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the `$asin` operator to calculate the arcsine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 0.5 },
  { "_id": 3, "value": 1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "angle": { $asin: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 0.5235987755982989 },
  { "_id": 3, "angle": 1.5707963267948966 }
]
```

Code examples

To view a code example for using the `$asin` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "angle": { $asin: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'angle': { '$asin': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$asinh

New from version 8.0.1.

The $asinh operator in Amazon DocumentDB returns the inverse hyperbolic sine of a value. Use it in the aggregation pipeline to perform hyperbolic trigonometric calculations on numeric fields.

Parameters

- expression: An expression that resolves to a number.

The result is in radians. To obtain degrees, apply $radiansToDegrees to the output.

The return type is double by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the $asinh operator to calculate the inverse hyperbolic sine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 1 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $asinh: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "result": 0 },
  { "_id": 2, "result": 0.881373587019543 },
  { "_id": 3, "result": -0.881373587019543 }
]
```

Code examples

To view a code example for using the $asinh operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "result": { $asinh: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$asinh': '$value' }
        }}
```

```
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$atan

New from version 8.0.1.

The `$atan` operator in Amazon DocumentDB returns the inverse tangent (arc tangent) of a value.

The input expression must resolve to a numeric value.

The result is in radians. To obtain degrees, apply `$radiansToDegrees` to the output.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Parameters

- `expression`: An expression that resolves to a number.

Behavior

null and NaN

Example	Results
`{ $atan: NaN }`	`NaN`
`{ $atan: null }`	`null`

When the input is `null` or the referenced field is missing, the result is `null`. An input of `NaN` produces `NaN`.

Example (MongoDB Shell)

The following example shows how to use the $atan operator to calculate the arctangent of a value.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 1 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "angle": { $atan: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 0.7853981633974483 },
  { "_id": 3, "angle": -0.7853981633974483 }
]
```

Code examples

To view a code example for using the $atan operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "angle": { $atan: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'angle': { '$atan': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$atan2

New from version 8.0.1.

The $atan2 operator in Amazon DocumentDB returns the inverse tangent (arc tangent) of y / x, where y and x are the first and second values passed to the expression respectively.

Both expressions must resolve to numeric values. The operator uses the sign of both arguments to determine the correct quadrant.

The result is in radians. To obtain degrees, apply $radiansToDegrees to the output.

The return type is double by default. If the inputs are 128-bit decimals, the output is also a 128-bit decimal.

Parameters

- expression 1 (y): An expression that resolves to a number representing the y-coordinate.
- expression 2 (x): An expression that resolves to a number representing the x-coordinate.

Behavior

null and NaN

If either argument is null (or references a missing field), the result is null. If either argument is NaN, the result is NaN. When one argument is null and the other is NaN, null takes precedence.

Example	Results
{ $atan2: [NaN, <value>] }	NaN
{ $atan2: [<value>, NaN] }	NaN
{ $atan2: [null, <value>] }	null
{ $atan2: [<value>, null] }	null
{ $atan2: [NaN, null] }	null
{ $atan2: [null, NaN] }	null

Example (MongoDB Shell)

The following example shows how to use the $atan2 operator to calculate the arctangent of y/x for a set of points.

Create sample documents

```
db.points.insertMany([
  { "_id": 1, "y": 1, "x": 1 },
  { "_id": 2, "y": 1, "x": 0 },
  { "_id": 3, "y": -1, "x": -1 }
]);
```

Query example

```
db.points.aggregate([
  { $project: {
    "angle": { $atan2: ["$y", "$x"] }
  }}
]);
```

Output

```
[
  { "_id": 1, "angle": 0.7853981633974483 },
  { "_id": 2, "angle": 1.5707963267948966 },
  { "_id": 3, "angle": -2.356194490192345 }
]
```

Code examples

To view a code example for using the $atan2 operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('points');

  const result = await collection.aggregate([
    { $project: {
      "angle": { $atan2: ["$y", "$x"] }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['points']

    result = list(collection.aggregate([
        { '$project': {
            'angle': { '$atan2': ['$y', '$x'] }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$atanh

New from version 8.0.1.

The $atanh operator in Amazon DocumentDB returns the inverse hyperbolic tangent (hyperbolic arc tangent) of a value.

The input expression must resolve to a number in the range -1 to 1 (inclusive). Values outside this range cause an error.

The return type is double by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Parameters

- expression: An expression that resolves to a number between -1 and 1, inclusive.

Behavior

null, NaN, and +/- Infinity

Example	Results
{ $atanh: NaN }	NaN
{ $atanh: null }	null
{ $atanh: 1 }	Infinity
{ $atanh: -1 }	-Infinity
{ $atanh: Infinity } or { $atanh: -Infinity }	Throws an error.

When the input is null or the referenced field is missing, the result is null. An input of NaN produces NaN. The boundary values 1 and -1 produce Infinity and -Infinity respectively. Any value outside the [-1, 1] range (including ±Infinity) causes an error.

Example (MongoDB Shell)

The following example shows how to use the $atanh operator to calculate the inverse hyperbolic tangent of a value.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 0.5 },
  { "_id": 3, "value": -0.5 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $atanh: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "result": 0 },
  { "_id": 2, "result": 0.5493061443340548 },
  { "_id": 3, "result": -0.5493061443340548 }
]
```

Code examples

To view a code example for using the `$atanh` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "result": { $atanh: "$value" }
```

```
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$atanh': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$avg

The `$avg` aggregation operator in Amazon DocumentDB calculates the average value of the specified expression across the documents that are input to the stage. This operator is useful for computing the average of a numeric field or expression across a set of documents.

Parameters

- `expression`: The expression to use to calculate the average. This can be a field path (e.g. `"$field"`) or an expression (e.g. `{ $multiply: ["$field1", "$field2"] }`).

Example (MongoDB Shell)

The following example demonstrates how to use the $avg operator to calculate the average score across a set of student documents.

Create sample documents

```
db.students.insertMany([
  { name: "John", score: 85 },
  { name: "Jane", score: 92 },
  { name: "Bob", score: 78 },
  { name: "Alice", score: 90 }
]);
```

Query example

```
db.students.aggregate([
  { $group: {
    _id: null,
    avgScore: { $avg: "$score" }
  }}
]);
```

Output

```
[
  {
    "_id": null,
    "avgScore": 86.25
  }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The $avg operator can also be used as a window operator in the $setWindowFields stage. In this context, it returns the average of the specified expression for the documents in each window. You specify the operator under the output field, and optionally define the window boundaries with a window document.

Create sample documents

```
db.readings.insertMany([
  { _id: 1, city: "Denver", day: 1, temp: 70 },
  { _id: 2, city: "Denver", day: 2, temp: 80 },
  { _id: 3, city: "Seattle", day: 1, temp: 60 },
  { _id: 4, city: "Seattle", day: 2, temp: 64 },
  { _id: 5, city: "Seattle", day: 3, temp: 68 }
]);
```

Query example

The following example partitions the documents by `city`, sorts each partition by `day`, and returns a running average of `temp` from the start of the partition through the current document.

```
db.readings.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$city",
      sortBy: { day: 1 },
      output: {
        runningAvg: {
          $avg: "$temp",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "city": "Denver", "day": 1, "temp": 70, "runningAvg": 70 },
  { "_id": 2, "city": "Denver", "day": 2, "temp": 80, "runningAvg": 75 },
  { "_id": 3, "city": "Seattle", "day": 1, "temp": 60, "runningAvg": 60 },
  { "_id": 4, "city": "Seattle", "day": 2, "temp": 64, "runningAvg": 62 },
  { "_id": 5, "city": "Seattle", "day": 3, "temp": 68, "runningAvg": 64 }
]
```

Each document is augmented with `runningAvg`, the average of `temp` within its partition up to and including the current document.

Code examples

To view a code example for using the $avg operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in $group) and window operator usage (in $setWindowFields):

Node.js

```
const { MongoClient } = require('mongodb');

async function calculateAvgScore() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  // Accumulator usage: average across grouped documents
  const students = db.collection('students');
  await students.insertMany([
    { name: "John", score: 85 },
    { name: "Jane", score: 92 },
    { name: "Bob", score: 78 },
    { name: "Alice", score: 90 }
  ]);
  const accumulatorResult = await students.aggregate([
    { $group: {
      _id: null,
      avgScore: { $avg: '$score' }
    }}
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Window operator usage: running average within each partition
  const readings = db.collection('readings');
  await readings.insertMany([
    { _id: 1, city: "Denver", day: 1, temp: 70 },
    { _id: 2, city: "Denver", day: 2, temp: 80 },
    { _id: 3, city: "Seattle", day: 1, temp: 60 },
    { _id: 4, city: "Seattle", day: 2, temp: 64 },
    { _id: 5, city: "Seattle", day: 3, temp: 68 }
  ]);
  const windowResult = await readings.aggregate([
    {
```

```
      $setWindowFields: {
        partitionBy: "$city",
        sortBy: { day: 1 },
        output: {
          runningAvg: {
            $avg: "$temp",
            window: { documents: ["unbounded", "current"] }
          }
        }
      }
    }
  ]).toArray();
  console.log('Window result:', windowResult);

  await client.close();
}

calculateAvgScore();
```

Python

```
from pymongo import MongoClient

def calculate_avg_score():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test

    # Accumulator usage: average across grouped documents
    students = db.students
    students.insert_many([
        { 'name': 'John', 'score': 85 },
        { 'name': 'Jane', 'score': 92 },
        { 'name': 'Bob', 'score': 78 },
        { 'name': 'Alice', 'score': 90 }
    ])
    accumulator_result = list(students.aggregate([
        { '$group': {
            '_id': None,
            'avgScore': { '$avg': '$score' }
        }}
    ]))
```

```
    print('Accumulator result:', accumulator_result)

    # Window operator usage: running average within each partition
    readings = db.readings
    readings.insert_many([
        { '_id': 1, 'city': 'Denver', 'day': 1, 'temp': 70 },
        { '_id': 2, 'city': 'Denver', 'day': 2, 'temp': 80 },
        { '_id': 3, 'city': 'Seattle', 'day': 1, 'temp': 60 },
        { '_id': 4, 'city': 'Seattle', 'day': 2, 'temp': 64 },
        { '_id': 5, 'city': 'Seattle', 'day': 3, 'temp': 68 }
    ])
    window_result = list(readings.aggregate([
        {
            '$setWindowFields': {
                'partitionBy': '$city',
                'sortBy': { 'day': 1 },
                'output': {
                    'runningAvg': {
                        '$avg': '$temp',
                        'window': { 'documents': ['unbounded', 'current'] }
                    }
                }
            }
        }
    ]))
    print('Window result:', window_result)

    client.close()

calculate_avg_score()
```

$binarySize

New from version 8.0.1.

The `$binarySize` operator in Amazon DocumentDB returns the size in bytes of a given string or binary data value. For string values, this is the UTF-8 encoded byte count, not the character count. Multi-byte UTF-8 characters (such as accented characters or CJK characters) will produce a byte count larger than the number of characters in the string.

Parameters

- `expression`: An expression that resolves to a string or binary data value.

Example (MongoDB Shell)

The following example shows how to use the `$binarySize` operator to return the byte size of string fields.

Create sample documents

```
db.docs.insertMany([
  {_id: 1, content: "hello"},
  {_id: 2, content: "Amazon DocumentDB"},
  {_id: 3, content: ""}
]);
```

Query example

```
db.docs.aggregate([
  { $project: { size: { $binarySize: "$content" } } }
]);
```

Output

```
[
  {_id: 1, size: 5},
  {_id: 2, size: 17},
  {_id: 3, size: 0}
]
```

Code examples

To view a code example for using the `$binarySize` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('docs');
    const result = await collection.aggregate([
      { $project: { size: { $binarySize: "$content" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['docs']
        result = list(collection.aggregate([
            {'$project': {'size': {'$binarySize': '$content'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$bitAnd

New from version 8.0.1.

The $bitAnd operator in Amazon DocumentDB performs a bitwise AND operation on integer or long values.

Parameters

- expressions: An array of two or more expressions, which can resolve to integers or longs.

Example (MongoDB Shell)

The following example shows how to use the $bitAnd operator to perform bitwise AND on two fields.

Create sample documents

```
db.flags.insertMany([
  {_id: 1, a: 13, b: 10},
  {_id: 2, a: 7, b: 5},
  {_id: 3, a: 15, b: 9}
]);
```

Query example

```
db.flags.aggregate([
  { $project: { result: { $bitAnd: ["$a", "$b"] } } }
]);
```

Output

```
[
  {_id: 1, result: 8},
  {_id: 2, result: 5},
  {_id: 3, result: 9}
]
```

In binary: 13 (1101) AND 10 (1010) = 8 (1000); 7 (0111) AND 5 (0101) = 5 (0101); 15 (1111) AND 9 (1001) = 9 (1001).

Code examples

To view a code example for using the $bitAnd operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('flags');
    const result = await collection.aggregate([
      { $project: { result: { $bitAnd: ["$a", "$b"] } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['flags']
        result = list(collection.aggregate([
            {'$project': {'result': {'$bitAnd': ['$a', '$b']}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$bitNot

New from version 8.0.1.

The `$bitNot` operator in Amazon DocumentDB performs a bitwise NOT operation on an integer or long value, returning the bitwise complement. For a given integer or long n, the result is -(n+1).

Parameters

- `expression`: An expression that resolves to an integer or long. Returns the bitwise complement, which is -(n+1) for a given integer or long n.

Example (MongoDB Shell)

The following example shows how to use the `$bitNot` operator to compute the bitwise complement of integer values.

Create sample documents

```
db.numbers.insertMany([
  {_id: 1, value: 0},
  {_id: 2, value: 5},
  {_id: 3, value: -3}
]);
```

Query example

```
db.numbers.aggregate([
  { $project: { result: { $bitNot: "$value" } } }
]);
```

Output

```
[
  {_id: 1, result: -1},
  {_id: 2, result: -6},
  {_id: 3, result: 2}
]
```

In binary: NOT 0 = -1; NOT 5 = -6; NOT -3 = 2.

Code examples

To view a code example for using the `$bitNot` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('numbers');
    const result = await collection.aggregate([
      { $project: { result: { $bitNot: "$value" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['numbers']
        result = list(collection.aggregate([
            {'$project': {'result': {'$bitNot': '$value'}}}
        ]))
        print(result)
    finally:
```

```
        client.close()

example()
```

$bitOr

New from version 8.0.1.

The `$bitOr` operator in Amazon DocumentDB performs a bitwise OR operation on integer or long values.

Parameters

- `expressions`: An array of two or more expressions, which can resolve to integers or longs.

Example (MongoDB Shell)

The following example shows how to use the `$bitOr` operator to perform bitwise OR on two fields.

Create sample documents

```
db.flags.insertMany([
  {_id: 1, a: 13, b: 10},
  {_id: 2, a: 7, b: 5},
  {_id: 3, a: 15, b: 9}
]);
```

Query example

```
db.flags.aggregate([
  { $project: { result: { $bitOr: ["$a", "$b"] } } }
]);
```

Output

```
[
  {_id: 1, result: 15},
  {_id: 2, result: 7},
```

```
  {_id: 3, result: 15}
]
```

In binary: 13 (1101) OR 10 (1010) = 15 (1111); 7 (0111) OR 5 (0101) = 7 (0111); 15 (1111) OR 9 (1001) = 15 (1111).

Code examples

To view a code example for using the `$bitOr` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('flags');
    const result = await collection.aggregate([
      { $project: { result: { $bitOr: ["$a", "$b"] } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
```

```
        db = client['test']
        collection = db['flags']
        result = list(collection.aggregate([
            {'$project': {'result': {'$bitOr': ['$a', '$b']}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$bitXor

New from version 8.0.1.

The `$bitXor` operator in Amazon DocumentDB performs a bitwise XOR operation on integer or long values.

Parameters

- `expressions`: An array of two or more expressions, which can resolve to integers or longs.

Example (MongoDB Shell)

The following example shows how to use the `$bitXor` operator to perform bitwise XOR on two fields.

Create sample documents

```
db.flags.insertMany([
  {_id: 1, a: 13, b: 10},
  {_id: 2, a: 7, b: 5},
  {_id: 3, a: 15, b: 9}
]);
```

Query example

```
db.flags.aggregate([
  { $project: { result: { $bitXor: ["$a", "$b"] } } }
]);
```

Output

```
[
  {_id: 1, result: 7},
  {_id: 2, result: 2},
  {_id: 3, result: 6}
]
```

In binary: 13 (1101) XOR 10 (1010) = 7 (0111); 7 (0111) XOR 5 (0101) = 2 (0010); 15 (1111) XOR 9 (1001) = 6 (0110).

Code examples

To view a code example for using the `$bitXor` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('flags');
    const result = await collection.aggregate([
      { $project: { result: { $bitXor: ["$a", "$b"] } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['flags']
        result = list(collection.aggregate([
            {'$project': {'result': {'$bitXor': ['$a', '$b']}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$bottom

New from version 8.0.1.

Use the `$bottom` accumulator in the `$group` stage to return the lowest-ranked document per group according to a specified sort order.

Parameters

- `sortBy`: A document specifying the sort order. Use `1` for ascending or `-1` for descending.
- `output`: An expression that specifies the fields to return from the bottom document.

Example (MongoDB Shell)

The following example shows how to use the `$bottom` accumulator to find the bottom sale (lowest quantity) per item in a sales collection.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, price: 5 },
  { item: "abc", quantity: 5, price: 8 },
  { item: "xyz", quantity: 15, price: 3 },
  { item: "xyz", quantity: 7, price: 6 }
```

```
])
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", bottomSale: { $bottom: { sortBy: { quantity: -1 }, output:
 { quantity: "$quantity", price: "$price" } } } } }
])
```

Output

```
[
  { "_id": "xyz", "bottomSale": { "quantity": 7, "price": 6 } },
  { "_id": "abc", "bottomSale": { "quantity": 5, "price": 8 } }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$bottom` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the `output` from the bottom-ranked document (according to the operator's own `sortBy`) among the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$bottom` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.matchScores.insertMany([
  { _id: 1, player: "Alice", round: 1, score: 20 },
  { _id: 2, player: "Alice", round: 2, score: 35 },
  { _id: 3, player: "Alice", round: 3, score: 28 },
  { _id: 4, player: "Bob", round: 1, score: 15 },
  { _id: 5, player: "Bob", round: 2, score: 40 }
```

```
]);
```

Query example

The following example partitions the documents by `player`, sorts each partition by `round`, and returns the lowest-scoring round seen from the start of the partition through the current document.

```
db.matchScores.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$player",
      sortBy: { round: 1 },
      output: {
        worstRoundSoFar: {
          $bottom: { sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "player": "Alice", "round": 1, "score": 20, "worstRoundSoFar": { "round":
 1, "score": 20 } },
  { "_id": 2, "player": "Alice", "round": 2, "score": 35, "worstRoundSoFar": { "round":
 1, "score": 20 } },
  { "_id": 3, "player": "Alice", "round": 3, "score": 28, "worstRoundSoFar": { "round":
 1, "score": 20 } },
  { "_id": 4, "player": "Bob", "round": 1, "score": 15, "worstRoundSoFar": { "round":
 1, "score": 15 } },
  { "_id": 5, "player": "Bob", "round": 2, "score": 40, "worstRoundSoFar": { "round":
 1, "score": 15 } }
]
```

Each document is augmented with `worstRoundSoFar`, the lowest-scoring round within its partition up to and including the current document.

Code examples

To view a code example for using the $bottom operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in $group) and window operator usage (in $setWindowFields):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: bottom document per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { item: "abc", quantity: 10, price: 5 },
      { item: "abc", quantity: 5, price: 8 },
      { item: "xyz", quantity: 15, price: 3 },
      { item: "xyz", quantity: 7, price: 6 }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", bottomSale: { $bottom: { sortBy: { quantity: -1 },
 output: { quantity: "$quantity", price: "$price" } } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: worst round so far within each partition
    const matchScores = db.collection('matchScores');
    await matchScores.insertMany([
      { _id: 1, player: "Alice", round: 1, score: 20 },
      { _id: 2, player: "Alice", round: 2, score: 35 },
      { _id: 3, player: "Alice", round: 3, score: 28 },
      { _id: 4, player: "Bob", round: 1, score: 15 },
      { _id: 5, player: "Bob", round: 2, score: 40 }
```

```
    ]);
    const windowResult = await matchScores.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$player",
          sortBy: { round: 1 },
          output: {
            worstRoundSoFar: {
              $bottom: { sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: bottom document per group
        sales = db['sales']
```

```
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'price': 5 },
            { 'item': 'abc', 'quantity': 5, 'price': 8 },
            { 'item': 'xyz', 'quantity': 15, 'price': 3 },
            { 'item': 'xyz', 'quantity': 7, 'price': 6 }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'bottomSale': { '$bottom':
{ 'sortBy': { 'quantity': -1 }, 'output': { 'quantity': '$quantity', 'price':
'$price' } } } } }
        ]))
        pprint(accumulator_result)

        # Window operator usage: worst round so far within each partition
        match_scores = db['matchScores']
        match_scores.insert_many([
            { '_id': 1, 'player': 'Alice', 'round': 1, 'score': 20 },
            { '_id': 2, 'player': 'Alice', 'round': 2, 'score': 35 },
            { '_id': 3, 'player': 'Alice', 'round': 3, 'score': 28 },
            { '_id': 4, 'player': 'Bob', 'round': 1, 'score': 15 },
            { '_id': 5, 'player': 'Bob', 'round': 2, 'score': 40 }
        ])
        window_result = list(match_scores.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$player',
                    'sortBy': { 'round': 1 },
                    'output': {
                        'worstRoundSoFar': {
                            '$bottom': { 'sortBy': { 'score': -1 }, 'output':
{ 'round': '$round', 'score': '$score' } },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
```

```
            client.close()

example()
```

$bottomN

New from version 8.0.1.

Use the `$bottomN` accumulator in the `$group` stage to return the bottom N elements within a group according to a specified sort order. If a group contains fewer than N elements, `$bottomN` returns all elements in the group.

Parameters

- n: A positive integer, or an expression that resolves to one, specifying how many bottom results to return per group.
- `sortBy`: A document specifying the sort order. Use 1 for ascending or -1 for descending.
- `output`: An expression that specifies the fields to return from each of the bottom N documents.

Example (MongoDB Shell)

The following example shows how to use the `$bottomN` accumulator to find the bottom 2 sales (lowest quantity) per item in a sales collection.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, price: 5 },
  { item: "abc", quantity: 7, price: 8 },
  { item: "abc", quantity: 5, price: 10 },
  { item: "xyz", quantity: 15, price: 3 },
  { item: "xyz", quantity: 9, price: 6 },
  { item: "xyz", quantity: 3, price: 12 }
])
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", bottomTwoSales: { $bottomN: { n: 2, sortBy: { quantity:
 -1 }, output: { quantity: "$quantity", price: "$price" } } } } }
```

```
])
```

Output

```
[
  { "_id": "xyz", "bottomTwoSales": [{ "quantity": 9, "price": 6 }, { "quantity": 3,
 "price": 12 }] },
  { "_id": "abc", "bottomTwoSales": [{ "quantity": 7, "price": 8 }, { "quantity": 5,
 "price": 10 }] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$bottomN` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of the `output` from the bottom N documents (according to the operator's own `sortBy`) among the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$bottomN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.matchScores.insertMany([
  { _id: 1, player: "Alice", round: 1, score: 20 },
  { _id: 2, player: "Alice", round: 2, score: 35 },
  { _id: 3, player: "Alice", round: 3, score: 28 },
  { _id: 4, player: "Bob", round: 1, score: 15 },
  { _id: 5, player: "Bob", round: 2, score: 40 }
]);
```

Query example

The following example partitions the documents by `player`, sorts each partition by `round`, and returns the two lowest-scoring rounds seen from the start of the partition through the current document.

```
db.matchScores.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$player",
      sortBy: { round: 1 },
      output: {
        bottomTwoRoundsSoFar: {
          $bottomN: { n: 2, sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "player": "Alice", "round": 1, "score": 20, "bottomTwoRoundsSoFar":
 [{ "round": 1, "score": 20 }] },
  { "_id": 2, "player": "Alice", "round": 2, "score": 35, "bottomTwoRoundsSoFar":
 [{ "round": 2, "score": 35 }, { "round": 1, "score": 20 }] },
  { "_id": 3, "player": "Alice", "round": 3, "score": 28, "bottomTwoRoundsSoFar":
 [{ "round": 3, "score": 28 }, { "round": 1, "score": 20 }] },
  { "_id": 4, "player": "Bob", "round": 1, "score": 15, "bottomTwoRoundsSoFar":
 [{ "round": 1, "score": 15 }] },
  { "_id": 5, "player": "Bob", "round": 2, "score": 40, "bottomTwoRoundsSoFar":
 [{ "round": 2, "score": 40 }, { "round": 1, "score": 15 }] }
]
```

Each document is augmented with `bottomTwoRoundsSoFar`, the two lowest-scoring rounds within its partition up to and including the current document. If the window contains fewer than two documents, all of them are returned.

Code examples

To view a code example for using the `$bottomN` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: bottom N documents per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { item: "abc", quantity: 10, price: 5 },
      { item: "abc", quantity: 7, price: 8 },
      { item: "abc", quantity: 5, price: 10 },
      { item: "xyz", quantity: 15, price: 3 },
      { item: "xyz", quantity: 9, price: 6 },
      { item: "xyz", quantity: 3, price: 12 }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", bottomTwoSales: { $bottomN: { n: 2, sortBy:
 { quantity: -1 }, output: { quantity: "$quantity", price: "$price" } } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: bottom two rounds so far within each partition
    const matchScores = db.collection('matchScores');
    await matchScores.insertMany([
      { _id: 1, player: "Alice", round: 1, score: 20 },
      { _id: 2, player: "Alice", round: 2, score: 35 },
      { _id: 3, player: "Alice", round: 3, score: 28 },
      { _id: 4, player: "Bob", round: 1, score: 15 },
      { _id: 5, player: "Bob", round: 2, score: 40 }
    ]);
    const windowResult = await matchScores.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$player",
```

```
          sortBy: { round: 1 },
          output: {
            bottomTwoRoundsSoFar: {
              $bottomN: { n: 2, sortBy: { score: -1 }, output: { round: "$round",
 score: "$score" } },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: bottom N documents per group
        sales = db['sales']
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'price': 5 },
            { 'item': 'abc', 'quantity': 7, 'price': 8 },
            { 'item': 'abc', 'quantity': 5, 'price': 10 },
            { 'item': 'xyz', 'quantity': 15, 'price': 3 },
```

```
            { 'item': 'xyz', 'quantity': 9, 'price': 6 },
            { 'item': 'xyz', 'quantity': 3, 'price': 12 }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'bottomTwoSales': { '$bottomN': { 'n':
2, 'sortBy': { 'quantity': -1 }, 'output': { 'quantity': '$quantity', 'price':
'$price' } } } } }
        ]))
        pprint(accumulator_result)

        # Window operator usage: bottom two rounds so far within each partition
        match_scores = db['matchScores']
        match_scores.insert_many([
            { '_id': 1, 'player': 'Alice', 'round': 1, 'score': 20 },
            { '_id': 2, 'player': 'Alice', 'round': 2, 'score': 35 },
            { '_id': 3, 'player': 'Alice', 'round': 3, 'score': 28 },
            { '_id': 4, 'player': 'Bob', 'round': 1, 'score': 15 },
            { '_id': 5, 'player': 'Bob', 'round': 2, 'score': 40 }
        ])
        window_result = list(match_scores.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$player',
                    'sortBy': { 'round': 1 },
                    'output': {
                        'bottomTwoRoundsSoFar': {
                            '$bottomN': { 'n': 2, 'sortBy': { 'score': -1 },
'output': { 'round': '$round', 'score': '$score' } },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()
```

```
example()
```

$bsonSize

New from version 8.0.1.

The `$bsonSize` operator in Amazon DocumentDB returns the size in bytes of a document when encoded as BSON.

Parameters

- `expression`: An expression that resolves to a document or object.

Example (MongoDB Shell)

The following example shows how to use the `$bsonSize` operator to return the BSON size of each document.

Create sample documents

```
db.items.insertMany([
  {_id: 1, name: "a", tags: ["x"]},
  {_id: 2, name: "ab", tags: ["x", "y", "z"]},
  {_id: 3, name: "abc", tags: []}
]);
```

Query example

```
db.items.aggregate([
  { $project: { docSize: { $bsonSize: "$$ROOT" } } }
]);
```

Output

```
[
  {_id: 1, docSize: 46},
  {_id: 2, docSize: 65},
  {_id: 3, docSize: 39}
]
```

> **Note**
>
> Exact sizes depend on BSON encoding of the document fields.

Code examples

To view a code example for using the `$bsonSize` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('items');
    const result = await collection.aggregate([
      { $project: { docSize: { $bsonSize: "$$ROOT" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
```

```
        collection = db['items']
        result = list(collection.aggregate([
            {'$project': {'docSize': {'$bsonSize': '$$ROOT'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$bucket

New from version 8.0

Not supported by Elastic cluster.

The `$bucket` aggregation stage in Amazon DocumentDB allows you to group input documents into buckets based on a specified expression and bucket boundaries. This can be useful for analyzing data that falls within certain value ranges or categories.

Parameters

- `groupBy` (required): The expression that specifies the value to group by.
- `boundaries` (required): An array of double values that define the bucket boundaries. Documents are assigned to buckets based on the `groupBy` expression value falling within the specified boundaries.
- `default` (optional): A literal value that is output for documents whose `groupBy` expression value does not fall into any of the specified boundaries.
- `output` (optional): An object that specifies the information to output for each bucket. You can use accumulator operators like `$sum`, `$avg`, `$min`, and `$max` to compute aggregations for each bucket.

Example (MongoDB Shell)

The following example demonstrates how to use the `$bucket` stage to group sales data by price range.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", price: 10, quantity: 2, date: new Date("2020-09-01") },
  { item: "def", price: 20, quantity: 1, date: new Date("2020-10-01") },
  { item: "ghi", price: 5, quantity: 3, date: new Date("2020-11-01") },
  { item: "jkl", price: 15, quantity: 2, date: new Date("2020-12-01") },
  { item: "mno", price: 25, quantity: 1, date: new Date("2021-01-01") }
]);
```

Query example

```
db.sales.aggregate([
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [0, 10, 20, 30],
      default: "Other",
      output: {
        "count": { $sum: 1 },
        "totalQuantity": { $sum: "$quantity" }
      }
    }
  },
  {
    $sort: { _id: 1 }
  }
])
```

Output

```
[
  { _id: 0, count: 1, totalQuantity: 3 },
  { _id: 10, count: 2, totalQuantity: 4 },
  { _id: 20, count: 2, totalQuantity: 2 }
]
```

Code examples

To view a code example for using the `$bucket` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const sales = db.collection('sales');

  const result = await sales.aggregate([
    {
      $bucket: {
        groupBy: "$price",
        boundaries: [0, 10, 20, 30],
        default: "Other",
        output: {
          "count": { $sum: 1 },
          "totalQuantity": { $sum: "$quantity" }
        }
      }
    },
    {
      $sort: { _id: 1 }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&lsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
```

```
    sales = db['sales']

    result = list(sales.aggregate([
        {
            '$bucket': {
                'groupBy': '$price',
                'boundaries': [0, 10, 20, 30],
                'default': 'Other',
                'output': {
                    'count': {'$sum': 1},
                    'totalQuantity': {'$sum': '$quantity'}
                }
            }
        },
        {
            "$sort": { "_id": 1 }
        }
    ]))

    print(result)
    client.close()

example()
```

$bucketAuto

New from version 8.0.2.

The $bucketAuto aggregation stage in Amazon DocumentDB automatically groups input documents into a specified number of buckets based on a specified expression. Unlike $bucket, where you define the bucket boundaries explicitly, $bucketAuto automatically determines the boundaries so that the documents are distributed as evenly as possible across the requested number of buckets. This can be useful when you want to categorize data into a fixed number of groups without knowing the distribution of the values in advance.

Parameters

- groupBy (required): The expression that specifies the value to group by.
- buckets (required): A positive integer that specifies the number of buckets into which the documents are grouped. Depending on the input documents and the groupBy expression, the actual number of buckets can be less than the specified number.

- `output` (optional): An object that specifies the information to output for each bucket. You can use accumulator operators like `$sum`, `$avg`, `$min`, and `$max` to compute aggregations for each bucket. If omitted, each bucket outputs a `count` field by default.
- `granularity` (optional): A string that specifies the preferred number series to use for the bucket boundaries. This snaps the computed boundaries to a standard numerical progression. This parameter is available only if all `groupBy` values are numeric and none of them are `NaN`. Valid values are: `R5`, `R10`, `R20`, `R40`, `R80`, `1-2-5`, `E6`, `E12`, `E24`, `E48`, `E96`, `E192`, and `POWERSOF2`.

Output

Each bucket is returned as a separate document that contains an `_id` object describing the bounds of the bucket:

- `_id.min`: The inclusive lower bound of the bucket.
- `_id.max`: The upper bound of the bucket. This bound is exclusive for every bucket except the final bucket in the series, where it is inclusive.

Example (MongoDB Shell)

The following example demonstrates how to use the `$bucketAuto` stage to group sales data into buckets by price.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", price: 10, quantity: 2, date: new Date("2020-09-01") },
  { item: "def", price: 20, quantity: 1, date: new Date("2020-10-01") },
  { item: "ghi", price: 5, quantity: 3, date: new Date("2020-11-01") },
  { item: "jkl", price: 15, quantity: 2, date: new Date("2020-12-01") },
  { item: "mno", price: 25, quantity: 1, date: new Date("2021-01-01") }
]);
```

Query example

The following command groups the sample documents into three buckets by price:

```
db.sales.aggregate([
  {
    $bucketAuto: {
      groupBy: "$price",
```

```
      buckets: 3,
      output: {
        "count": { $sum: 1 },
        "totalQuantity": { $sum: "$quantity" }
      }
    }
  }
])
```

Output

The command returns the following output:

```
[
  { _id: { min: 5, max: 15 }, count: 2, totalQuantity: 5 },
  { _id: { min: 15, max: 25 }, count: 2, totalQuantity: 3 },
  { _id: { min: 25, max: 25 }, count: 1, totalQuantity: 1 }
]
```

Query example with granularity

The following example adds the `granularity` option to snap the computed bucket boundaries to powers of 2.

```
db.sales.aggregate([
  {
    $bucketAuto: {
      groupBy: "$price",
      buckets: 2,
      granularity: "POWERSOF2",
      output: {
        "count": { $sum: 1 },
        "totalQuantity": { $sum: "$quantity" }
      }
    }
  }
])
```

Output

The command returns the following output:

```
[
```

```
  { _id: { min: 4, max: 16 }, count: 3, totalQuantity: 7 },
  { _id: { min: 16, max: 32 }, count: 2, totalQuantity: 2 }
]
```

Code examples

To view a code example for using the $bucketAuto command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const sales = db.collection('sales');

  const result = await sales.aggregate([
    {
      $bucketAuto: {
        groupBy: "$price",
        buckets: 3,
        output: {
          "count": { $sum: 1 },
          "totalQuantity": { $sum: "$quantity" }
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    sales = db['sales']

    result = list(sales.aggregate([
        {
            '$bucketAuto': {
                'groupBy': '$price',
                'buckets': 3,
                'output': {
                    'count': {'$sum': 1},
                    'totalQuantity': {'$sum': '$quantity'}
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$ceil

New from version 4.0

The `$ceil` operator in Amazon DocumentDB, as in MongoDB, rounds a number up to the nearest integer. This is useful when you need to perform mathematical operations on numeric fields and ensure the result is a whole number.

Parameters

- `expression`: The numeric expression to round up.

Example (MongoDB Shell)

This example demonstrates how to use the `$ceil` operator to round up a numeric field.

Create sample documents

```
db.numbers.insertMany([
  { "_id": 1, "value": 3.14 },
  { "_id": 2, "value": -2.7 },
  { "_id": 3, "value": 0 }
])
```

Query example

```
db.numbers.aggregate([
  { $project: {
    "roundedUp": { $ceil: "$value" }
  }}
])
```

Output

```
{ "_id": 1, "roundedUp": 4 }
{ "_id": 2, "roundedUp": -2 }
{ "_id": 3, "roundedUp": 0 }
```

Code examples

To view a code example for using the `$ceil` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      "roundedUp": { $ceil: "$value" }
    }}
  ]).toArray();
```

```
  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.numbers

    result = list(collection.aggregate([
        { '$project': {
            "roundedUp": { '$ceil': "$value" }
        }}
    ]))

    print(result)
    client.close()

example()
```

$changeStream

Not supported by Elastic cluster.

The `$changeStream` aggregation stage opens a change stream cursor to monitor real-time changes to a collection. It returns change event documents when insert, update, replace, or delete operations occur.

Parameters

- `fullDocument`: Specifies whether to return the full document for update operations. Options are `default` or `updateLookup`.

- `resumeAfter`: Optional. Resume token to continue from a specific point in the change stream.
- `startAtOperationTime`: Optional. Timestamp to start the change stream from.
- `allChangesForCluster`: Optional. Boolean value. When `true`, watches all changes across the cluster (for admin database). When `false` (default), watches only the specified collection.
- `showExpandedEvents` – Optional. Specifies whether expanded data definition language (DDL) event notifications, including `createCollection`, `createIndexes`, and `dropIndexes` events, are included in the change stream. The default is `false`. The `createCollection` and `createIndexes` events are available in Amazon DocumentDB 8.0 starting with engine version 8.0.2. The `dropIndexes` event is available starting with engine version 8.0.3.

Example (MongoDB Shell)

The following example demonstrates using the `$changeStream` stage to monitor changes to a collection.

Query example

```
// Open change stream first
const changeStream = db.inventory.aggregate([
  { $changeStream: { fullDocument: "updateLookup" } }
]);

// In another session, insert a document
db.inventory.insertOne({ _id: 1, item: "Widget", qty: 10 });

// Back in the first session, read the change event
if (changeStream.hasNext()) {
  print(tojson(changeStream.next()));
}
```

Output

```
{
  _id: { _data: '...' },
  operationType: 'insert',
  clusterTime: Timestamp(1, 1234567890),
  fullDocument: { _id: 1, item: 'Widget', qty: 10 },
  ns: { db: 'test', coll: 'inventory' },
  documentKey: { _id: 1 }
```

```
}
```

Code examples

To view a code example for using the `$changeStream` aggregation stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  // Open change stream
  const changeStream = collection.watch([]);

  changeStream.on('change', (change) => {
    console.log('Change detected:', change);
  });

  // Simulate insert in another operation
  setTimeout(async () => {
    await collection.insertOne({ _id: 1, item: 'Widget', qty: 10 });
  }, 1000);

  // Keep connection open to receive changes
  // In production, handle cleanup appropriately
}

example();
```

Python

```
from pymongo import MongoClient
import threading
import time

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    # Open change stream
    change_stream = collection.watch([])

    # Insert document in separate thread after delay
    def insert_doc():
        time.sleep(1)
        collection.insert_one({'_id': 1, 'item': 'Widget', 'qty': 10})

    threading.Thread(target=insert_doc).start()

    # Watch for changes
    for change in change_stream:
        print('Change detected:', change)
        break  # Exit after first change

    client.close()

example()
```

$changeStreamSplitLargeEvent

This aggregation stage is not supported by Elastic clusters.

This stage is available starting with Amazon DocumentDB engine version 8.0.2.

The `$changeStreamSplitLargeEvent` aggregation stage is used within a $changeStream pipeline to split events exceeding 16 MB into smaller fragments. It returns each fragment sequentially via the change stream cursor.

Parameters

None. The input to the `$changeStreamSplitLargeEvent` stage should be an empty document.

Example – MongoDB Shell

The following example demonstrates using the `$changeStreamSplitLargeEvent` stage to split an oversized change event into multiple fragments.

Query example

```
// Insert a document
db.inventory.insertOne({ _id: 1, item: "Widget", payload: "x".repeat(9 * 1024 *
 1024) })

// Open change stream with the $changeStreamSplitLargeEvent stage
var changeStream = db.inventory.aggregate([
  { $changeStream: { fullDocument: "updateLookup" } },
  { $changeStreamSplitLargeEvent: {} }
]);

// Update the document
db.inventory.updateOne({ _id: 1 }, { $set: { payload: "y".repeat(9 * 1024 * 1024) } })

// Read the change event
if (changeStream.hasNext()) {
  print(tojson(changeStream.next()));
}
```

Output

```
{
  _id: { _data: '...' },
  splitEvent: { fragment: 1, of: 2},
  ns: { db: 'test', coll: 'inventory' },
  clusterTime: Timestamp(4, 1789154779),
  documentKey: { _id: 1 },
  fullDocument: { _id: 1, item: 'Widget', payload: yyyyyyyy...(9437184 chars) },
  operationType: 'update',
}
{
  _id: { _data: '...' },
  splitEvent: { fragment: 2, of: 2},
  updateDescription: { updatedFields: { payload: yyyyyyyy...(9437184 chars) },
 removedFields: [] }
}
```

Code examples

To view a code example for using the `$changeStreamSplitLargeEvent` aggregation stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  // Insert a large document first
  await collection.insertOne({ _id: 1, item: 'Widget', payload: 'x'.repeat(9 * 1024
 * 1024) });

  // Open change stream with $changeStreamSplitLargeEvent
  const changeStream = collection.watch(
    [{ $changeStreamSplitLargeEvent: {} }],
    { fullDocument: 'updateLookup' }
  );

  changeStream.on('change', (change) => {
    console.log('Change detected:', change);
  });

  // Update to trigger a split event — fullDocument + updateDescription > 16 MB
  setTimeout(async () => {
    console.log('Triggering update...');
    await collection.updateOne({ _id: 1 }, { $set: { payload: 'y'.repeat(9 * 1024 *
 1024) } });
  }, 1000);

  // Keep connection open to receive changes
  // In production, handle cleanup appropriately
}

example();
```

Python

```
from pymongo import MongoClient
import threading
import time
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    # Insert large document first
    collection.drop()
    collection.insert_one({'_id': 1, 'item': 'Widget', 'payload': 'x' * (9 * 1024 *
 1024)})

    # Open change stream with $changeStreamSplitLargeEvent
    change_stream = collection.watch(
        [{'$changeStreamSplitLargeEvent': {}}],
        full_document='updateLookup'
    )

    # Update in separate thread after delay
    def update_doc():
        collection.update_one({'_id': 1}, {'$set': {'payload': 'y' * (9 * 1024 *
 1024)}})

    threading.Thread(target=update_doc).start()

    # Watch for changes
    for change in change_stream:
        print('Change detected:', change)

    client.close()

example()
```

$cmp

The `$cmp` operator in Amazon DocumentDB is used to compare two values and return an integer value that indicates their relative order. It is a comparison operator that compares two expressions and returns an integer value of -1, 0, or 1, depending on whether the first value is less than, equal to, or greater than the second value, respectively.

Parameters

- `expression1`: The first expression to compare.
- `expression2`: The second expression to compare.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$cmp` operator to compare two numeric values.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, value1: 10, value2: 20 },
  { _id: 2, value1: 15, value2: 15 },
  { _id: 3, value1: 20, value2: 10 }
]);
```

Query example

```
db.collection.find({
  $expr: {
    $cmp: ["$value1", "$value2"]
  }
})
```

Output

```
[
  { "_id" : 1, "value1" : 10, "value2" : 20 },
  { "_id" : 3, "value1" : 20, "value2" : 10 }
]
```

In this example, the `$cmp` operator compares the `value1` and `value2` fields for each document. The result is:

```
- `$cmp: ["$value1", "$value2"]` returns -1 for the first document (10 < 20), 0 for the
 second document (15 = 15), and 1 for the third document (20 > 10).
```

Code examples

To view a code example for using the $cmp command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the $cmp operator in a Node.js application with the mongodb driver:

```
const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

async function main() {
  await client.connect();
  const db = client.db('test');
  const collection = db.collection('mycollection');

  // Insert sample documents
  await collection.insertMany([
    { _id: 1, value1: 10, value2: 20 },
    { _id: 2, value1: 15, value2: 15 },
    { _id: 3, value1: 20, value2: 10 }
  ]);

  // Query using $cmp operator
  const result = await collection.find({
    $expr: {
      $cmp: ['$value1', '$value2']
    }
  }).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

Here's an example of using the $cmp operator in a Python application with the pymongo driver:

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['mycollection']

# Insert sample documents
collection.insert_many([
    {'_id': 1, 'value1': 10, 'value2': 20},
    {'_id': 2, 'value1': 15, 'value2': 15},
    {'_id': 3, 'value1': 20, 'value2': 10}
])

# Query using $cmp operator
result = list(collection.find({
    '$expr': {
        '$cmp': ['$value1', '$value2']
    }
}))

print(result)

client.close()
```

The output of both the Node.js and Python examples will be the same as the MongoDB Shell example:

```
[
  { "_id" : 1, "value1" : 10, "value2" : 20 },
  { "_id" : 2, "value1" : 15, "value2" : 15 },
  { "_id" : 3, "value1" : 20, "value2" : 10 }
]
```

$collStats

New from version 4.0

The `$collStats` aggregation stage in Amazon DocumentDB provides statistics about the specified collection, similar to the `db.collection.stats()` command in the MongoDB shell.

This stage can be used to retrieve information about the collection, such as the number of documents, the total size of the collection, and various performance metrics.

Parameters

- `latencyStats`: (optional) A document that specifies the options for collecting latency statistics. This parameter is not supported in Amazon DocumentDB.
- `recordStats`: (optional) A document that specifies the options for collecting record statistics. This parameter is not supported in Amazon DocumentDB.
- `queryExecStats`: (optional) A document that specifies the options for collecting query execution statistics. This parameter is not supported in Amazon DocumentDB.

Example (MongoDB Shell)

The following example demonstrates how to use the `$collStats` aggregation stage to retrieve statistics about a collection named `test` in the db database.

Create sample documents

```
db.test.insertMany([
  { "name": "John", "age": 30 },
  { "name": "Jane", "age": 25 },
  { "name": "Bob", "age": 35 }
]);
```

Query example

```
db.test.aggregate([
  { $collStats: {} }
]);
```

Output

```
{
  "ns" : "db.test",
  "count" : 3,
  "size" : 87,
  "avgObjSize" : 29,
  "storageSize" : 40960,
  "capped" : false,
```

```
  "nindexes" : 1,
  "totalIndexSize" : 8192,
  "indexSizes" : {
    "_id_" : 8192
  },
  "collScans" : 0,
  "idxScans" : 0,
  "opCounter" : {
    "numDocsIns" : 3,
    "numDocsUpd" : 0,
    "numDocsDel" : 0
  },
  "cacheStats" : {
    "collBlksHit" : 0,
    "collBlksRead" : 0,
    "collHitRatio" : 0,
    "idxBlksHit" : 0,
    "idxBlksRead" : 0,
    "idxHitRatio" : 0
  },
  "lastReset" : "2023-04-11T12:00:00Z",
  "ok" : 1,
  "operationTime" : Timestamp(1681206000, 1)
}
```

Code examples

To view a code example for using the `$collStats` command, choose the tab for the language that you want to use:

Node.js

Here's an example of how to use the `$collStats` aggregation stage in a Node.js application using the official MongoDB Node.js driver:

```
const { MongoClient } = require('mongodb');

async function runCollStatsExample() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('db');
  const collection = db.collection('test');
```

```
  const result = await collection.aggregate([
    { $collStats: {} }
  ]).toArray();

  console.log(result);

  await client.close();
}

runCollStatsExample();
```

Python

Here's an example of how to use the `$collStats` aggregation stage in a Python application using the PyMongo driver:

```
from pymongo import MongoClient

def run_coll_stats_example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['db']
    collection = db['test']

    result = list(collection.aggregate([
        { '$collStats': {} }
    ]))

    print(result)

    client.close()

run_coll_stats_example()
```

$concat

The `$concat` aggregation operator in Amazon DocumentDB concatenates (or combines) multiple strings in a document to produce a single string that can be returned to the application. This

reduces the work done in the application, as the string manipulation is performed at the database level.

Parameters

- `expression1`: The first string to concatenate.
- `expression2`: The second string to concatenate.
- `...`: Additional strings to concatenate (optional).

Example (MongoDB Shell)

In this example, we concatenate the first and last names of users to produce each person's full name.

Create sample documents

```
db.people.insertMany([
  { "_id":1, "first_name":"Jane", "last_name":"Doe", "DOB":"2/1/1999", "Desk": "MSP102-
MN"},
  { "_id":2, "first_name":"John", "last_name":"Doe", "DOB":"12/21/1992", "Desk":
 "DSM301-IA"},
  { "_id":3, "first_name":"Steve", "last_name":"Smith", "DOB":"3/21/1981",
 "Desk":"MKE233-WI"}
])
```

Query example

```
db.people.aggregate([
  { $project: { full_name: { $concat: [ "$first_name", " ", "$last_name"] } } }
])
```

Output

```
{ "_id" : 1, "full_name" : "Jane Doe" }
{ "_id" : 2, "full_name" : "John Doe" }
{ "_id" : 3, "full_name" : "Steve Smith" }
```

Code examples

To view a code example for using the `$concat` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function concatenateNames() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  const result = await db.collection('people').aggregate([
    { $project: { full_name: { $concat: [ "$first_name", " ", "$last_name"] } } }
  ]).toArray();

  console.log(result);

  await client.close();
}

concatenateNames();
```

Python

```
from pymongo import MongoClient

def concatenate_names():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']

    result = list(db.people.aggregate([
        { '$project': { 'full_name': { '$concat': [ '$first_name', ' ',
 '$last_name' ] } } }
    ]))

    print(result)

    client.close()

concatenate_names()
```

$concatArrays

The `$concatArrays` aggregation operator in Amazon DocumentDB is used to concatenate two or more arrays into a single array. This can be useful when you need to combine multiple arrays of data into a single array for further processing or analysis.

Parameters

- `array1`: The first array to be concatenated.
- `array2`: The second array to be concatenated.
- `[array3, ...]`: (optional) Additional arrays to be concatenated.

Example (MongoDB Shell)

The following example demonstrates how to use the `$concatArrays` operator to combine two arrays into a single array.

Create sample documents

```
db.collection.insertMany([
  {
    "_id": 1,
    "name": "John Doe",
    "hobbies": ["reading", "swimming"],
    "skills": ["programming", "design"]
  },
  {
    "_id": 2,
    "name": "Jane Smith",
    "hobbies": ["hiking", "cooking"],
    "skills": ["marketing", "analysis"]
  }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      _id: 0,
```

```
      name: 1,
      all_activities: { $concatArrays: ["$hobbies", "$skills"] }
    }
  }
]);
```

Output

```
[
  {
    "name": "John Doe",
    "all_activities": [
      "reading",
      "swimming",
      "programming",
      "design"
    ]
  },
  {
    "name": "Jane Smith",
    "all_activities": [
      "hiking",
      "cooking",
      "marketing",
      "analysis"
    ]
  }
]
```

Code examples

To view a code example for using the `$concatArrays` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('mydb');
```

```
  const collection = db.collection('mycollection');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 0,
        name: 1,
        all_activities: { $concatArrays: ['$hobbies', '$skills'] }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['mydb']
    collection = db['mycollection']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 0,
                'name': 1,
                'all_activities': { '$concatArrays': ['$hobbies', '$skills'] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$cond

The $cond operator in Amazon DocumentDB is used to evaluate a conditional expression and return one of two possible result expressions.

Parameters

- `if`: The boolean expression to evaluate.
- `then`: The expression to return if the `if` expression is true.
- `else`: The expression to return if the `if` expression is false.

Example (MongoDB Shell)

The following example demonstrates the use of the $cond operator to return a value based on the age of a person.

Create sample documents

```
db.people.insertMany([
  { _id: 1, name: "John Doe", age: 35 },
  { _id: 2, name: "Jane Doe", age: 25 },
  { _id: 3, name: "Bob Smith", age: 65 }
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      name: 1,
      ageGroup: {
        $cond: {
          if: { $lt: ["$age", 30] },
          then: "young",
          else: {
            $cond: {
              if: { $lt: ["$age", 65] },
              then: "middle-aged",
              else: "elderly"
            }
          }
```

```
        }
      }
    }
  }
])
```

Output

```
[
  { "_id" : 1, "name" : "John Doe", "ageGroup" : "middle-aged" },
  { "_id" : 2, "name" : "Jane Doe", "ageGroup" : "young" },
  { "_id" : 3, "name" : "Bob Smith", "ageGroup" : "elderly" }
]
```

Code examples

To view a code example for using the $cond command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    { $project: {
        name: 1,
        ageGroup: {
            $cond: {
                if: { $lt: ["$age", 30] },
                then: "young",
                else: {
                    $cond: {
                        if: { $lt: ["$age", 65] },
                        then: "middle-aged",
                        else: "elderly"
                    }
```

```
                }
            }
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient


def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.people

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'ageGroup': {
                    '$cond': {
                        'if': { '$lt': ["$age", 30]},
                        'then': "young",
                        'else': {
                            '$cond': {
                                'if': { '$lt': ["$age", 65]},
                                'then': "middle-aged",
                                'else': "elderly"
                            }
                        }
                    }
                }
            }
```

```
        }
    ]))

    print(result)
    client.close()

example()
```

$convert

New from version 4.0

The `$convert` operator in Amazon DocumentDB is used to convert a value from one data type to another. This operator is useful when you need to perform operations on data of different types, such as converting a string to a number or a date to a timestamp.

Parameters

- `to`: The target data type to convert the value to. Supported values are `"string"`, `"double"`, `"long"`, `"int"`, `"date"`, and `"boolean"`.
- `from`: The current data type of the value. If not specified, Amazon DocumentDB will attempt to automatically detect the data type.
- on`Error`: (optional) The value to return if the conversion fails. Can be a specific value or one of the following special values: `"null"`, `"zerofill"`, or `"error"`.
- on`Null`: (optional) The value to return if the input value is `null`. Can be a specific value or one of the following special values: `"null"`, `"zerofill"`, or `"error"`.

Example (MongoDB Shell)

The following example demonstrates converting a string value to a date using the `$convert` operator.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "John Doe", joinedOn: "2022-01-01" },
  { _id: 2, name: "Jane Smith", joinedOn: "2023-02-15" },
  { _id: 3, name: "Bob Johnson", joinedOn: "invalid date" }
]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      _id: 1,
      name: 1,
      joinedOn: {
        $convert: {
          input: "$joinedOn",
          to: "date",
          onError: "null",
          onNull: "null"
        }
      }
    }
  }
])
```

Output

```
[
  { "_id" : 1, "name" : "John Doe", "joinedOn" : ISODate("2022-01-01T00:00:00Z") },
  { "_id" : 2, "name" : "Jane Smith", "joinedOn" : ISODate("2023-02-15T00:00:00Z") },
  { "_id" : 3, "name" : "Bob Johnson", "joinedOn" : null }
]
```

Code examples

To view a code example for using the `$convert` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require("mongodb");

async function example() {
  const client = await MongoClient.connect("mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false");
  const db = client.db("test");
  const users = db.collection("users");
```

```
  const results = await users.aggregate([
    {
      $project: {
        _id: 1,
        name: 1,
        joinedOn: {
          $convert: {
            input: "$joinedOn",
            to: "date",
            onError: "null",
            onNull: "null"
          }
        }
      }
    }
  ]).toArray();

  console.log(results);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient("mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false")
    db = client["test"]
    users = db["users"]

    results = list(users.aggregate([
        {
            "$project": {
                "_id": 1,
                "name": 1,
                "joinedOn": {
                    "$convert": {
                        "input": "$joinedOn",
```

```
                        "to": "date",
                        "onError": "null",
                        "onNull": "null"
                    }
                }
            }
        }
    ]))

    print(results)
    client.close()

example()
```

$cos

New from version 8.0.1.

The $cos operator in Amazon DocumentDB returns the cosine of a value that is measured in radians. Use it in the aggregation pipeline to perform trigonometric calculations on numeric fields.

Parameters

- expression: An expression that resolves to a number in radians.

Example (MongoDB Shell)

The following example shows how to use the $cos operator to calculate the cosine of angle values in radians.

Create sample documents

```
db.angles.insertMany([
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 1.0471975511965976 },
  { "_id": 3, "angle": 3.141592653589793 }
]);
```

Query example

```
db.angles.aggregate([
```

```
  { $project: {
    "cosine": { $cos: "$angle" }
  }}
]);
```

Output

```
[
  { "_id": 1, "cosine": 1 },
  { "_id": 2, "cosine": 0.5000000000000001 },
  { "_id": 3, "cosine": -1 }
]
```

Code examples

To view a code example for using the $cos operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('angles');

  const result = await collection.aggregate([
    { $project: {
      "cosine": { $cos: "$angle" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['angles']

    result = list(collection.aggregate([
        { '$project': {
            'cosine': { '$cos': '$angle' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$cosh

New from version 8.0.1.

The `$cosh` operator in Amazon DocumentDB returns the hyperbolic cosine of a value. Use it in the aggregation pipeline to perform hyperbolic trigonometric calculations on numeric fields.

Parameters

- `expression`: An expression that resolves to a number.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the `$cosh` operator to calculate the hyperbolic cosine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 1 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $cosh: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "result": 1 },
  { "_id": 2, "result": 1.5430806348152437 },
  { "_id": 3, "result": 1.5430806348152437 }
]
```

Code examples

To view a code example for using the `$cosh` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
```

```
    { $project: {
      "result": { $cosh: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$cosh': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$count (stage)

The `$count` aggregation stage in Amazon DocumentDB is used to count the number of documents that pass into the stage. It is often used as the final stage in an aggregation pipeline to return the total count of documents matching the previous stages.

Parameters

- `field`: The field to count. This parameter is optional, and if not provided, the stage will count the total number of input documents.

Example (MongoDB Shell)

The following example demonstrates how to use the `$count` stage to get the total number of documents in a collection.

Create sample documents

```
db.users.insertMany([
  { name: "John", age: 30 },
  { name: "Jane", age: 25 },
  { name: "Bob", age: 35 },
  { name: "Alice", age: 28 }
]);
```

Query example

```
db.users.aggregate([
  { $count: "total" }
]);
```

Output

```
{ "total" : 4 }
```

The example aggregates the `users` collection and uses the `$count` stage to count the total number of documents.

Code examples

To view a code example for using the `$count` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function countDocuments() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('mydb');
  const collection = db.collection('users');

  const result = await collection.aggregate([
    { $count: "total" }
  ]).toArray();

  console.log(result[0].total);

  await client.close();
}

countDocuments();
```

Python

```
from pymongo import MongoClient

def count_documents():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['mydb']
    collection = db['users']

    result = list(collection.aggregate([
        { '$count': "total" }
    ]))

    print(result[0]['total'])

    client.close()

count_documents()
```

$count (accumulator)

New from version 8.0.1.

Use the `$count` accumulator within a `$group` stage to return the number of documents in each group. It accepts an empty object `{}` as its argument.

This is different from the `$count` pipeline stage, which is a standalone stage that counts all documents passing through the pipeline. The `$count` accumulator instead counts documents within each group produced by `$group`.

Syntax

```
{ $count: {} }
```

Parameters

- The `$count` accumulator takes no arguments. It accepts an empty object `{}` and returns the count of documents in each group.

Example (MongoDB Shell)

The following example shows how to use the `$count` accumulator to count the number of products in each category.

Create sample documents

```
db.products.insertMany([
  { name: "Widget", category: "A" },
  { name: "Gadget", category: "A" },
  { name: "Doohickey", category: "B" },
  { name: "Thingamajig", category: "B" },
  { name: "Whatsit", category: "B" }
])
```

Query example

```
db.products.aggregate([
  { $group: { _id: "$category", count: { $count: {} } } }
])
```

Output

```
[
```

```
  { "_id": "A", "count": 2 },
  { "_id": "B", "count": 3 }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$count` accumulator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the number of documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

Create sample documents

```
db.events.insertMany([
  { _id: 1, category: "A", day: 1 },
  { _id: 2, category: "A", day: 2 },
  { _id: 3, category: "A", day: 3 },
  { _id: 4, category: "B", day: 1 },
  { _id: 5, category: "B", day: 2 }
])
```

Query example

The following example partitions the documents by `category`, sorts each partition by `day`, and returns a running count of documents from the start of the partition through the current document.

```
db.events.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$category",
      sortBy: { day: 1 },
      output: {
        runningCount: {
          $count: {},
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
])
```

Output

```
[
  { "_id": 1, "category": "A", "day": 1, "runningCount": 1 },
  { "_id": 2, "category": "A", "day": 2, "runningCount": 2 },
  { "_id": 3, "category": "A", "day": 3, "runningCount": 3 },
  { "_id": 4, "category": "B", "day": 1, "runningCount": 1 },
  { "_id": 5, "category": "B", "day": 2, "runningCount": 2 }
]
```

Each document is augmented with `runningCount`, the number of documents within its partition up to and including the current document.

Code examples

To view a code example for using the `$count` accumulator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: count per group
    const products = db.collection('products');
    await products.insertMany([
      { name: "Widget", category: "A" },
      { name: "Gadget", category: "A" },
      { name: "Doohickey", category: "B" },
      { name: "Thingamajig", category: "B" },
      { name: "Whatsit", category: "B" }
    ]);
```

```
    const accumulatorResult = await products.aggregate([
      { $group: { _id: "$category", count: { $count: {} } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: running count within each partition
    const events = db.collection('events');
    await events.insertMany([
      { _id: 1, category: "A", day: 1 },
      { _id: 2, category: "A", day: 2 },
      { _id: 3, category: "A", day: 3 },
      { _id: 4, category: "B", day: 1 },
      { _id: 5, category: "B", day: 2 }
    ]);
    const windowResult = await events.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$category",
          sortBy: { day: 1 },
          output: {
            runningCount: {
              $count: {},
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint
```

```
def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: count per group
        products = db['products']
        products.insert_many([
            { 'name': 'Widget', 'category': 'A' },
            { 'name': 'Gadget', 'category': 'A' },
            { 'name': 'Doohickey', 'category': 'B' },
            { 'name': 'Thingamajig', 'category': 'B' },
            { 'name': 'Whatsit', 'category': 'B' }
        ])
        accumulator_result = list(products.aggregate([
            { '$group': { '_id': '$category', 'count': { '$count': {} } } }
        ]))
        pprint(accumulator_result)

        # Window operator usage: running count within each partition
        events = db['events']
        events.insert_many([
            { '_id': 1, 'category': 'A', 'day': 1 },
            { '_id': 2, 'category': 'A', 'day': 2 },
            { '_id': 3, 'category': 'A', 'day': 3 },
            { '_id': 4, 'category': 'B', 'day': 1 },
            { '_id': 5, 'category': 'B', 'day': 2 }
        ])
        window_result = list(events.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$category',
                    'sortBy': { 'day': 1 },
                    'output': {
                        'runningCount': {
                            '$count': {},
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
```

```
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$covariancePop

New from version 8.0.2.

The `$covariancePop` operator in Amazon DocumentDB returns the population covariance of two numeric expressions. It is a window operator that is used only in the `$setWindowFields` stage; it is not valid in the `$group` or `$bucket` stages. You specify the operator under the stage's `output` field, and optionally define the window boundaries with a `window` document.

Parameters

- `$covariancePop` takes a two-element array `[ <expression1>, <expression2> ]`, where each element is a numeric expression (a field path such as `"$x"`, an expression, or a constant). It returns the population covariance of the two expressions across the documents in the window.

Example (MongoDB Shell)

The following example partitions the documents by `series`, sorts each partition by `t`, and computes the population covariance of `x` and `y` over the whole partition.

Create sample documents

```
db.measurements.insertMany([
  { _id: 1, series: "A", t: 1, x: 1, y: 2 },
  { _id: 2, series: "A", t: 2, x: 2, y: 5 },
  { _id: 3, series: "A", t: 3, x: 3, y: 8 },
```

```
  { _id: 4, series: "B", t: 1, x: 1, y: 8 },
  { _id: 5, series: "B", t: 2, x: 2, y: 5 },
  { _id: 6, series: "B", t: 3, x: 3, y: 2 }
]);
```

Query example

```
db.measurements.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$series",
      sortBy: { t: 1 },
      output: {
        covariance: {
          $covariancePop: [ "$x", "$y" ],
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "series": "A", "t": 1, "x": 1, "y": 2, "covariance": 2 },
  { "_id": 2, "series": "A", "t": 2, "x": 2, "y": 5, "covariance": 2 },
  { "_id": 3, "series": "A", "t": 3, "x": 3, "y": 8, "covariance": 2 },
  { "_id": 4, "series": "B", "t": 1, "x": 1, "y": 8, "covariance": -2 },
  { "_id": 5, "series": "B", "t": 2, "x": 2, "y": 5, "covariance": -2 },
  { "_id": 6, "series": "B", "t": 3, "x": 3, "y": 2, "covariance": -2 }
]
```

Every document in a partition receives the same `covariance`, the population covariance of `x` and `y` across all documents in that partition: `2` for series `A` (x and y increase together) and `-2` for series `B` (y decreases as x increases).

Code examples

To view a code example for using the `$covariancePop` operator in the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const measurements = db.collection('measurements');

    await measurements.insertMany([
      { _id: 1, series: "A", t: 1, x: 1, y: 2 },
      { _id: 2, series: "A", t: 2, x: 2, y: 5 },
      { _id: 3, series: "A", t: 3, x: 3, y: 8 },
      { _id: 4, series: "B", t: 1, x: 1, y: 8 },
      { _id: 5, series: "B", t: 2, x: 2, y: 5 },
      { _id: 6, series: "B", t: 3, x: 3, y: 2 }
    ]);

    const result = await measurements.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$series",
          sortBy: { t: 1 },
          output: {
            covariance: {
              $covariancePop: [ "$x", "$y" ],
              window: { documents: ["unbounded", "unbounded"] }
            }
          }
        }
      }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        measurements = db['measurements']

        measurements.insert_many([
            { '_id': 1, 'series': 'A', 't': 1, 'x': 1, 'y': 2 },
            { '_id': 2, 'series': 'A', 't': 2, 'x': 2, 'y': 5 },
            { '_id': 3, 'series': 'A', 't': 3, 'x': 3, 'y': 8 },
            { '_id': 4, 'series': 'B', 't': 1, 'x': 1, 'y': 8 },
            { '_id': 5, 'series': 'B', 't': 2, 'x': 2, 'y': 5 },
            { '_id': 6, 'series': 'B', 't': 3, 'x': 3, 'y': 2 }
        ])

        result = list(measurements.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$series',
                    'sortBy': { 't': 1 },
                    'output': {
                        'covariance': {
                            '$covariancePop': [ '$x', '$y' ],
                            'window': { 'documents': ['unbounded', 'unbounded'] }
                        }
                    }
                }
            }
        ]))

        print(result)
    finally:
        client.close()
```

```
example()
```

$covarianceSamp

New from version 8.0.2.

The `$covarianceSamp` operator in Amazon DocumentDB returns the sample covariance of two numeric expressions. It is a window operator that is used only in the `$setWindowFields` stage; it is not valid in the `$group` or `$bucket` stages. You specify the operator under the stage's `output` field, and optionally define the window boundaries with a `window` document.

Parameters

- `$covarianceSamp` takes a two-element array `[ <expression1>, <expression2> ]`, where each element is a numeric expression (a field path such as `"$x"`, an expression, or a constant). It returns the sample covariance of the two expressions across the documents in the window.

Example (MongoDB Shell)

The following example partitions the documents by `series`, sorts each partition by `t`, and computes the sample covariance of `x` and `y` over the whole partition.

Create sample documents

```
db.measurements.insertMany([
  { _id: 1, series: "A", t: 1, x: 1, y: 2 },
  { _id: 2, series: "A", t: 2, x: 2, y: 5 },
  { _id: 3, series: "A", t: 3, x: 3, y: 8 },
  { _id: 4, series: "B", t: 1, x: 1, y: 8 },
  { _id: 5, series: "B", t: 2, x: 2, y: 5 },
  { _id: 6, series: "B", t: 3, x: 3, y: 2 }
]);
```

Query example

```
db.measurements.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$series",
      sortBy: { t: 1 },
```

```
      output: {
        covariance: {
          $covarianceSamp: [ "$x", "$y" ],
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "series": "A", "t": 1, "x": 1, "y": 2, "covariance": 3 },
  { "_id": 2, "series": "A", "t": 2, "x": 2, "y": 5, "covariance": 3 },
  { "_id": 3, "series": "A", "t": 3, "x": 3, "y": 8, "covariance": 3 },
  { "_id": 4, "series": "B", "t": 1, "x": 1, "y": 8, "covariance": -3 },
  { "_id": 5, "series": "B", "t": 2, "x": 2, "y": 5, "covariance": -3 },
  { "_id": 6, "series": "B", "t": 3, "x": 3, "y": 2, "covariance": -3 }
]
```

Every document in a partition receives the same `covariance`, the sample covariance of `x` and `y` across all documents in that partition: `3` for series `A` (x and y increase together) and `-3` for series `B` (y decreases as x increases). The sample covariance divides by `n - 1` rather than n, so it is larger in magnitude than the population covariance for the same data.

Code examples

To view a code example for using the `$covarianceSamp` operator in the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
```

```
    const db = client.db('test');
    const measurements = db.collection('measurements');

    await measurements.insertMany([
      { _id: 1, series: "A", t: 1, x: 1, y: 2 },
      { _id: 2, series: "A", t: 2, x: 2, y: 5 },
      { _id: 3, series: "A", t: 3, x: 3, y: 8 },
      { _id: 4, series: "B", t: 1, x: 1, y: 8 },
      { _id: 5, series: "B", t: 2, x: 2, y: 5 },
      { _id: 6, series: "B", t: 3, x: 3, y: 2 }
    ]);

    const result = await measurements.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$series",
          sortBy: { t: 1 },
          output: {
            covariance: {
              $covarianceSamp: [ "$x", "$y" ],
              window: { documents: ["unbounded", "unbounded"] }
            }
          }
        }
      }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    try:
        db = client['test']
        measurements = db['measurements']

        measurements.insert_many([
            { '_id': 1, 'series': 'A', 't': 1, 'x': 1, 'y': 2 },
            { '_id': 2, 'series': 'A', 't': 2, 'x': 2, 'y': 5 },
            { '_id': 3, 'series': 'A', 't': 3, 'x': 3, 'y': 8 },
            { '_id': 4, 'series': 'B', 't': 1, 'x': 1, 'y': 8 },
            { '_id': 5, 'series': 'B', 't': 2, 'x': 2, 'y': 5 },
            { '_id': 6, 'series': 'B', 't': 3, 'x': 3, 'y': 2 }
        ])

        result = list(measurements.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$series',
                    'sortBy': { 't': 1 },
                    'output': {
                        'covariance': {
                            '$covarianceSamp': [ '$x', '$y' ],
                            'window': { 'documents': ['unbounded', 'unbounded'] }
                        }
                    }
                }
            }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$currentOp

The `$currentOp` aggregation stage returns information about currently running operations in the database. This stage is useful for monitoring active queries and operations in an aggregation pipeline.

Parameters

- `allUsers` (optional): When set to `true`, returns operations for all users. Default is `false`.
- `idleConnections` (optional): When set to `true`, includes idle connections. Default is `false`.
- `idleCursors` (optional): When set to `true`, includes information about idle cursors. Default is `false`.
- `idleSessions` (optional): When set to `true`, includes information about idle sessions. Default is `true`.
- `localOps` (optional): When set to `true`, includes local operations. Default is `false`.

Example (MongoDB Shell)

The following example demonstrates using the `$currentOp` aggregation stage to retrieve information about active read operations.

Query example

```
db.aggregate([
  { $currentOp: { allUsers: true, idleConnections: false } },
  { $match: { op: "query" } }
])
```

Output

```
[
  {
    "opid": "12345",
    "active": true,
    "op": "query",
    "ns": "test.users",
    "secs_running": 2
  }
]
```

Code examples

To view a code example for using the `$currentOp` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('admin');

  const result = await db.aggregate([
    { $currentOp: { allUsers: true, idleConnections: false } },
    { $match: { op: "query" } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['admin']

    result = list(db.aggregate([
        { '$currentOp': { 'allUsers': True, 'idleConnections': False } },
        { '$match': { 'op': 'query' } }
    ]))

    print(result)
    client.close()

example()
```

$dateAdd

New from version 5.0

The $dateAdd aggregation operator in Amazon DocumentDB allows you to add a duration to a date and time value.

Parameters

- date: A date and time value to add a duration to.
- duration: The duration to add to the date value. This can be specified as an object with keys for years, months, weeks, days, hours, minutes, and seconds.
- timezone: (optional) The time zone to use when performing the date addition. If not specified, the default time zone of the Amazon DocumentDB cluster is used.

Example (MongoDB Shell)

The following example demonstrates how to use the $dateAdd operator to add 2 days and 12 hours to a date.

Create sample documents

```
db.events.insertMany([
  { _id: 1, eventDate: ISODate("2023-04-01T10:00:00Z") },
  { _id: 2, eventDate: ISODate("2023-04-02T12:00:00Z") },
  { _id: 3, eventDate: ISODate("2023-04-03T14:00:00Z") }
]);
```

Query example

```
db.events.aggregate([
  {
    $project: {
      _id: 1,
      eventDate: 1,
      eventDatePlustwodaysandtwelvehours: {
        $dateAdd: {
          startDate: {
            $dateAdd: {
              startDate: "$eventDate",
              unit: "day",
              amount: 2
            }
          },
          unit: "hour",
```

```
          amount: 12
        }
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "eventDate": "2023-04-01T10:00:00Z",
    "eventDatePlustwodaysandtwelvehours": ISODate("2023-04-03T22:00:00Z)"
  },
  {
    "_id": 2,
    "eventDate": "2023-04-02T12:00:00Z",
    "eventDatePlustwodaysandtwelvehours": ISODate("2023-04-05T00:00:00Z)"
  },
  {
    "_id": 3,
    "eventDate": "2023-04-03T14:00:00Z",
    "eventDatePlustwodaysandtwelvehours": ISODate("2023-04-06T02:00:00Z)"
  }
]
```

Code examples

To view a code example for using the `$dateAdd` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('events');
```

```
  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        eventDate: 1,
        eventDatePlustwodaysandtwelvehours: {
          $dateAdd: {
            startDate: {
              $dateAdd: {
                startDate: "$eventDate",
                unit: "day",
                amount: 2
              }
            },
            unit: "hour",
            amount: 12
          }
        }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
from bson.date_time import datetime

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['events']

    result = list(collection.aggregate([
        {
```

```
            '$project': {
                '_id': 1,
                'eventDate': 1,
                'eventDatePlustwodaysandtwelvehours': {
                    '$dateAdd': {
                        'startDate': {
                            '$dateAdd' : {
                                'startDate': '$eventDate',
                                'unit': 'day',
                                'amount': 2,
                            }
                        },
                        'unit': 'hour',
                        'amount': 12,
                    }
                }
            }
        }
    ]))

    print(result)

    client.close()

example()
```

$dateDiff

New from version 5.0

Not supported by Elastic cluster.

The `$dateDiff` aggregation operator calculates the difference between two dates in specified units. It returns the number of unit boundaries crossed between the start and end dates.

Parameters

- `startDate`: The beginning date expression.
- `endDate`: The ending date expression.
- `unit`: The time unit for the difference. Supported units are `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`, and `millisecond`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$dateDiff` operator to calculate the number of days between order placement and delivery.

Create sample documents

```
db.shipments.insertMany([
  {
    orderId: 1001,
    orderDate: ISODate("2025-01-10T08:00:00Z"),
    deliveryDate: ISODate("2025-01-15T14:30:00Z")
  },
  {
    orderId: 1002,
    orderDate: ISODate("2025-02-05T10:00:00Z"),
    deliveryDate: ISODate("2025-02-12T16:45:00Z")
  }
]);
```

Query example

```
db.shipments.aggregate([
  {
    $project: {
      orderId: 1,
      orderDate: 1,
      deliveryDate: 1,
      daysToDeliver: {
        $dateDiff: {
          startDate: "$orderDate",
          endDate: "$deliveryDate",
          unit: "day"
        }
      }
    }
  }
]);
```

Output

```
[
  {
```

```
    _id: ObjectId('6924a5f2d66dcae121d29517'),
    orderId: 1001,
    orderDate: ISODate('2025-01-10T08:00:00.000Z'),
    deliveryDate: ISODate('2025-01-15T14:30:00.000Z'),
    daysToDeliver: 5
  },
  {
    _id: ObjectId('6924a5f2d66dcae121d29518'),
    orderId: 1002,
    orderDate: ISODate('2025-02-05T10:00:00.000Z'),
    deliveryDate: ISODate('2025-02-12T16:45:00.000Z'),
    daysToDeliver: 7
  }
]
```

Code examples

To view a code example for using the `$dateDiff` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const shipments = db.collection('shipments');

  const result = await shipments.aggregate([
    {
      $project: {
        orderId: 1,
        orderDate: 1,
        deliveryDate: 1,
        daysToDeliver: {
          $dateDiff: {
            startDate: "$orderDate",
            endDate: "$deliveryDate",
            unit: "day"
          }
```

```
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    shipments = db['shipments']

    result = list(shipments.aggregate([
        {
            "$project": {
                "orderId": 1,
                "orderDate": 1,
                "deliveryDate": 1,
                "daysToDeliver": {
                    "$dateDiff": {
                        "startDate": "$orderDate",
                        "endDate": "$deliveryDate",
                        "unit": "day"
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$dateFromParts

New from version 5.0.1 and 8.0

The `$dateFromParts` aggregation operator in Amazon DocumentDB constructs and returns a date and time value from its constituent parts, such as the year, month, day, hour, minute, second, and millisecond.

Parameters

- `year`: The calendar year. Required when not using ISO 8601 parts.
- `month`, `day`, `hour`, `minute`, `second`, `millisecond`: (optional) The remaining date and time components. Omitted components default to their lowest valid value.
- `isoWeekYear`, `isoWeek`, `isoDayOfWeek`: (optional) ISO 8601 date components. Use these instead of `year`, `month`, and `day` to construct a date from an ISO 8601 week date.
- `timezone`: (optional) The time zone to use when constructing the date. If not specified, UTC is used.

Example (MongoDB Shell)

The following example demonstrates how to use the `$dateFromParts` operator to construct a date from individual components.

Create sample documents

```
db.parts.insertMany([
  { _id: 1, y: 2023, m: 4, d: 1 },
  { _id: 2, y: 2023, m: 12, d: 25 }
]);
```

Query example

```
db.parts.aggregate([
  {
    $project: {
      _id: 1,
      constructedDate: {
        $dateFromParts: { year: "$y", month: "$m", day: "$d" }
      }
```

```
    }
  }
])
```

Output

```
[
  { "_id": 1, "constructedDate": ISODate("2023-04-01T00:00:00Z") },
  { "_id": 2, "constructedDate": ISODate("2023-12-25T00:00:00Z") }
]
```

Code examples

To view a code example for using the `$dateFromParts` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('parts');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        constructedDate: {
          $dateFromParts: { year: "$y", month: "$m", day: "$d" }
        }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['parts']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'constructedDate': {
                    '$dateFromParts': { 'year': '$y', 'month': '$m', 'day': '$d' }
                }
            }
        }
    ]))

    print(result)

    client.close()

example()
```

$dateFromString

The `$dateFromString` aggregation operator in Amazon DocumentDB allows you to convert a date-time string into a date object. This is useful when your data is stored as strings but you need to perform date-based operations on the data.

Parameters

- `dateString`: A string that represents a date and time.
- `format`: (optional) A string that specifies the format of the `dateString`. If not provided, Amazon DocumentDB will attempt to parse the string in the ISO-8601 format.

- `timezone`: (optional) A string that specifies the time zone. If not provided, Amazon DocumentDB will use the time zone of the server.
- on`Error`: (optional) Specifies the action to take if the conversion fails. Possible values are `'error'` (the default, which throws an error), `'null'` (returns `null`), or `'replace'` (replaces the value with the replacement string specified in the on`ErrorMessage` option).
- on`ErrorMessage`: (optional) If on`Error` is set to `'replace'`, this option specifies the replacement string.

Example (MongoDB Shell)

The following example demonstrates how to use `$dateFromString` to convert a date string to a date object in Amazon DocumentDB.

Create sample documents

```
db.missionLog.insertMany([
{ _id: 1, event: "missionStart", logDate: "2020-03-15T13:41:33"},
{ _id: 2, event: "jumpPoint1", logDate: "2020-03-15T13:45:34"},
{ _id: 3, event: "jumpPoint2", logDate: "2020-03-15T13:48:21"},
{ _id: 4, event: "jumpPoint3", logDate: "2020-03-15T13:52:09"},
{ _id: 5, event: "missionEnd", logDate: "2020-03-15T13:58:44"}
]);
```

Query example

```
db.missionLog.aggregate([
  {
    $project: {
      event: '$event',
      logDate: {
        $dateFromString: {
          dateString: '$logDate'
        }
      }
    }
  }
]);
```

Output

```
[
  {
    "_id": 1,
    "event": "missionStart",
    "logDate": ISODate("2020-03-15T13:41:33Z")
  },
  {
    "_id": 2,
    "event": "jumpPoint1",
    "logDate": ISODate("2020-03-15T13:45:34Z")
  },
  {
    "_id": 3,
    "event": "jumpPoint2",
    "logDate": ISODate("2020-03-15T13:48:21Z")
  },
  {
    "_id": 4,
    "event": "jumpPoint3",
    "logDate": ISODate("2020-03-15T13:52:09Z")
  },
  {
    "_id": 5,
    "event": "missionEnd",
    "logDate": ISODate("2020-03-15T13:58:44Z")
  }
]
```

Code examples

To view a code example for using the `$dateFromString` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('missionLog');
```

```
  const result = await collection.aggregate([
    {
      $project: {
        event: '$event',
        logDate: {
          $dateFromString: {
            dateString: '$logDate'
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['missionLog']

    result = list(collection.aggregate([
        {
            '$project': {
                'event': '$event',
                'logDate': {
                    '$dateFromString': {
                        'dateString': '$logDate'
                    }
                }
            }
        }
    ]))
```

```
    print(result)
    client.close()

example()
```

$dateSubtract

New from version 5.0

The `$dateSubtract` aggregation operator in Amazon DocumentDB allows you to subtract a specified duration from a date value.

Parameters

- `date`: A date expression that resolves to a date or timestamp.
- `subtrahend`: A duration expression that specifies the amount of time to subtract from the `date` expression.
- `unit`: A string that specifies the time unit for the `subtrahend` expression. Supported units are "year", "quarter", "month", "week", "day", "hour", "minute", "second", and "millisecond".

Example (MongoDB Shell)

The following example demonstrates how to use the `$dateSubtract` operator to calculate the date one year ago from the current date.

Create sample documents

```
db.events.insertOne({
  eventName: "Player joined",
  eventTime: ISODate("2023-04-01T12:00:00Z")
});
```

Query example

```
db.events.aggregate([
  {
    $project: {
      eventName: 1,
      oneYearAgo: {
```

```
        $dateSubtract: {
          startDate: "$eventTime",
          amount: 1,
          unit: "year"
        }
      }
    }
  }
])
```

Output

```
{
  "_id" : ObjectId("64567890abcdef012345678"),
  "eventName" : "Player joined",
  "oneYearAgo" : ISODate("2022-04-01T12:00:00Z")
}
```

Code examples

To view a code example for using the `$dateSubtract` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    {
      $project: {
        eventName: 1,
        oneYearAgo: {
          $dateSubtract: {
            startDate: "$eventTime",
            amount: 1,
            unit: "year"
```

```
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
from bson.date_time import datetime

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    events = db['events']

    result = list(events.aggregate([
        {
            '$project': {
                'eventName': 1,
                'oneYearAgo': {
                    '$dateSubtract': {
                        'startDate': '$eventTime',
                        'amount': 1,
                        'unit': 'year'
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$dateToParts

New from version 5.0.1 and 8.0

The `$dateToParts` aggregation operator in Amazon DocumentDB returns a document that contains the constituent parts of a date and time value, such as the year, month, day, hour, minute, second, and millisecond.

Parameters

- `date`: The date and time value to break into parts.
- `timezone`: (optional) The time zone to use when extracting the date parts. If not specified, the parts are returned in UTC.
- `iso8601`: (optional) If set to `true`, the returned document uses ISO 8601 date parts (`isoWeekYear`, `isoWeek`, `isoDayOfWeek`). The default is `false`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$dateToParts` operator to return the individual parts of a date.

Create sample documents

```
db.events.insertMany([
  { _id: 1, eventDate: ISODate("2023-04-01T10:30:15Z") },
  { _id: 2, eventDate: ISODate("2023-12-25T18:45:00Z") }
]);
```

Query example

```
db.events.aggregate([
  {
    $project: {
      _id: 1,
      parts: { $dateToParts: { date: "$eventDate" } }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "parts": { "year": 2023, "month": 4, "day": 1, "hour": 10, "minute": 30, "second":
 15, "millisecond": 0 }
  },
  {
    "_id": 2,
    "parts": { "year": 2023, "month": 12, "day": 25, "hour": 18, "minute": 45,
 "second": 0, "millisecond": 0 }
  }
]
```

Code examples

To view a code example for using the `$dateToParts` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('events');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        parts: { $dateToParts: { date: "$eventDate" } }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['events']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'parts': { '$dateToParts': { 'date': '$eventDate' } }
            }
        }
    ]))

    print(result)

    client.close()

example()
```

$dateToString

The `$dateToString` aggregation operator in Amazon DocumentDB is used to convert a date or timestamp value to a string representation. This is useful when you need to format the date and time in a specific way for display or further processing.

Parameters

- `date`: The date or timestamp value to be converted to a string.
- `format`: A string that specifies the format in which the date should be represented. The format string can include various format specifiers, such as `%Y` for the four-digit year, `%m` for the two-digit month, `%d` for the two-digit day of the month, etc.

- `timezone`: (optional) The time zone to use for the conversion. If not specified, the time zone of the server hosting the Amazon DocumentDB cluster is used.
- `onNull`: (optional) The value to be returned if the `date` parameter is `null`.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$dateToString` operator to format the `logDate` field of the `missionLog` collection.

Create sample documents

```
db.missionLog.insertMany([
  { _id: 1, "event":"missionStart", logDate: new Date("2020-03-15T13:41:33Z") },
  { _id: 2, "event":"jumpPoint1", logDate: new Date("2020-03-15T13:45:34Z") },
  { _id: 3, "event":"jumpPoint2", logDate: new Date("2020-03-15T13:48:21Z") },
  { _id: 4, "event":"jumpPoint3", logDate: new Date("2020-03-15T13:52:09Z") },
  { _id: 5, "event":"missionEnd", logDate: new Date("2020-03-15T13:58:44Z") }
]);
```

Query example

```
db.missionLog.aggregate([
  {
    $project: {
      event: "$event",
      logDateFormatted: {
        $dateToString: {
          format: "%Y-%m-%d %H:%M:%S",
          date: "$logDate"
        }
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "event": "missionStart",
```

```
    "logDateFormatted": "2020-03-15 13:41:33"
  },
  {
    "_id": 2,
    "event": "jumpPoint1",
    "logDateFormatted": "2020-03-15 13:45:34"
  },
  {
    "_id": 3,
    "event": "jumpPoint2",
    "logDateFormatted": "2020-03-15 13:48:21"
  },
  {
    "_id": 4,
    "event": "jumpPoint3",
    "logDateFormatted": "2020-03-15 13:52:09"
  },
  {
    "_id": 5,
    "event": "missionEnd",
    "logDateFormatted": "2020-03-15 13:58:44"
  }
]
```

Code examples

To view a code example for using the `$dateToString` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$dateToString` operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('missionLog');

  const result = await collection.aggregate([
```

```
    {
      $project: {
        event: "$event",
        logDateFormatted: {
          $dateToString: {
            format: "%Y-%m-%d %H:%M:%S",
            date: "$logDate"
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

Here's an example of using the `$dateToString` operator in a Python application:

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['missionLog']

    pipeline = [
        {
            '$project': {
                'event': '$event',
                'logDateFormatted': {
                    '$dateToString': {
                        'format': '%Y-%m-%d %H:%M:%S',
                        'date': '$logDate'
                    }
                }
            }
        }
```

```
    ]

    result = list(collection.aggregate(pipeline))
    print(result)
    client.close()

example()
```

$dateTrunc

New from version 8.0

Not supported by Elastic cluster.

The `$dateTrunc` aggregation operator in Amazon DocumentDB truncates a date to a specified unit.

Parameters

- `date`: A date expression that resolves to a date or timestamp.
- `unit`: A string that specifies the time unit for the subtrahend expression. Supported units are `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`, and `millisecond`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$dateTrunc` operator to truncate a date to to the hour.

Create sample documents

```
db.events.insertMany([
  {
    eventName: "Event 1",
    eventTime: ISODate("2025-04-01T12:15:00Z")
  },
  {
    eventName: "Event 2",
    eventTime: ISODate("2025-08-15T14:33:22Z")
  },
]);
```

Query example

```
db.events.aggregate([
  {
    $project: {
      eventName: 1,
      eventTime: 1,
      truncatedToHour: {
        $dateTrunc: {
          date: "$eventTime",
          unit: "hour"
        }
      }
    }
  }
]);
```

Output

```
[
  {
    _id: ObjectId('6924a258d66dcae121d29515'),
    eventName: 'Event 1',
    eventTime: ISODate('2025-04-01T12:15:00.000Z'),
    truncatedToHour: ISODate('2025-04-01T12:00:00.000Z')
  },
  {
    _id: ObjectId('6924a258d66dcae121d29516'),
    eventName: 'Event 2',
    eventTime: ISODate('2025-08-15T14:33:22.000Z'),
    truncatedToHour: ISODate('2025-08-15T14:00:00.000Z')
  }
]
```

Code examples

To view a code example for using the `$dateTrunc` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    {
      $project: {
        eventName: 1,
        eventTime: 1,
        truncatedToHour: {
          $dateTrunc: {
            date: "$eventTime",
            unit: "hour"
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    events = db['events']

    result = list(events.aggregate([
        {
            "$project": {
```

```
                "eventName": 1,
                "eventTime": 1,
                "truncatedToHour": {
                    "$dateTrunc": {
                        "date": "$eventTime",
                        "unit": "hour"
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$dayOfMonth

The `$dayOfMonth` aggregation operator in Amazon DocumentDB retrieves the day of the month (from 1 to 31) for a given date. This operator is useful for grouping, filtering, or extracting the day of the month from date fields in your documents.

Parameters

- `date expression`: The date expression can be a date field from the document, a date object, or a date string.

Example (MongoDB Shell)

This example demonstrates how to use the `$dayOfMonth` operator to extract the day of the month from a date field in the document.

Create sample documents

```
db.events.insertMany([
  { _id: 1, eventDate: new Date("2022-01-15T12:00:00Z") },
  { _id: 2, eventDate: new Date("2022-02-28T15:30:00Z") },
  { _id: 3, eventDate: new Date("2022-03-01T09:45:00Z") },
  { _id: 4, eventDate: new Date("2022-04-30T23:59:59Z") }
]);
```

Query example

```
db.events.aggregate([
  { $project: {
    eventDay: { $dayOfMonth: "$eventDate" }
  }}
])
```

Output

```
[
  { "_id" : 1, "eventDay" : 15 },
  { "_id" : 2, "eventDay" : 28 },
  { "_id" : 3, "eventDay" : 1 },
  { "_id" : 4, "eventDay" : 30 }
]
```

Code examples

To view a code example for using the `$dayOfMonth` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    { $project: {
      eventDay: { $dayOfMonth: "$eventDate" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.mydatabase
    events = db.events

    result = list(events.aggregate([
        { "$project": {
            "eventDay": { "$dayOfMonth": "$eventDate" }
        }}
    ]))

    print(result)
    client.close()

example()
```

$dayOfWeek

The `$dayOfWeek` operator in Amazon DocumentDB extracts the day of the week from a given date field. It returns the day of the week as a number between 1 (Sunday) and 7 (Saturday), which is the same behavior as in MongoDB.

Parameters

- `date field`: The date field to extract the day of the week from.

Example (MongoDB Shell)

This example demonstrates how to use the `$dayOfWeek` operator to extract the day of the week from the `date` field in the `weather` collection.

Create sample documents

```
db.weather.insertMany([
  {
    "temperature": 97.5,
    "humidity": 0.60,
    "date": new Date("2023-04-01")
  },
  {
    "temperature": 95.2,
    "humidity": 0.55,
    "date": new Date("2023-04-02")
  },
  {
    "temperature": 92.8,
    "humidity": 0.65,
    "date": new Date("2023-04-03")
  }
]);
```

Query example

```
db.weather.aggregate([
  {
    $project: {
      dayOfWeek: { $dayOfWeek: "$date" }
    }
  }
]).pretty();
```

Output

```
{ "_id" : ObjectId("64272c6663f4f8ce422c2d91"), "dayOfWeek" : 7 }
{ "_id" : ObjectId("64272c6663f4f8ce422c2d92"), "dayOfWeek" : 1 }
{ "_id" : ObjectId("64272c6663f4f8ce422c2d93"), "dayOfWeek" : 2 }
```

Code examples

To view a code example for using the `$dayOfWeek` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('weather');

  const result = await collection.aggregate([
    {
      $project: {
        dayOfWeek: { $dayOfWeek: '$date' }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.weather

    result = list(collection.aggregate([
        {
            '$project': {
                'dayOfWeek': { '$dayOfWeek': '$date' }
            }
        }
    ]))

    print(result)
```

```
    client.close()

if __name__ == '__main__':
    main()
```

$dayOfYear

The `$dayOfYear` operator in Amazon DocumentDB returns the day of the year for a date as a number between 1 and 366 (365 in non-leap years).

Parameters

- `expression`: The date field or expression from which to extract the day of the year.

Example (MongoDB Shell)

This example demonstrates how to use the `$dayOfYear` operator to extract the day of the year from a date field in an Amazon DocumentDB collection.

Create sample documents

```
db.weather.insert([
  {
    "temperature" : 97.5,
    "humidity": 0.60,
    "date" : new Date("2023-03-15")
  },
  {
    "temperature" : 82.3,
    "humidity": 0.75,
    "date" : new Date("2023-12-31")
  }
])
```

Query example

```
db.weather.aggregate([
  {
    $project: {
```

```
      dayOfYear: { $dayOfYear: "$date" }
    }
  }
]).pretty()
```

Output

```
{ "_id" : ObjectId("642b86fc7d8e07af279bbe63"), "dayOfYear" : 74 }
{ "_id" : ObjectId("642b86fc7d8e07af279bbe64"), "dayOfYear" : 365 }
```

Code examples

To view a code example for using the `$dayOfYear` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('weather');

  const result = await collection.aggregate([
    {
      $project: {
        dayOfYear: { $dayOfYear: "$date" }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['weather']

    result = list(collection.aggregate([
        {
            '$project': {
                'dayOfYear': { '$dayOfYear': '$date' }
            }
        }
    ]))

    print(result)

    client.close()

example()
```

$degreesToRadians

New from version 8.0.1.

The `$degreesToRadians` operator in Amazon DocumentDB converts a value from degrees to radians.

Parameters

- `expression`: An expression that resolves to a numeric value in degrees.

Example (MongoDB Shell)

The following example shows how to use the `$degreesToRadians` operator to convert degree values to radians.

Create sample documents

```
db.angles.insertMany([
  { "_id": 1, "degrees": 0 },
  { "_id": 2, "degrees": 90 },
  { "_id": 3, "degrees": 180 },
  { "_id": 4, "degrees": 360 }
]);
```

Query example

```
db.angles.aggregate([
  { $project: {
    "radians": { $degreesToRadians: "$degrees" }
  }}
]);
```

Output

```
[
  { "_id": 1, "radians": 0 },
  { "_id": 2, "radians": 1.5707963267948966 },
  { "_id": 3, "radians": 3.141592653589793 },
  { "_id": 4, "radians": 6.283185307179586 }
]
```

Code examples

To view a code example for using the `$degreesToRadians` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('angles');

  const result = await collection.aggregate([
```

```
    { $project: {
      "radians": { $degreesToRadians: "$degrees" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['angles']

    result = list(collection.aggregate([
        { '$project': {
            'radians': { '$degreesToRadians': '$degrees' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$denseRank

New from version 8.0.2.

The $denseRank window operator in Amazon DocumentDB returns the position of each document within its partition, according to the $setWindowFields stage sortBy order. Documents with the same sortBy value receive the same rank, and the next distinct value uses the immediately

following position, leaving no gaps (for example, `1, 2, 2, 3`). It is specified under the `output` field of the `$setWindowFields` stage.

It is a window operator that is used only in the `$setWindowFields` stage.

Parameters

- `$denseRank` takes an empty document `{}` as its value.
- It does not accept a `window` frame.
- It requires exactly one `sortBy` key in the `$setWindowFields` stage.

Example (MongoDB Shell)

The following example partitions the documents by `game`, sorts each partition by `score` in descending order, and assigns each player a dense rank.

Create sample documents

```
db.players.insertMany([
  { _id: 1, game: "A", player: "Alice", score: 90 },
  { _id: 2, game: "A", player: "Bob", score: 85 },
  { _id: 3, game: "A", player: "Carol", score: 85 },
  { _id: 4, game: "A", player: "Dave", score: 70 },
  { _id: 5, game: "B", player: "Erin", score: 95 },
  { _id: 6, game: "B", player: "Frank", score: 80 }
]);
```

Query example

```
db.players.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$game",
      sortBy: { score: -1 },
      output: {
        denseRank: { $denseRank: {} }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "game": "A", "player": "Alice", "score": 90, "denseRank": 1 },
  { "_id": 2, "game": "A", "player": "Bob", "score": 85, "denseRank": 2 },
  { "_id": 3, "game": "A", "player": "Carol", "score": 85, "denseRank": 2 },
  { "_id": 4, "game": "A", "player": "Dave", "score": 70, "denseRank": 3 },
  { "_id": 5, "game": "B", "player": "Erin", "score": 95, "denseRank": 1 },
  { "_id": 6, "game": "B", "player": "Frank", "score": 80, "denseRank": 2 }
]
```

Within each `game` partition, players are ranked by descending `score`. Bob and Carol tie at 85 and share rank 2, and Dave follows at rank 3 with no gap.

Code examples

To view a code example for using the `$denseRank` operator in the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const players = db.collection('players');

    await players.insertMany([
      { _id: 1, game: "A", player: "Alice", score: 90 },
      { _id: 2, game: "A", player: "Bob", score: 85 },
      { _id: 3, game: "A", player: "Carol", score: 85 },
      { _id: 4, game: "A", player: "Dave", score: 70 },
      { _id: 5, game: "B", player: "Erin", score: 95 },
      { _id: 6, game: "B", player: "Frank", score: 80 }
    ]);

    const result = await players.aggregate([
```

```
      {
        $setWindowFields: {
          partitionBy: "$game",
          sortBy: { score: -1 },
          output: {
            denseRank: { $denseRank: {} }
          }
        }
      }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        players = db['players']

        players.insert_many([
            { '_id': 1, 'game': 'A', 'player': 'Alice', 'score': 90 },
            { '_id': 2, 'game': 'A', 'player': 'Bob', 'score': 85 },
            { '_id': 3, 'game': 'A', 'player': 'Carol', 'score': 85 },
            { '_id': 4, 'game': 'A', 'player': 'Dave', 'score': 70 },
            { '_id': 5, 'game': 'B', 'player': 'Erin', 'score': 95 },
            { '_id': 6, 'game': 'B', 'player': 'Frank', 'score': 80 }
        ])

        result = list(players.aggregate([
            {
```

```
                '$setWindowFields': {
                    'partitionBy': '$game',
                    'sortBy': { 'score': -1 },
                    'output': {
                        'denseRank': { '$denseRank': {} }
                    }
                }
            }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$divide

The `$divide` operator in the Amazon DocumentDB aggregation pipeline is used to divide one number by another. It is a useful operator for performing mathematical operations on numeric fields within your documents.

Parameters

- `numerator`: The dividend or the number to be divided.
- `denominator`: The divisor or the number to divide by.

Example (MongoDB Shell)

This example demonstrates how to use the `$divide` operator to calculate the hourly rate for employees based on their yearly salary and the number of working hours per year.

Create sample documents

```
db.employees.insertMany([
  { _id: 1, name: "John Doe", salary: 80000, hoursPerYear: 2080 },
  { _id: 2, name: "Jane Smith", salary: 90000, hoursPerYear: 2080 },
  { _id: 3, name: "Bob Johnson", salary: 75000, hoursPerYear: 2080 }
]);
```

Query example

```
db.employees.aggregate([
  {
    $project: {
      name: 1,
      hourlyRate: { $divide: ["$salary", "$hoursPerYear"] }
    }
  }
])
```

Output

```
[
  { "_id" : 1, "name" : "John Doe", "hourlyRate" : 38.46153846153846 },
  { "_id" : 2, "name" : "Jane Smith", "hourlyRate" : 43.26923076923077 },
  { "_id" : 3, "name" : "Bob Johnson", "hourlyRate" : 36.05769230769231 }
]
```

Code examples

To view a code example for using the `$divide` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function calculateHourlyRate() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const employees = db.collection('employees');

  const result = await employees.aggregate([
    {
      $project: {
        name: 1,
        hourlyRate: { $divide: ["$salary", "$hoursPerYear"] }
      }
    }
  ]).toArray();
```

```
  console.log(result);
  client.close();
}

calculateHourlyRate();
```

Python

```
from pymongo import MongoClient

def calculate_hourly_rate():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.mydatabase
    employees = db.employees

    result = list(employees.aggregate([
        {
            '$project': {
                'name': 1,
                'hourlyRate': { '$divide': ['$salary', '$hoursPerYear'] }
            }
        }
    ]))

    print(result)
    client.close()

calculate_hourly_rate()
```

$documentNumber

New from version 8.0.2.

The `$documentNumber` window operator in Amazon DocumentDB returns a distinct sequential number for each document in its partition, according to the `$setWindowFields` stage `sortBy` order. Unlike `$rank` and `$denseRank`, it does not treat ties specially: every document receives a unique number (for example, `1, 2, 3, 4`), and documents with equal `sortBy` values are numbered in the order they are encountered. It is specified under the `output` field of the `$setWindowFields` stage.

It is a window operator that is used only in the `$setWindowFields` stage.

Parameters

- `$documentNumber` takes an empty document `{}` as its value.
- It does not accept a `window` frame.
- It requires exactly one `sortBy` key in the `$setWindowFields` stage.

Example (MongoDB Shell)

The following example partitions the documents by `game`, sorts each partition by `score` in descending order, and assigns each player a sequential document number.

Create sample documents

```
db.players.insertMany([
  { _id: 1, game: "A", player: "Alice", score: 90 },
  { _id: 2, game: "A", player: "Bob", score: 85 },
  { _id: 3, game: "A", player: "Carol", score: 85 },
  { _id: 4, game: "A", player: "Dave", score: 70 },
  { _id: 5, game: "B", player: "Erin", score: 95 },
  { _id: 6, game: "B", player: "Frank", score: 80 }
]);
```

Query example

```
db.players.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$game",
      sortBy: { score: -1 },
      output: {
        documentNumber: { $documentNumber: {} }
      }
    }
  }
]);
```

Output

```
[
```

```
  { "_id": 1, "game": "A", "player": "Alice", "score": 90, "documentNumber": 1 },
  { "_id": 2, "game": "A", "player": "Bob", "score": 85, "documentNumber": 2 },
  { "_id": 3, "game": "A", "player": "Carol", "score": 85, "documentNumber": 3 },
  { "_id": 4, "game": "A", "player": "Dave", "score": 70, "documentNumber": 4 },
  { "_id": 5, "game": "B", "player": "Erin", "score": 95, "documentNumber": 1 },
  { "_id": 6, "game": "B", "player": "Frank", "score": 80, "documentNumber": 2 }
]
```

Within each `game` partition, each player receives a distinct sequential number in descending `score` order. Bob and Carol tie at 85 but are still numbered 2 and 3 in the order they are encountered.

Code examples

To view a code example for using the `$documentNumber` operator in the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const players = db.collection('players');

    await players.insertMany([
      { _id: 1, game: "A", player: "Alice", score: 90 },
      { _id: 2, game: "A", player: "Bob", score: 85 },
      { _id: 3, game: "A", player: "Carol", score: 85 },
      { _id: 4, game: "A", player: "Dave", score: 70 },
      { _id: 5, game: "B", player: "Erin", score: 95 },
      { _id: 6, game: "B", player: "Frank", score: 80 }
    ]);

    const result = await players.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$game",
```

```
          sortBy: { score: -1 },
          output: {
            documentNumber: { $documentNumber: {} }
          }
        }
      }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        players = db['players']

        players.insert_many([
            { '_id': 1, 'game': 'A', 'player': 'Alice', 'score': 90 },
            { '_id': 2, 'game': 'A', 'player': 'Bob', 'score': 85 },
            { '_id': 3, 'game': 'A', 'player': 'Carol', 'score': 85 },
            { '_id': 4, 'game': 'A', 'player': 'Dave', 'score': 70 },
            { '_id': 5, 'game': 'B', 'player': 'Erin', 'score': 95 },
            { '_id': 6, 'game': 'B', 'player': 'Frank', 'score': 80 }
        ])

        result = list(players.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$game',
                    'sortBy': { 'score': -1 },
```

```
                    'output': {
                        'documentNumber': { '$documentNumber': {} }
                    }
                }
            }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$eq

The $eq aggregation operator compares two values and returns `true` if they are equal, otherwise returns `false`.

Parameters

- `expression1`: The first value to compare.
- `expression2`: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the $eq operator to check if product quantities match target values.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "Widget", qty: 50, target: 50 },
  { _id: 2, item: "Gadget", qty: 30, target: 50 },
  { _id: 3, item: "Tool", qty: 50, target: 40 }
]);
```

Query example

```
db.inventory.aggregate([
  {
```

```
    $project: {
      item: 1,
      qty: 1,
      target: 1,
      meetsTarget: { $eq: ["$qty", "$target"] }
    }
  }
]);
```

Output

```
[
  { _id: 1, item: 'Widget', qty: 50, target: 50, meetsTarget: true },
  { _id: 2, item: 'Gadget', qty: 30, target: 50, meetsTarget: false },
  { _id: 3, item: 'Tool', qty: 50, target: 40, meetsTarget: false }
]
```

Code examples

To view a code example for using the $eq aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  const result = await collection.aggregate([
    {
      $project: {
        item: 1,
        qty: 1,
        target: 1,
        meetsTarget: { $eq: ["$qty", "$target"] }
      }
    }
```

```
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    result = list(collection.aggregate([
        {
            '$project': {
                'item': 1,
                'qty': 1,
                'target': 1,
                'meetsTarget': { '$eq': ['$qty', '$target'] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$exp

New from version 4.0

The $exp operator in Amazon DocumentDB allows you to raise the constant e to a given number.

Parameters

- `expression`: The expression to evaluate. This can be any valid aggregation expression, including field references, arithmetic operations, and other aggregation stages.

Example (MongoDB Shell)

The following example demonstrates the use of the `$exp` operator to find all documents where the `quantity` field is greater than the `price` field.

Create sample documents

```
db.items.insertMany([
  { item: "canvas", quantity: 4 },
  { item: "journal", quantity: 2 }
]);
```

Query example

```
db.items.aggregate([
  { $project: {
    "quantityRaised": {$exp: "$quantity"}}
  }
]);
```

Output

```
[
  {
    _id: ObjectId('6920b785311cf98b79d2950d'),
    quantityRaised: 54.598150033144236
  },
  {
    _id: ObjectId('6920b785311cf98b79d2950e'),
    quantityRaised: 7.38905609893065
  }
]
```

Code examples

To view a code example for using the `$exp` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function aggregateExp() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const items = db.collection('items');

  const result = await items.aggregate([
    { $project: {
      "quantityRaised": {$exp: "$quantity"}}
    }
    ]).toArray();

  console.log(result);
  client.close();
}

aggregateExp();
```

Python

```
from pymongo import MongoClient

def aggregate_exp():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    items = db.items

    result = list(items.aggregate([
      { "$project": {
        "quantityRaised": {"$exp": "$quantity"}}
      }
    ]))

    print(result)
    client.close()
```

```
aggregate_exp()
```

$facet

New from version 8.0.2.

The `$facet` aggregation stage in Amazon DocumentDB processes multiple aggregation pipelines within a single stage on the same set of input documents. Each output field defines its own pipeline, and every pipeline receives the same input documents. This lets you compute several independent aggregations, such as counts, groupings, and summary statistics, in a single query instead of running the collection through multiple separate queries.

Each sub-pipeline runs independently and produces an array of documents that is stored in its corresponding output field. Because the sub-pipelines do not share state, the output of one pipeline does not affect another.

Syntax

```
{
  $facet: {
    <outputField1>: [ <stage1>, <stage2>, ... ],
    <outputField2>: [ <stage1>, <stage2>, ... ],
    ...
  }
}
```

Parameters

- `outputField` (required): The name of an output field that holds the result array of its associated aggregation pipeline. You can specify one or more output fields. Each output field maps to an array of aggregation pipeline stages that is applied to the input documents.

Output

The `$facet` stage returns a single document. Each output field that you specify contains an array of documents produced by its corresponding sub-pipeline.

Limitations

- A `$facet` sub-pipeline cannot include the following stages: `$changeStream`, `$collStats`, `$facet` (nesting is not allowed), `$geoNear`, `$indexStats`, `$listSearchIndexes`, `$merge`, `$out`, `$planCacheStats`, `$search`, `$searchMeta`, and `$vectorSearch`.
- The document that the `$facet` stage returns must stay within the 16 MB BSON document size limit.
- In versions earlier than 8.0.3, a `$sort` stage that comes before a `$facet` stage cannot determine the order of an order-sensitive accumulator (`$first`, `$last`, `$push`, `$firstN`, `$lastN`, or `$mergeObjects`) in a `$group`, `$bucket`, or `$bucketAuto` stage when that ordering would have to be applied across the `$facet` boundary. This applies whether the grouping stage is inside a `$facet` sub-pipeline or after the `$facet` stage. In this case, Amazon DocumentDB returns an error instead of running the pipeline. To order such an accumulator, place the `$sort` stage inside the `$facet` sub-pipeline, immediately before the grouping stage. Order-insensitive accumulators, such as `$sum`, `$min`, `$max`, and `$count`, are not affected.

> **Note**
>
> Starting in version 8.0.3, an order-sensitive accumulator (`$first`, `$last`, `$push`, `$firstN`, `$lastN`, or `$mergeObjects`) in a `$group`, `$bucket`, or `$bucketAuto` stage inside a `$facet` sub-pipeline honors the order of a preceding `$sort` stage. Such a stage placed after the `$facet` stage also no longer returns an error.

Example (MongoDB Shell)

The following example demonstrates how to use the `$facet` stage to compute several aggregations over the same sales data in a single query.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", price: 10, quantity: 2, date: new Date("2020-09-01") },
  { item: "def", price: 20, quantity: 1, date: new Date("2020-10-01") },
  { item: "ghi", price: 5, quantity: 3, date: new Date("2020-11-01") },
  { item: "jkl", price: 15, quantity: 2, date: new Date("2020-12-01") },
  { item: "abc", price: 25, quantity: 1, date: new Date("2021-01-01") }
]);
```

Query example

The following command uses `$facet` to compute, in a single stage, the total quantity sold per item, overall price statistics, and the total document count:

```
db.sales.aggregate([
  {
    $facet: {
      "totalByItem": [
        { $group: { _id: "$item", totalQuantity: { $sum: "$quantity" } } }
      ],
      "priceStats": [
        { $group: { _id: null, avgPrice: { $avg: "$price" }, maxPrice: { $max:
 "$price" } } }
      ],
      "documentCount": [
        { $count: "total" }
      ]
    }
  }
])
```

Output

The command returns the following output:

```
[
  {
    totalByItem: [
      { _id: "abc", totalQuantity: 3 },
      { _id: "def", totalQuantity: 1 },
      { _id: "ghi", totalQuantity: 3 },
      { _id: "jkl", totalQuantity: 2 }
    ],
    priceStats: [
      { _id: null, avgPrice: 15, maxPrice: 25 }
    ],
    documentCount: [
      { total: 5 }
    ]
  }
]
```

Code examples

To view a code example for using the `$facet` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const sales = db.collection('sales');

  const result = await sales.aggregate([
    {
      $facet: {
        "totalByItem": [
          { $group: { _id: "$item", totalQuantity: { $sum: "$quantity" } } }
        ],
        "priceStats": [
          { $group: { _id: null, avgPrice: { $avg: "$price" }, maxPrice: { $max:
 "$price" } } }
        ],
        "documentCount": [
          { $count: "total" }
        ]
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    sales = db['sales']

    result = list(sales.aggregate([
        {
            '$facet': {
                'totalByItem': [
                    {'$group': {'_id': '$item', 'totalQuantity': {'$sum':
 '$quantity'}}}
                ],
                'priceStats': [
                    {'$group': {'_id': None, 'avgPrice': {'$avg': '$price'},
 'maxPrice': {'$max': '$price'}}}
                ],
                'documentCount': [
                    {'$count': 'total'}
                ]
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$filter

The `$filter` operator in Amazon DocumentDB is used to apply a filter expression to each element of an array and return an array containing only the elements that match the specified condition. This operator is useful when you need to perform complex filtering operations on array fields within your documents.

Parameters

- `input`: The array field to filter.
- `as`: The variable name to use for each element of the `input` array within the `cond` expression.

- cond: The boolean expression that determines whether a given element should be included in the output array.

Example (MongoDB Shell)

The following example demonstrates how to use the `$filter` operator to project each order's customer and create a new array field paidItems containing only the items from the items array where the price is greater than 15. Essentially, it filters each order's items to include only products that cost more than 15.

Create sample documents

```
db.orders.insertMany([
  { _id: 1, customer: "abc123", items: [
    { name: "Product A", price: 10, qty: 2 },
    { name: "Product B", price: 20, qty: 1 }
  ]},
  { _id: 2, customer: "def456", items: [
    { name: "Product C", price: 5, qty: 3 },
    { name: "Product D", price: 15, qty: 4 }
  ]},
  { _id: 3, customer: "ghi789", items: [
    { name: "Product E", price: 8, qty: 3 },
    { name: "Product F", price: 12, qty: 1 }
  ]}
]);
```

Query example

```
db.orders.aggregate([
  {
    $project: {
      customer: 1,
      paidItems: {
        $filter: {
          input: "$items",
          as: "item",
          cond: { $gt: ["$$item.price", 15] }
        }
      }
    }
  }
```

```
]).pretty();
```

Output

```
[
  {
    _id: 1,
    customer: 'abc123',
    paidItems: [ { name: 'Product B', price: 20, qty: 1 } ]
  },
  { _id: 2, customer: 'def456', paidItems: [] },
  { _id: 3, customer: 'ghi789', paidItems: [] }
]
```

Code examples

To view a code example for using the `$filter` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');
    const collection = db.collection('orders');

    const result = await collection.aggregate([
      {
        $project: {
          customer: 1,
          paidItems: {
            $filter: {
              input: "$items",
              as: "item",
```

```
              cond: { $gt: ["$$item.price", 15] }
            }
          }
        }
      }
    ]).toArray();

    console.log(JSON.stringify(result, null, 2));

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false'

    with MongoClient(uri) as client:
        db = client.test
        collection = db.orders

        result = list(collection.aggregate([
            {
                '$project': {
                    'customer': 1,
                    'paidItems': {
                        '$filter': {
                            'input': '$items',
                            'as': 'item',
                            'cond': { '$gt': ['$$item.price', 15] }
                        }
                    }
```

```
                }
            }
        ]))

        for doc in result:
            pprint(doc)

example()
```

$first

New from version 5.0.

Not supported by Elastic cluster.

The `$first` operator in Amazon DocumentDB returns the first document from a grouped set of documents. It is commonly used in aggregation pipelines to retrieve the first document that matches a specific condition.

Parameters

- `expression`: The expression to return as the first value in each group.

Example (MongoDB Shell)

The following example demonstrates the use of the `$first` operator to retrieve the first item value encountered for each category during the aggregation.

Note: `$first` returns the first document based on the current order of documents in the pipeline. To ensure a specific order (e.g., by date, price, etc.), a `$sort` stage should be used before the `$group` stage.

Create sample documents

```
db.products.insertMany([
  { _id: 1, item: "abc", price: 10, category: "food" },
  { _id: 2, item: "jkl", price: 20, category: "food" },
  { _id: 3, item: "xyz", price: 5, category: "toy" },
  { _id: 4, item: "abc", price: 5, category: "toy" }
]);
```

Query example

```
db.products.aggregate([
  { $group: { _id: "$category", firstItem: { $first: "$item" } } }
]);
```

Output

```
[
  { "_id" : "food", "firstItem" : "abc" },
  { "_id" : "toy", "firstItem" : "xyz" }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$first` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the value of the `expression` from the first document in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$first` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the first (earliest) price recorded in each partition.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        firstPrice: {
          $first: "$price",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "firstPrice": 50 },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "firstPrice": 50 },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "firstPrice": 50 },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "firstPrice": 50 },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "firstPrice": 30 },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "firstPrice": 30 },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "firstPrice": 30 }
]
```

Each document is augmented with `firstPrice`, the price from the first document in its window (the earliest `hour` in the partition).

Code examples

To view a code example for using the `$first` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: first value per group
    const products = db.collection('products');
    await products.insertMany([
      { _id: 1, item: "abc", price: 10, category: "food" },
      { _id: 2, item: "jkl", price: 20, category: "food" },
      { _id: 3, item: "xyz", price: 5, category: "toy" },
      { _id: 4, item: "abc", price: 5, category: "toy" }
    ]);
    const accumulatorResult = await products.aggregate([
      { $group: { _id: "$category", firstItem: { $first: "$item" } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: first value within each partition
    const stockPrices = db.collection('stockPrices');
    await stockPrices.insertMany([
      { _id: 1, ticker: "ABC", hour: 1, price: 50 },
      { _id: 2, ticker: "ABC", hour: 2, price: 45 },
      { _id: 3, ticker: "ABC", hour: 3, price: 60 },
      { _id: 4, ticker: "ABC", hour: 4, price: 40 },
      { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
      { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
      { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
    ]);
    const windowResult = await stockPrices.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$ticker",
          sortBy: { hour: 1 },
          output: {
            firstPrice: {
              $first: "$price",
```

```
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: first value per group
        products = db['products']
        products.insert_many([
            { '_id': 1, 'item': 'abc', 'price': 10, 'category': 'food' },
            { '_id': 2, 'item': 'jkl', 'price': 20, 'category': 'food' },
            { '_id': 3, 'item': 'xyz', 'price': 5, 'category': 'toy' },
            { '_id': 4, 'item': 'abc', 'price': 5, 'category': 'toy' }
        ])
        accumulator_result = list(products.aggregate([
            { '$group': { '_id': '$category', 'firstItem': { '$first': '$item' } } }
        ]))
        pprint(accumulator_result)
```

```
        # Window operator usage: first value within each partition
        stock_prices = db['stockPrices']
        stock_prices.insert_many([
            { '_id': 1, 'ticker': 'ABC', 'hour': 1, 'price': 50 },
            { '_id': 2, 'ticker': 'ABC', 'hour': 2, 'price': 45 },
            { '_id': 3, 'ticker': 'ABC', 'hour': 3, 'price': 60 },
            { '_id': 4, 'ticker': 'ABC', 'hour': 4, 'price': 40 },
            { '_id': 5, 'ticker': 'XYZ', 'hour': 1, 'price': 30 },
            { '_id': 6, 'ticker': 'XYZ', 'hour': 2, 'price': 35 },
            { '_id': 7, 'ticker': 'XYZ', 'hour': 3, 'price': 25 }
        ])
        window_result = list(stock_prices.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$ticker',
                    'sortBy': { 'hour': 1 },
                    'output': {
                        'firstPrice': {
                            '$first': '$price',
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$firstN

New from version 8.0.1.

The $firstN operator in Amazon DocumentDB returns the first N elements. When used as an accumulator in a $group stage, it returns an array of the first N values in each group. When used as an array expression operator, it returns the first N elements of an array.

Parameters

- input: The expression that resolves to the field or array from which to return values.
- n: A positive integer that specifies how many values to return. When used as an accumulator in a $group stage, n can also be an expression, as long as it resolves to a positive integer based on the group _id field.

Example (MongoDB Shell)

The following example shows how to use the $firstN accumulator to retrieve the first two quantities for each item during the aggregation.

> **Note**
>
> $firstN selects values in the order that documents reach the $group stage. To return the first N values for a specific ordering (for example, by date or score), add a $sort stage before $group.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, date: ISODate("2023-01-01") },
  { item: "abc", quantity: 5, date: ISODate("2023-01-02") },
  { item: "abc", quantity: 8, date: ISODate("2023-01-03") },
  { item: "xyz", quantity: 15, date: ISODate("2023-01-01") },
  { item: "xyz", quantity: 7, date: ISODate("2023-01-02") },
  { item: "xyz", quantity: 3, date: ISODate("2023-01-03") }
]);
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", firstTwoQuantities: { $firstN: { input: "$quantity", n:
 2 } } } }
```

```
]);
```

Output

```
[
  { "_id": "abc", "firstTwoQuantities": [10, 5] },
  { "_id": "xyz", "firstTwoQuantities": [15, 7] }
]
```

Expression usage example (MongoDB Shell)

The `$firstN` operator can also be used as an expression within a `$project` stage to return the first N elements of an array field.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "abc", tags: ["red", "green", "blue", "yellow", "purple"] },
  { _id: 2, item: "xyz", tags: ["alpha", "beta", "gamma"] }
]);
```

Query example

```
db.inventory.aggregate([
  { $project: {
      firstThreeTags: { $firstN: { input: "$tags", n: 3 } }
    }}
]);
```

Output

```
[
  { "_id": 1, "firstThreeTags": ["red", "green", "blue"] },
  { "_id": 2, "firstThreeTags": ["alpha", "beta", "gamma"] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$firstN` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of up to the first n values for the documents in each window. You

specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$firstN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the first two prices seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        firstTwoSoFar: {
          $firstN: { input: "$price", n: 2 },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "firstTwoSoFar": [50] },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "firstTwoSoFar": [50, 45] },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "firstTwoSoFar": [50, 45] },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "firstTwoSoFar": [50, 45] },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "firstTwoSoFar": [30] },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "firstTwoSoFar": [30, 35] },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "firstTwoSoFar": [30, 35] }
]
```

Each document is augmented with `firstTwoSoFar`, the first two `price` values within its window (in `hour` order).

Code examples

To view a code example for using the `$firstN` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: first N values per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { item: "abc", quantity: 10, date: new Date("2023-01-01") },
      { item: "abc", quantity: 5, date: new Date("2023-01-02") },
      { item: "abc", quantity: 8, date: new Date("2023-01-03") },
      { item: "xyz", quantity: 15, date: new Date("2023-01-01") },
      { item: "xyz", quantity: 7, date: new Date("2023-01-02") },
      { item: "xyz", quantity: 3, date: new Date("2023-01-03") }
    ]);
```

```
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", firstTwoQuantities: { $firstN: { input: "$quantity",
n: 2 } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Expression usage: first N elements of an array field
    const inventory = db.collection('inventory');
    await inventory.insertMany([
      { _id: 1, item: "abc", tags: ["red", "green", "blue", "yellow", "purple"] },
      { _id: 2, item: "xyz", tags: ["alpha", "beta", "gamma"] }
    ]);
    const expressionResult = await inventory.aggregate([
      { $project: { firstThreeTags: { $firstN: { input: "$tags", n: 3 } } } }
    ]).toArray();
    console.log('Expression result:', expressionResult);

    // Window operator usage: first N values over a window in each partition
    const stockPrices = db.collection('stockPrices');
    await stockPrices.insertMany([
      { _id: 1, ticker: "ABC", hour: 1, price: 50 },
      { _id: 2, ticker: "ABC", hour: 2, price: 45 },
      { _id: 3, ticker: "ABC", hour: 3, price: 60 },
      { _id: 4, ticker: "ABC", hour: 4, price: 40 },
      { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
      { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
      { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
    ]);
    const windowResult = await stockPrices.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$ticker",
          sortBy: { hour: 1 },
          output: {
            firstTwoSoFar: {
              $firstN: { input: "$price", n: 2 },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);
```

```
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from datetime import datetime

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: first N values per group
        sales = db['sales']
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'date': datetime(2023, 1, 1) },
            { 'item': 'abc', 'quantity': 5, 'date': datetime(2023, 1, 2) },
            { 'item': 'abc', 'quantity': 8, 'date': datetime(2023, 1, 3) },
            { 'item': 'xyz', 'quantity': 15, 'date': datetime(2023, 1, 1) },
            { 'item': 'xyz', 'quantity': 7, 'date': datetime(2023, 1, 2) },
            { 'item': 'xyz', 'quantity': 3, 'date': datetime(2023, 1, 3) }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'firstTwoQuantities': { '$firstN':
 { 'input': '$quantity', 'n': 2 } } } }
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: first N elements of an array field
        inventory = db['inventory']
        inventory.insert_many([
            { '_id': 1, 'item': 'abc', 'tags': ['red', 'green', 'blue', 'yellow',
 'purple'] },
            { '_id': 2, 'item': 'xyz', 'tags': ['alpha', 'beta', 'gamma'] }
        ])
```

```
        expression_result = list(inventory.aggregate([
            { '$project': { 'firstThreeTags': { '$firstN': { 'input': '$tags', 'n':
 3 } } } }
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: first N values over a window in each partition
        stock_prices = db['stockPrices']
        stock_prices.insert_many([
            { '_id': 1, 'ticker': 'ABC', 'hour': 1, 'price': 50 },
            { '_id': 2, 'ticker': 'ABC', 'hour': 2, 'price': 45 },
            { '_id': 3, 'ticker': 'ABC', 'hour': 3, 'price': 60 },
            { '_id': 4, 'ticker': 'ABC', 'hour': 4, 'price': 40 },
            { '_id': 5, 'ticker': 'XYZ', 'hour': 1, 'price': 30 },
            { '_id': 6, 'ticker': 'XYZ', 'hour': 2, 'price': 35 },
            { '_id': 7, 'ticker': 'XYZ', 'hour': 3, 'price': 25 }
        ])
        window_result = list(stock_prices.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$ticker',
                    'sortBy': { 'hour': 1 },
                    'output': {
                        'firstTwoSoFar': {
                            '$firstN': { 'input': '$price', 'n': 2 },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$floor

New from version 4.0.

The `$floor` operator in Amazon DocumentDB returns the largest integer that is less than or equal to the specified number. This operator is useful for rounding down numeric values.

Parameters

- `expression`: The numeric expression to round down.

Example (MongoDB Shell)

The following example demonstrates the use of the `$floor` operator to round a decimal value down to the nearest integer.

Create sample documents

```
db.numbers.insertOne({ value: 3.14 });
```

Query example

```
db.numbers.aggregate([
  { $project: { _id: 0, floored: { $floor: "$value" } } }
]);
```

Output

```
{ "floored" : 3 }
```

Code examples

To view a code example for using the `$floor` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);
```

```
  try {
    await client.connect();

    const db = client.db('test');
    const collection = db.collection('numbers');

    const result = await collection.aggregate([
      { $project: { _id: 0, floored: { $floor: "$value" } } }
    ]).toArray();

    console.log(result);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client.test
        collection = db.numbers

        result = list(collection.aggregate([
            { '$project': { '_id': 0, 'floored': { '$floor': '$value' }}}
        ]))

        pprint(result)
```

```
    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$geoNear

The `$geoNear` aggregation stage returns documents in order of proximity to a specified point. It calculates the distance from the point and includes the distance in the output documents.

Parameters

- `near`: The point from which to calculate distances, specified as GeoJSON or legacy coordinates.
- `distanceField`: The field name to store the calculated distance.
- `spherical`: Boolean indicating whether to use spherical geometry (required for GeoJSON points).
- `maxDistance`: Optional. Maximum distance from the center point.
- `minDistance`: Optional. Minimum distance from the center point.
- `query`: Optional. Additional filter criteria to apply.
- `limit`: Optional. Maximum number of documents to return.
- `key`: Optional. Field to use for geospatial query when multiple geospatial indexes exist.

Example (MongoDB Shell)

The following example demonstrates using the `$geoNear` stage to find stores nearest to a given location.

Create sample documents

```
db.stores.createIndex({ location: "2dsphere" });

db.stores.insertMany([
  { _id: 1, name: "Store A", location: { type: "Point", coordinates: [-122.4,
 37.8] } },
```

```
  { _id: 2, name: "Store B", location: { type: "Point", coordinates: [-122.5,
 37.7] } },
  { _id: 3, name: "Store C", location: { type: "Point", coordinates: [-122.3, 37.9] } }
]);
```

Query example

```
db.stores.aggregate([
  {
    $geoNear: {
      near: { type: "Point", coordinates: [-122.4, 37.8] },
      distanceField: "distance",
      spherical: true
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Store A', location: { type: 'Point', coordinates: [ -122.4,
 37.8 ] }, distance: 0 },
  { _id: 3, name: 'Store C', location: { type: 'Point', coordinates: [ -122.3,
 37.9 ] }, distance: 13877.82 },
  { _id: 2, name: 'Store B', location: { type: 'Point', coordinates: [ -122.5,
 37.7 ] }, distance: 15557.89 }
]
```

Code examples

To view a code example for using the `$geoNear` aggregation stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const collection = db.collection('stores');

  const result = await collection.aggregate([
    {
      $geoNear: {
        near: { type: "Point", coordinates: [-122.4, 37.8] },
        distanceField: "distance",
        spherical: true
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['stores']

    result = list(collection.aggregate([
        {
            '$geoNear': {
                'near': { 'type': 'Point', 'coordinates': [-122.4, 37.8] },
                'distanceField': 'distance',
                'spherical': True
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$graphLookup

New from version 8.0.2.

The `$graphLookup` aggregation stage in Amazon DocumentDB performs a recursive search on a collection. For each input document, `$graphLookup` starts from one or more values and repeatedly matches a connecting field to a target field, following the chain of matching documents until there are no more matches or an optional recursion depth is reached. This is useful for traversing graph-like or hierarchical data, such as organizational reporting structures, category trees, or network relationships.

Parameters

- `from`: The target collection to search recursively. The `from` collection must be in the same database.
- `startWith`: The expression that specifies the value or values to begin the recursive search with. This value is compared against the `connectToField` of the documents in the `from` collection. If `startWith` evaluates to an array, the search runs from all array elements at the same time.
- `connectFromField`: The field name whose value is used to connect to the next document in the recursive search. If the value is an array, each element is followed individually through the traversal.
- `connectToField`: The field name in the `from` collection that is matched against the value of the `connectFromField`.
- `as`: The name of the array field added to each output document. This field contains the documents found during the recursive search. For information about the order of these documents, see the Result ordering item in the following Behavior section.
- `maxDepth`: A non-negative integer that specifies the maximum number of recursive iterations to perform after the initial lookup. This parameter is optional. A value of `0` performs only the initial match and no recursion. If you omit `maxDepth`, the search continues recursively until there are no more matching documents.
- `depthField`: The name of the field to add to each document in the `as` array that holds the recursion depth at which the document was found. This parameter is optional. The depth is represented as a `NumberLong` and starts at zero, so the initial lookup corresponds to a depth of `0`.

- `restrictSearchWithMatch`: A document that specifies an additional match condition applied to the documents of the search. This parameter is optional. The condition uses the same query syntax as a `$match` stage and cannot include aggregation expressions.

> **Note**
>
> The `$graphLookup` stage is subject to a 100 MB memory limit and doesn't support `allowDiskUse:true`. An operation that exceeds this limit returns an error. If the aggregation includes other stages, the `allowDiskUse:true` option remains in effect for those stages.

Behavior

- **Cross-collection traversal**: Set `from` to the collection you want to search. This can be the same collection the pipeline runs against or a different collection in the same database.
- **Result ordering**: The documents in the `as` array are not returned in any guaranteed order.

Example (MongoDB Shell)

The following example uses `$graphLookup` to traverse an employee reporting hierarchy and, for each employee, return the full chain of managers above them.

The following commands, run in the mongosh shell, insert sample employee documents that establish a reporting hierarchy:

```
db.employees.insertMany([
  { _id: 1, name: "Dev", reportsTo: "Eliot" },
  { _id: 2, name: "Eliot", reportsTo: "Ron" },
  { _id: 3, name: "Ron", reportsTo: "Andrew" },
  { _id: 4, name: "Andrew", reportsTo: "Asya" },
  { _id: 5, name: "Asya" }
]);
```

The following aggregation returns the full management chain for each employee in a new `reportingHierarchy` array:

```
db.employees.aggregate([
```

```
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy",
      depthField: "level"
    }
  }
]);
```

This operation returns the following output:

```
[
  {
    _id: 1,
    name: 'Dev',
    reportsTo: 'Eliot',
    reportingHierarchy: [
      { _id: 2, name: 'Eliot', reportsTo: 'Ron', level: 0 },
      { _id: 3, name: 'Ron', reportsTo: 'Andrew', level: 1 },
      { _id: 4, name: 'Andrew', reportsTo: 'Asya', level: 2 },
      { _id: 5, name: 'Asya', level: 3 }
    ]
  },
  {
    _id: 2,
    name: 'Eliot',
    reportsTo: 'Ron',
    reportingHierarchy: [
      { _id: 3, name: 'Ron', reportsTo: 'Andrew', level: 0 },
      { _id: 4, name: 'Andrew', reportsTo: 'Asya', level: 1 },
      { _id: 5, name: 'Asya', level: 2 }
    ]
  },
  {
    _id: 3,
    name: 'Ron',
    reportsTo: 'Andrew',
    reportingHierarchy: [
      { _id: 4, name: 'Andrew', reportsTo: 'Asya', level: 0 },
      { _id: 5, name: 'Asya', level: 1 }
```

```
    ]
  },
  {
    _id: 4,
    name: 'Andrew',
    reportsTo: 'Asya',
    reportingHierarchy: [
      { _id: 5, name: 'Asya', level: 0 }
    ]
  },
  {
    _id: 5,
    name: 'Asya',
    reportingHierarchy: []
  }
]
```

Each employee document gains a `reportingHierarchy` array that contains the recursively matched manager documents. The `level` field records the recursion depth at which each manager was found. Asya has no `reportsTo` value, so her `reportingHierarchy` array is empty.

Example with maxDepth and restrictSearchWithMatch (MongoDB Shell)

The following example traverses a network of connections between people and uses `maxDepth` to limit the recursion and `restrictSearchWithMatch` to include only documents that match an additional condition. The search finds the connections of a starting person, up to two levels deep, keeping only people whose `hobbies` array contains `golf`.

The following commands, run in the mongosh shell, insert sample documents in which each person's `friends` point forward through a one-directional chain of connections:

```
db.people.insertMany([
  { _id: 1, name: "Tanya", friends: [ "Carole", "Shirley" ], hobbies: [ "tennis",
 "golf" ] },
  { _id: 2, name: "Carole", friends: [ "Joseph" ], hobbies: [ "archery", "golf" ] },
  { _id: 3, name: "Joseph", friends: [ "Angelo" ], hobbies: [ "tennis", "golf" ] },
  { _id: 4, name: "Angelo", friends: [ "Miguel" ], hobbies: [ "travel", "golf" ] },
  { _id: 5, name: "Miguel", friends: [], hobbies: [ "golf" ] },
  { _id: 6, name: "Shirley", friends: [], hobbies: [ "frisbee" ] }
]);
```

The following aggregation starts from Tanya, traverses her network of connections up to two levels deep, and keeps only the people who list `golf` as a hobby:

```
db.people.aggregate([
  { $match: { name: "Tanya" } },
  {
    $graphLookup: {
      from: "people",
      startWith: "$friends",
      connectFromField: "friends",
      connectToField: "name",
      as: "golfConnections",
      maxDepth: 2,
      restrictSearchWithMatch: { hobbies: "golf" }
    }
  },
  { $project: { name: 1, "golfConnections.name": 1 } }
]);
```

This operation returns the following output:

```
[
  {
    _id: 1,
    name: 'Tanya',
    golfConnections: [
      { name: 'Carole' },
      { name: 'Joseph' },
      { name: 'Angelo' }
    ]
  }
]
```

Only connections whose `hobbies` array contains `golf` are included, and the traversal stops after two levels of recursion. Shirley is excluded because she does not list `golf`, and Miguel is excluded because he is three levels away from Tanya, beyond the `maxDepth` of 2. Because the connections form a one-directional chain that never points back to Tanya, the starting document is not matched during the recursive search.

Code examples

To view a code example that builds the employee reporting hierarchy from the first example using a MongoDB driver, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const employees = db.collection('employees');

    const result = await employees.aggregate([
      {
        $graphLookup: {
          from: 'employees',
          startWith: '$reportsTo',
          connectFromField: 'reportsTo',
          connectToField: 'name',
          as: 'reportingHierarchy',
          depthField: 'level'
        }
      }
    ]).toArray();

    console.log(JSON.stringify(result, null, 2));
  } catch (err) {
    console.error('Failed to run $graphLookup aggregation:', err.message);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pymongo.errors import PyMongoError

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        employees = db['employees']

        pipeline = [
            {
                "$graphLookup": {
                    "from": "employees",
                    "startWith": "$reportsTo",
                    "connectFromField": "reportsTo",
                    "connectToField": "name",
                    "as": "reportingHierarchy",
                    "depthField": "level"
                }
            }
        ]

        result = list(employees.aggregate(pipeline))

        for doc in result:
            print(doc)
    except PyMongoError as error:
        print(f"Failed to run $graphLookup aggregation: {error}")
    finally:
        client.close()

example()
```

$group

The $group aggregation stage in Amazon DocumentDB allows you to group documents by a specified expression and perform various accumulative operations on the grouped data. This can be useful for tasks such as calculating totals, averages, or other statistics based on the grouped data.

Parameters

- _id: Specifies the expression by which the input documents should be grouped. This can be a field name, a computed expression, or a combination of both.
- accumulator expressions: (optional) One or more accumulator expressions that should be applied to the grouped data. These expressions use the accumulator operators mentioned above.

Example (MongoDB Shell)

The following example groups customers by their city and calculates the total order amount for each city.

Create sample documents

```
db.customers.insertMany([
  { name: "John Doe", city: "New York", orders: [{ amount: 100 }, { amount: 200 }] },
  { name: "Jane Smith", city: "Los Angeles", orders: [{ amount: 150 }, { amount:
 300 }] },
  { name: "Bob Johnson", city: "New York", orders: [{ amount: 75 }, { amount: 125 }] },
  { name: "Samantha Lee", city: "Chicago", orders: [{ amount: 50 }, { amount: 100 }] }
]);
```

Query example

```
db.customers.aggregate([
  {
    $group: {
      _id: "$city",
      totalOrders: { $sum: { $sum: "$orders.amount" } }
    }
  }
]);
```

Output

```
[
  { _id: 'Chicago', totalOrders: 150 },
  { _id: 'Los Angeles', totalOrders: 450 },
  { _id: 'New York', totalOrders: 500 }
]
```

Code examples

To view a code example for using the $group command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function groupByCity() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');
    const result = await db.collection('customers').aggregate([
      { $unwind: '$orders' },
      {
        $group: {
          _id: '$city',
          totalOrders: { $sum: '$orders.amount' }
        }
      }
    ]).toArray();

    console.log(result);
  } catch (err) {
    console.error('Error during aggregation:', err);
  } finally {
    await client.close();
  }
}
```

```
groupByCity();
```

Python

```
from pymongo import MongoClient

def group_by_city():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client.test
        result = list(db.customers.aggregate([
            {'$unwind': '$orders'},
            {
                '$group': {
                    '_id': '$city',
                    'totalOrders': {'$sum': '$orders.amount'}
                }
            }
        ]))
        print(result)
    except Exception as e:
        print(f"Error during aggregation: {e}")
    finally:
        client.close()

group_by_city()
```

$gt

The `$gt` aggregation operator compares two values and returns `true` if the first value is greater than the second, otherwise returns `false`.

Parameters

- `expression1`: The first value to compare.
- `expression2`: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the $gt operator to identify products exceeding a price threshold.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Laptop", price: 1200 },
  { _id: 2, name: "Mouse", price: 25 },
  { _id: 3, name: "Monitor", price: 400 }
]);
```

Query example

```
db.products.aggregate([
  {
    $project: {
      name: 1,
      price: 1,
      expensive: { $gt: ["$price", 100] }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Laptop', price: 1200, expensive: true },
  { _id: 2, name: 'Mouse', price: 25, expensive: false },
  { _id: 3, name: 'Monitor', price: 400, expensive: true }
]
```

Code examples

To view a code example for using the $gt aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        price: 1,
        expensive: { $gt: ["$price", 100] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'price': 1,
                'expensive': { '$gt': ['$price', 100] }
            }
        }
```

```
    ]))

    print(result)
    client.close()

example()
```

$gte

The $gte aggregation operator compares two values and returns true if the first value is greater than or equal to the second, otherwise returns false.

Parameters

- expression1: The first value to compare.
- expression2: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the $gte operator to check if students passed an exam.

Create sample documents

```
db.students.insertMany([
  { _id: 1, name: "Alice", score: 85 },
  { _id: 2, name: "Bob", score: 60 },
  { _id: 3, name: "Charlie", score: 72 }
]);
```

Query example

```
db.students.aggregate([
  {
    $project: {
      name: 1,
      score: 1,
      passed: { $gte: ["$score", 70] }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Alice', score: 85, passed: true },
  { _id: 2, name: 'Bob', score: 60, passed: false },
  { _id: 3, name: 'Charlie', score: 72, passed: true }
]
```

Code examples

To view a code example for using the $gte aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('students');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        score: 1,
        passed: { $gte: ["$score", 70] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['students']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'score': 1,
                'passed': { '$gte': ['$score', 70] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$hour

The $hour operator extracts the hour component from a date or timestamp field.

Parameters

- dateExpression: The date to which the operator is applied. This must resolve to a valid BSON date (e.g., a field like $createdAt or a date literal).

A parameter can also be specified as a document in the following format:

{ date: <dateExpression>, timezone: <timezoneExpression> }

This allows to apply timezone-aware date operations.

```
- `<tzExpression>`: (optional) The timezone of the operation result. It must be a valid
 expression that resolves to a string formatted as either an Olson Timezone Identifier
 or a UTC Offset. If no timezone is provided, the result is in UTC.
```

Example (MongoDB Shell)

The following example demonstrates how to use the $hour operator to extract the hour component from a date field and group the data accordingly.

Create sample documents

```
db.events.insertMany([
  { timestamp: new Date("2023-04-01T10:30:00Z") },
  { timestamp: new Date("2023-04-01T12:45:00Z") },
  { timestamp: new Date("2023-04-02T08:15:00Z") },
  { timestamp: new Date("2023-04-02T16:20:00Z") },
  { timestamp: new Date("2023-04-03T23:59:00Z") }
]);
```

Query example

```
db.events.aggregate([
  {
    $project: {
      hour: { $hour: "$timestamp" }
    }
  },
  {
    $group: {
      _id: "$hour",
      count: { $sum: 1 }
    }
  },
  {
    $sort: { _id: 1 }
  }
]);
```

Output

```
[
  { "_id": 8, "count": 1 },
  { "_id": 10, "count": 1 },
  { "_id": 12, "count": 1 },
  { "_id": 16, "count": 1 },
  { "_id": 23, "count": 1 }
```

```
]
```

This query groups the events by the hour component of the `timestamp` field and counts the number of events for each hour.

Code examples

To view a code example for using the `$hour` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('events');

    const result = await collection.aggregate([
      {
        $project: {
          hour: { $hour: "$timestamp" }
        }
      },
      {
        $group: {
          _id: "$hour",
          count: { $sum: 1 }
        }
      },
      {
        $sort: { _id: 1 }
      }
    ]).toArray();

    console.log(result);
  } catch (error) {
```

```
    console.error('Error occurred:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from datetime import datetime

def example():
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client.test
        collection = db.events

        result = list(collection.aggregate([
            {
                "$project": {
                    "hour": {"$hour": "$timestamp"}
                }
            },
            {
                "$group": {
                    "_id": "$hour",
                    "count": {"$sum": 1}
                }
            },
            {
                "$sort": {"_id": 1}
            }
        ]))

        print(result)

    except Exception as e:
        print(f"An error occurred: {e}")
```

```
    finally:
        client.close()

example()
```

$ifNull

The `$ifNull` operator is used to return a specified value if the input expression evaluates to null or undefined. This operator can be useful in scenarios where you want to provide a default value or handle null/undefined cases.

Parameters

- `expression`: The expression to evaluate.
- `replacement`: The value to return if the <`expression`> evaluates to null or undefined.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$ifNull` operator to provide a default value when the `name` field is null or undefined.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "John" },
  { _id: 2, name: null },
  { _id: 3 }
]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      _id: 1,
      name: { $ifNull: ["$name", "No Name"] }
    }
  }
```

```
]);
```

Output

```
[
  { "_id": 1, "name": "John" },
  { "_id": 2, "name": "No Name" },
  { "_id": 3, "name": "No Name" }
]
```

Code examples

To view a code example for using the `$ifNull` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');

  const collection = db.collection('users');

  const pipeline = [
    {
      $project: {
        _id: 1,
        name: { $ifNull: ["$name", "No Name"] }
      }
    }
  ];

  const cursor = await collection.aggregate(pipeline);

  await cursor.forEach(doc => {
    console.log(doc);
  });
```

```
  await client.close();

}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client.test
    collection = db.users

    pipeline = [
        {
            "$project": {
                "_id": 1,
                "name": { "$ifNull": ["$name", "No Name"] }
            }
        }
    ]

    result = collection.aggregate(pipeline)

    for doc in result:
        print(doc)

    client.close()

example()
```

$in

The `$in` aggregation operator checks if a specified value exists within an array. It returns `true` if the value is found in the array, and `false` otherwise.

Parameters

- `value`: The value to search for.
- `array`: The array to search within.

Example (MongoDB Shell)

The following example demonstrates using the `$in` operator to check if a specific skill exists in each employee's skill set.

Create sample documents

```
db.employees.insertMany([
  { _id: 1, name: "Sarah", skills: ["Python", "JavaScript", "SQL"] },
  { _id: 2, name: "Mike", skills: ["Java", "C++", "Go"] },
  { _id: 3, name: "Emma", skills: ["Python", "Ruby", "Rust"] }
]);
```

Query example

```
db.employees.aggregate([
  {
    $project: {
      name: 1,
      hasPython: { $in: ["Python", "$skills"] }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Sarah', hasPython: true },
  { _id: 2, name: 'Mike', hasPython: false },
  { _id: 3, name: 'Emma', hasPython: true }
]
```

Code examples

To view a code example for using the `$in` aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('employees');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        hasPython: { $in: ["Python", "$skills"] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['employees']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'hasPython': { '$in': ['Python', '$skills'] }
            }
```

```
        }
    ]))

    print(result)
    client.close()

example()
```

$indexOfArray

The `$indexOfArray` operator in Amazon DocumentDB is used to find the index of the first occurrence of a specified element in an array. This operator returns the zero-based index position of the first element in the array that matches the specified value. If the value is not found, it returns -1.

Parameters

- `array`: The array to search.
- `value`: The value to search for in the array.
- `start`: (optional) The position in the array to start the search from. The default value is 0.

Example (MongoDB Shell)

The following example demonstrates how to use the $indexOfArray operator to find the index of the first occurrence of the element "mango" in the "fruits" array for each document.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, fruits: ["apple", "banana", "cherry", "durian"] },
  { _id: 2, fruits: ["mango", "orange", "pineapple"] },
  { _id: 3, fruits: ["kiwi", "lemon", "mango"] }
]);
```

Query example

```
db.collection.aggregate([
  {
```

```
    $project: {
      _id: 1,
      fruitIndex: { $indexOfArray: ["$fruits", "mango"] }
    }
  }
]);
```

Output

```
{ "_id" : 1, "fruitIndex" : 1 }
{ "_id" : 2, "fruitIndex" : 0 }
{ "_id" : 3, "fruitIndex" : 2 }
```

Code examples

To view a code example for using the `$indexOfArray` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        fruitIndex: { $indexOfArray: ["$fruits", "mango"] }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['collection']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'fruitIndex': { '$indexOfArray': ["$fruits", "mango"] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$indexOfBytes

The $indexOfBytes operator in Amazon DocumentDB is used to find the starting index of a substring within a string, based on the byte positions of the characters. This can be useful when working with text data that may contain multi-byte characters, such as those found in non-Latin scripts.

Parameters

- `string`: The input string to search.
- `substring`: The substring to search for within the input string.
- `[<start>]`: (optional) The starting position (zero-based) of the search. If not specified, the search starts at the beginning of the string.

Example (MongoDB Shell)

The following example demonstrates the use of `$indexOfBytes` to find the index of the first hyphen character in a set of strings representing desk locations.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-BVV-021" },
  { "_id": 2, "Desk": "Munich-HGG-32a" },
  { "_id": 3, "Desk": "Cologne-ayu-892.50" },
  { "_id": 4, "Desk": "Dortmund-Hop-78" }
]);
```

Query example

```
db.people.aggregate([
  { $project: { stateLocation: { $indexOfBytes: ["$Desk", "-"] } } }
]);
```

Output

```
{ "_id" : 1, "stateLocation" : 11 }
{ "_id" : 2, "stateLocation" : 6 }
{ "_id" : 3, "stateLocation" : 7 }
{ "_id" : 4, "stateLocation" : 8 }
```

Code examples

To view a code example for using the `$indexOfBytes` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const events = db.collection('people');

  const result = await db.collection('people').aggregate([
    { $project: { stateLocation: { $indexOfBytes: ["$Desk", "-"] } } }
  ]).toArray();
  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['people']

    result = list(db.people.aggregate([
        { '$project': { 'stateLocation': { '$indexOfBytes': ['$Desk', '-'] } } }
    ]))
    print(result)
    client.close()

example()
```

$indexOfCP

The `$indexOfCP` operator in Amazon DocumentDB is used to find the index, in code points (CP), of the first occurrence of a specified substring within a string expression. This can be useful when parsing and extracting content from string fields.

Parameters

- `string expression`: The string to search.
- `substring`: The substring to search for.
- `[<start>]`: (optional) The position to start the search (zero-based index). Default is 0.

Example (MongoDB Shell)

In this example, we use the `$indexOfCP` operator to find the index of the first occurrence of the hyphen character - in the Desk field of each document.

Create sample documents

```
db.people.insertMany([
  { "_id":1, "name":"John Doe", "Manager":"Jane Doe", "Role":"Developer", "Desk":
 "Düsseldorf-BVV-021"},
  { "_id":2, "name":"John Stiles", "Manager":"Jane Doe", "Role":"Manager", "Desk":
 "Munich-HGG-32a"},
  { "_id":3, "name":"Richard Roe", "Manager":"Jorge Souza", "Role":"Product", "Desk":
 "Cologne-ayu-892.50"},
  { "_id":4, "name":"Mary Major", "Manager":"Jane Doe", "Role":"Solution Architect",
 "Desk": "Dortmund-Hop-78"}
])
```

Query example

```
db.people.aggregate([
  { $project: { stateLocation: { $indexOfCP: [ "$Desk", "-"] } } }
])
```

Output

```
{ "_id" : 1, "stateLocation" : 10 }
{ "_id" : 2, "stateLocation" : 6 }
{ "_id" : 3, "stateLocation" : 7 }
{ "_id" : 4, "stateLocation" : 8 }
```

Code examples

To view a code example for using the `$indexOfCP` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    { $project: { stateLocation: { $indexOfCP: [ "$Desk", "-"] } } }
  ]).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['people']

result = list(collection.aggregate([
    { '$project': { 'stateLocation': { '$indexOfCP': [ '$Desk', '-' ] } } }
]))

print(result)

client.close()
```

$indexStats

The `$indexStats` aggregation stage in Amazon DocumentDB provides insight into the usage of indexes within a collection. This operator allows you to monitor the access patterns of your indexes, which can help you make informed decisions about index management and optimization.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the `$indexStats` operator to analyze the index usage in an Amazon DocumentDB collection.

Create sample documents

```
db.grocery.insertMany([
  { _id: 1, product: "milk", quantity: 10 },
  { _id: 2, product: "eggs", quantity: 20 },
  { _id: 3, product: "bread", quantity: 5 },
  { _id: 4, product: "cheese", quantity: 15 },
  { _id: 5, product: "apple", quantity: 8 }
]);
```

Query example

```
db.grocery.aggregate([
  { $indexStats: {} }
]);
```

Output

```
[
  {
    "name": "_id_",
    "key": {
      "_id": 1
    },
    "host": "docdb-cluster-1.cluster-123456789.us-west-2.docdb.amazonaws.com",
    "accesses": {
      "ops": NumberLong(5),
      "since": ISODate("2023-04-06T12:34:56.789Z")
    }
  },
  {
    "name": "product_1",
    "key": {
      "product": 1
    },
    "host": "docdb-cluster-1.cluster-123456789.us-west-2.docdb.amazonaws.com",
```

```
    "accesses": {
      "ops": NumberLong(10),
      "since": ISODate("2023-04-06T12:34:56.789Z")
    }
  }
]
```

In this example, the `$indexStats` operator shows that the `_id_` index has been accessed 5 times, and the `product_1` index has been accessed 10 times since the last reset or server reboot.

Code examples

To view a code example for using the `$indexStats` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function indexStats() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const result = await db.collection('grocery').aggregate([
    { $indexStats: {} }
  ]).toArray();
  console.log(result);
  await client.close();
}

indexStats();
```

Python

```
from pymongo import MongoClient

def index_stats():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    result = list(db.grocery.aggregate([
```

```
        { '$indexStats': {} }
    ]))
    print(result)
    client.close()

index_stats()
```

$isArray

The `$isArray` operator in Amazon DocumentDB is used to check if a field in a document is an array. This operator can be useful in aggregation pipelines and conditional expressions to handle array-type fields.

Parameters

- `field`: The field path to check if it is an array.

Example (MongoDB Shell)

This example demonstrates how to use the `$isArray` operator to identify documents where the "inventory" field is an array.

Create sample documents

```
db.videos.insertMany([
  { "_id":1, "name":"Live Soft", "inventory": {"Des Moines": 1000, "Ames" : 500}},
  { "_id":2, "name":"Top Pilot", "inventory": {"Mason City": 250, "Des Moines": 1000}},
  { "_id":3, "name":"Romancing the Rock", "inventory": {"Mason City": 250, "Ames" :
 500}},
  { "_id":4, "name":"Bravemind", "inventory": [{"location": "Mason City", "count":
 250}, {"location": "Des Moines", "count": 1000}, {"location": "Ames", "count": 500}]}
]);
```

Query example

```
db.videos.aggregate([
  {
    $match: {
      $isArray: "$inventory"
    }
  },
```

```
  {
    $project: {
      _id: 1,
      name: 1,
      "inventory.location": 1,
      "inventory.count": 1
    }
  }
]).pretty();
```

Output

```
{
  "_id": 4,
  "name": "Bravemind",
  "inventory": [
    {
      "location": "Mason City",
      "count": 250
    },
    {
      "location": "Des Moines",
      "count": 1000
    },
    {
      "location": "Ames",
      "count": 500
    }
  ]
}
```

Code examples

To view a code example for using the `$isArray` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function run() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('videos');

  const result = await collection.aggregate([
    {
      $match: {
        $isArray: '$inventory'
      }
    },
    {
      $project: {
        _id: 1,
        name: 1,
        "inventory.location": 1,
        "inventory.count": 1
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

run();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['videos']

result = list(collection.aggregate([
    {
        '$match': {
            '$isArray': '$inventory'
```

```
        }
    },
    {
        '$project': {
            '_id': 1,
            'name': 1,
            'inventory.location': 1,
            'inventory.count': 1
        }
    }
]))

print(result)

client.close()
```

$isNumber

New from version 8.0.1.

The `$isNumber` operator in Amazon DocumentDB returns a boolean indicating whether a given expression resolves to a numeric type (integer, decimal, double, long). Note that arrays are not considered numeric, even if all elements in the array are numbers. For example, `$isNumber` applied to the value `[1, 2, 3]` returns `false`.

Parameters

- `expression`: Any expression to check whether it resolves to a numeric type.

Example (MongoDB Shell)

The following example shows how to use the `$isNumber` operator to check whether field values are numeric.

Create sample documents

```
db.data.insertMany([
  {_id: 1, value: 42},
  {_id: 2, value: "hello"},
  {_id: 3, value: 3.14},
  {_id: 4, value: null}
```

```
]);
```

Query example

```
db.data.aggregate([
  { $project: { isNum: { $isNumber: "$value" } } }
]);
```

Output

```
[
  {_id: 1, isNum: true},
  {_id: 2, isNum: false},
  {_id: 3, isNum: true},
  {_id: 4, isNum: false}
]
```

Code examples

To view a code example for using the `$isNumber` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('data');
    const result = await collection.aggregate([
      { $project: { isNum: { $isNumber: "$value" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['data']
        result = list(collection.aggregate([
            {'$project': {'isNum': {'$isNumber': '$value'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$isoDayOfWeek

The `$isoDayOfWeek` operator in Amazon DocumentDB returns the ISO day of the week for a date as an integer value. The ISO week date system defines each week starting on a Monday and ending on a Sunday, with week 1 being the week that contains the year's first Thursday.

Parameters

- `expression`: The date expression for which to return the ISO day of the week.

Example (MongoDB Shell)

The following example demonstrates how to use the `$isoDayOfWeek` operator to retrieve the ISO day of the week for a set of event documents.

Create sample documents

```
db.events.insertMany([
```

```
  { _id: 1, eventDate: ISODate("2023-04-01T12:00:00Z") },
  { _id: 2, eventDate: ISODate("2023-04-02T12:00:00Z") },
  { _id: 3, eventDate: ISODate("2023-04-03T12:00:00Z") },
  { _id: 4, eventDate: ISODate("2023-04-04T12:00:00Z") },
  { _id: 5, eventDate: ISODate("2023-04-05T12:00:00Z") },
  { _id: 6, eventDate: ISODate("2023-04-06T12:00:00Z") },
  { _id: 7, eventDate: ISODate("2023-04-07T12:00:00Z") }
]);
```

Query example

```
db.events.aggregate([
  { $project: {
    _id: 1,
    eventDate: 1,
    isoDayOfWeek: { $isoDayOfWeek: "$eventDate" }
  }}
]);
```

Output

```
[
  { "_id": 1, "eventDate": ISODate("2023-04-01T12:00:00Z"), "isoDayOfWeek": 6 },
  { "_id": 2, "eventDate": ISODate("2023-04-02T12:00:00Z"), "isoDayOfWeek": 7 },
  { "_id": 3, "eventDate": ISODate("2023-04-03T12:00:00Z"), "isoDayOfWeek": 1 },
  { "_id": 4, "eventDate": ISODate("2023-04-04T12:00:00Z"), "isoDayOfWeek": 2 },
  { "_id": 5, "eventDate": ISODate("2023-04-05T12:00:00Z"), "isoDayOfWeek": 3 },
  { "_id": 6, "eventDate": ISODate("2023-04-06T12:00:00Z"), "isoDayOfWeek": 4 },
  { "_id": 7, "eventDate": ISODate("2023-04-07T12:00:00Z"), "isoDayOfWeek": 5 }
]
```

Code examples

To view a code example for using the `$isoDayOfWeek` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    {
      $project: {
        _id: 1,
        eventDate: 1,
        isoDayOfWeek: { $isoDayOfWeek: '$eventDate' }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    events = db.events

    result = list(events.aggregate([
        {
            '$project': {
                '_id': 1,
                'eventDate': 1,
                'isoDayOfWeek': { '$isoDayOfWeek': '$eventDate' }
            }
        }
    ]))
```

```
    print(result)
    client.close()

example()
```

$isoWeek

The `$isoWeek` operator in Amazon DocumentDB returns the ISO week number for a date. The ISO week date system is a way of numbering weeks in a year, in which the first week of a new year is the week that contains the first Thursday of that year. This is different from the Gregorian calendar, where the first week of a new year is the week that contains January 1.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the `$isoWeek` operator to retrieve the ISO week number for a given date.

Create sample documents

```
db.dates.insertMany([
  { _id: 1, date: new ISODate("2022-01-01") },
  { _id: 2, date: new ISODate("2022-12-31") },
  { _id: 3, date: new ISODate("2023-01-01") }
])
```

Query example

```
db.dates.aggregate([
  {
    $project: {
      _id: 1,
      isoWeek: { $isoWeek: "$date" }
    }
  }
])
```

Output

```
[
  { "_id": 1, "isoWeek": 52 },
  { "_id": 2, "isoWeek": 52 },
  { "_id": 3, "isoWeek": 1 }
]
```

Code examples

To view a code example for using the `$isoWeek` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('dates');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        isoWeek: { $isoWeek: "$date" }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['dates']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'isoWeek': { '$isoWeek': '$date' }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$isoWeekYear

The `$isoWeekYear` operator in Amazon DocumentDB returns the ISO 8601 week year number for a given date. The ISO week year number differs from the Gregorian calendar year in that the week year can be different from the calendar year, especially at the start and end of the year.

Parameters

- `expression`: The date expression for which to return the ISO 8601 week year number.

Example (MongoDB Shell)

This example demonstrates how to use the `$isoWeekYear` operator to retrieve the ISO 8601 week year for the date field of each document in the events collection.

Create sample documents

```
db.events.insertMany([
  { _id: 1, name: "Event 1", date: ISODate("2022-12-31T00:00:00Z") },
  { _id: 2, name: "Event 2", date: ISODate("2023-01-01T00:00:00Z") },
  { _id: 3, name: "Event 3", date: ISODate("2023-01-02T00:00:00Z") }
```

```
]);
```

Query example

```
db.events.aggregate([
  { $project: {
    name: 1,
    isoWeekYear: { $isoWeekYear: "$date" }
  }}
]);
```

Output

```
[
  { "_id" : 1, "name" : "Event 1", "isoWeekYear" : 2023 },
  { "_id" : 2, "name" : "Event 2", "isoWeekYear" : 2023 },
  { "_id" : 3, "name" : "Event 3", "isoWeekYear" : 2023 }
]
```

Code examples

To view a code example for using the `$isoWeekYear` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    { $project: {
      name: 1,
      isoWeekYear: { $isoWeekYear: '$date' }
    }}
  ]).toArray();
```

```
  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    events = db.events

    result = list(events.aggregate([
        { '$project': {
            'name': 1,
            'isoWeekYear': { '$isoWeekYear': '$date' }
        }}
    ]))

    print(result)
    client.close()

example()
```

$last

The `$last` operator in Amazon DocumentDB is used to return the last element in an array that matches the query criteria. It is particularly useful for retrieving the most recent or the last element in an array that satisfies a specific condition.

Parameters

- `expression`: The expression to match the array elements.

Example (MongoDB Shell)

The following example demonstrates the use of the `$last` operator in combination with `$filter` to retrieve the last element from an array that meets a specific condition (e.g., subject is 'science').

Create sample documents

```
db.collection.insertMany([
  {
    "_id": 1,
    "name": "John",
    "scores": [
      { "subject": "math", "score": 82 },
      { "subject": "english", "score": 85 },
      { "subject": "science", "score": 90 }
    ]
  },
  {
    "_id": 2,
    "name": "Jane",
    "scores": [
      { "subject": "math", "score": 92 },
      { "subject": "english", "score": 88 },
      { "subject": "science", "score": 87 }
    ]
  },
  {
    "_id": 3,
    "name": "Bob",
    "scores": [
      { "subject": "math", "score": 75 },
      { "subject": "english", "score": 80 },
      { "subject": "science", "score": 85 }
    ]
  }
]);
```

Query example

```
db.collection.aggregate([
    { $match: { name: "John" } },
    {
      $project: {
```

```
        name: 1,
        lastScienceScore: {
          $last: {
            $filter: {
              input: "$scores",
              as: "score",
              cond: { $eq: ["$$score.subject", "science"] }
            }
          }
        }
      }
    }
  ]);
```

Output

```
[
  {
    _id: 1,
    name: 'John',
    lastScienceScore: { subject: 'science', score: 90 }
  }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$last` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the value of the `expression` from the last document in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
```

```
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the last (latest) price recorded in each partition.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        lastPrice: {
          $last: "$price",
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "lastPrice": 40 },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "lastPrice": 40 },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "lastPrice": 40 },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "lastPrice": 40 },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "lastPrice": 25 },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "lastPrice": 25 },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "lastPrice": 25 }
]
```

Each document is augmented with `lastPrice`, the price from the last document in its window (the latest `hour` in the partition).

Code examples

To view a code example for using the `$last` operator, choose the tab for the language that you want to use. The following examples show both expression usage (in `$project`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');

  // Expression usage: last element of a filtered array
  const collection = db.collection('collection');
  await collection.insertMany([
    {
      _id: 1,
      name: "John",
      scores: [
        { subject: "math", score: 82 },
        { subject: "english", score: 85 },
        { subject: "science", score: 90 }
      ]
    },
    {
      _id: 2,
      name: "Jane",
      scores: [
        { subject: "math", score: 92 },
        { subject: "english", score: 88 },
        { subject: "science", score: 87 }
      ]
    },
    {
      _id: 3,
      name: "Bob",
      scores: [
        { subject: "math", score: 75 },
        { subject: "english", score: 80 },
        { subject: "science", score: 85 }
      ]
    }
  ]);
  const expressionResult = await collection.aggregate([
    { $match: { name: "John" } },
```

```
  {
    $project: {
      name: 1,
      lastScienceScore: {
        $last: {
          $filter: {
            input: "$scores",
            as: "score",
            cond: { $eq: ["$$score.subject", "science"] }
          }
        }
      }
    }
  }
]).toArray();
console.log('Expression result:', JSON.stringify(expressionResult, null, 2));

// Window operator usage: last value within each partition
const stockPrices = db.collection('stockPrices');
await stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
const windowResult = await stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        lastPrice: {
          $last: "$price",
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]).toArray();
console.log('Window result:', JSON.stringify(windowResult, null, 2));
```

```
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():

    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client.test

    # Expression usage: last element of a filtered array
    collection = db.collection
    collection.insert_many([
        {
            "_id": 1,
            "name": "John",
            "scores": [
                { "subject": "math", "score": 82 },
                { "subject": "english", "score": 85 },
                { "subject": "science", "score": 90 }
            ]
        },
        {
            "_id": 2,
            "name": "Jane",
            "scores": [
                { "subject": "math", "score": 92 },
                { "subject": "english", "score": 88 },
                { "subject": "science", "score": 87 }
            ]
        },
        {
            "_id": 3,
            "name": "Bob",
            "scores": [
                { "subject": "math", "score": 75 },
```

```
            { "subject": "english", "score": 80 },
            { "subject": "science", "score": 85 }
        ]
    }
])
expression_pipeline = [
    { "$match": { "name": "John" } },
    {
        "$project": {
            "name": 1,
            "lastScienceScore": {
                "$last": {
                    "$filter": {
                        "input": "$scores",
                        "as": "score",
                        "cond": { "$eq": ["$$score.subject", "science"] }
                    }
                }
            }
        }
    }
]
expression_result = list(collection.aggregate(expression_pipeline))
print('Expression result:', expression_result)

# Window operator usage: last value within each partition
stock_prices = db.stockPrices
stock_prices.insert_many([
    { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50 },
    { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45 },
    { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60 },
    { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40 },
    { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30 },
    { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35 },
    { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25 }
])
window_pipeline = [
    {
        "$setWindowFields": {
            "partitionBy": "$ticker",
            "sortBy": { "hour": 1 },
            "output": {
                "lastPrice": {
                    "$last": "$price",
```

```
                        "window": { "documents": ["unbounded", "unbounded"] }
                    }
                }
            }
        }
    ]
    window_result = list(stock_prices.aggregate(window_pipeline))
    print('Window result:', window_result)

    client.close()

example()
```

$lastN

New from version 8.0.1.

The `$lastN` operator in Amazon DocumentDB returns the last N elements. When used as an accumulator in a `$group` stage, it returns an array of the last N values in each group. When used as an array expression operator, it returns the last N elements of an array.

Parameters

- `input`: The expression that resolves to the field or array from which to return values.
- n: A positive integer that specifies how many values to return. When used as an accumulator in a `$group` stage, n can also be an expression, as long as it resolves to a positive integer based on the group `_id` field.

Example (MongoDB Shell)

The following example shows how to use the `$lastN` accumulator to retrieve the last two quantities for each item during the aggregation.

> **Note**
>
> `$lastN` selects values in the order that documents reach the `$group` stage. To return the last N values for a specific ordering (for example, by date or score), add a `$sort` stage before `$group`.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, date: ISODate("2023-01-01") },
  { item: "abc", quantity: 5, date: ISODate("2023-01-02") },
  { item: "abc", quantity: 8, date: ISODate("2023-01-03") },
  { item: "xyz", quantity: 15, date: ISODate("2023-01-01") },
  { item: "xyz", quantity: 7, date: ISODate("2023-01-02") },
  { item: "xyz", quantity: 3, date: ISODate("2023-01-03") }
]);
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", lastTwoQuantities: { $lastN: { input: "$quantity", n:
 2 } } } }
]);
```

Output

```
[
  { "_id": "abc", "lastTwoQuantities": [5, 8] },
  { "_id": "xyz", "lastTwoQuantities": [7, 3] }
]
```

Expression usage example (MongoDB Shell)

The `$lastN` operator can also be used as an expression within a `$project` stage to return the last N elements of an array field.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "abc", tags: ["red", "green", "blue", "yellow", "purple"] },
  { _id: 2, item: "xyz", tags: ["alpha", "beta", "gamma"] }
]);
```

Query example

```
db.inventory.aggregate([
```

```
  { $project: {
      lastThreeTags: { $lastN: { input: "$tags", n: 3 } }
    }}
]);
```

Output

```
[
  { "_id": 1, "lastThreeTags": ["blue", "yellow", "purple"] },
  { "_id": 2, "lastThreeTags": ["alpha", "beta", "gamma"] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$lastN` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of up to the last n values for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$lastN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the two most recent prices seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        lastTwoSoFar: {
          $lastN: { input: "$price", n: 2 },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "lastTwoSoFar": [50] },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "lastTwoSoFar": [50, 45] },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "lastTwoSoFar": [45, 60] },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "lastTwoSoFar": [60, 40] },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "lastTwoSoFar": [30] },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "lastTwoSoFar": [30, 35] },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "lastTwoSoFar": [35, 25] }
]
```

Each document is augmented with `lastTwoSoFar`, the last two `price` values within its window (in `hour` order).

Code examples

To view a code example for using the `$lastN` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: last N values per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { item: "abc", quantity: 10, date: new Date("2023-01-01") },
      { item: "abc", quantity: 5, date: new Date("2023-01-02") },
      { item: "abc", quantity: 8, date: new Date("2023-01-03") },
      { item: "xyz", quantity: 15, date: new Date("2023-01-01") },
      { item: "xyz", quantity: 7, date: new Date("2023-01-02") },
      { item: "xyz", quantity: 3, date: new Date("2023-01-03") }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", lastTwoQuantities: { $lastN: { input: "$quantity",
 n: 2 } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Expression usage: last N elements of an array field
    const inventory = db.collection('inventory');
    await inventory.insertMany([
      { _id: 1, item: "abc", tags: ["red", "green", "blue", "yellow", "purple"] },
      { _id: 2, item: "xyz", tags: ["alpha", "beta", "gamma"] }
    ]);
    const expressionResult = await inventory.aggregate([
      { $project: { lastThreeTags: { $lastN: { input: "$tags", n: 3 } } } }
    ]).toArray();
    console.log('Expression result:', expressionResult);

    // Window operator usage: last N values over a window in each partition
    const stockPrices = db.collection('stockPrices');
    await stockPrices.insertMany([
      { _id: 1, ticker: "ABC", hour: 1, price: 50 },
```

```
      { _id: 2, ticker: "ABC", hour: 2, price: 45 },
      { _id: 3, ticker: "ABC", hour: 3, price: 60 },
      { _id: 4, ticker: "ABC", hour: 4, price: 40 },
      { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
      { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
      { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
    ]);
    const windowResult = await stockPrices.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$ticker",
          sortBy: { hour: 1 },
          output: {
            lastTwoSoFar: {
              $lastN: { input: "$price", n: 2 },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from datetime import datetime

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
```

```
        # Accumulator usage: last N values per group
        sales = db['sales']
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'date': datetime(2023, 1, 1) },
            { 'item': 'abc', 'quantity': 5, 'date': datetime(2023, 1, 2) },
            { 'item': 'abc', 'quantity': 8, 'date': datetime(2023, 1, 3) },
            { 'item': 'xyz', 'quantity': 15, 'date': datetime(2023, 1, 1) },
            { 'item': 'xyz', 'quantity': 7, 'date': datetime(2023, 1, 2) },
            { 'item': 'xyz', 'quantity': 3, 'date': datetime(2023, 1, 3) }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'lastTwoQuantities': { '$lastN':
{ 'input': '$quantity', 'n': 2 } } } }
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: last N elements of an array field
        inventory = db['inventory']
        inventory.insert_many([
            { '_id': 1, 'item': 'abc', 'tags': ['red', 'green', 'blue', 'yellow',
'purple'] },
            { '_id': 2, 'item': 'xyz', 'tags': ['alpha', 'beta', 'gamma'] }
        ])
        expression_result = list(inventory.aggregate([
            { '$project': { 'lastThreeTags': { '$lastN': { 'input': '$tags', 'n':
3 } } } }
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: last N values over a window in each partition
        stock_prices = db['stockPrices']
        stock_prices.insert_many([
            { '_id': 1, 'ticker': 'ABC', 'hour': 1, 'price': 50 },
            { '_id': 2, 'ticker': 'ABC', 'hour': 2, 'price': 45 },
            { '_id': 3, 'ticker': 'ABC', 'hour': 3, 'price': 60 },
            { '_id': 4, 'ticker': 'ABC', 'hour': 4, 'price': 40 },
            { '_id': 5, 'ticker': 'XYZ', 'hour': 1, 'price': 30 },
            { '_id': 6, 'ticker': 'XYZ', 'hour': 2, 'price': 35 },
            { '_id': 7, 'ticker': 'XYZ', 'hour': 3, 'price': 25 }
        ])
        window_result = list(stock_prices.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$ticker',
```

```
                    'sortBy': { 'hour': 1 },
                    'output': {
                        'lastTwoSoFar': {
                            '$lastN': { 'input': '$price', 'n': 2 },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$let

The `$let` operator in Amazon DocumentDB is used to bind variables to values and use those variables in the expression. It allows you to define local variables that can be used in subsequent expressions within the same stage of the aggregation pipeline.

Parameters

- `vars`: An object that defines the variables to be used in the expression.
- `in`: The expression in which the variables defined in the vars parameter are used.

Example (MongoDB Shell)

This example demonstrates the usage of the `$let` operator to calculate the area of a rectangle.

Create sample documents

```
db.shapes.insertMany([
  { name: "Rectangle 1", length: 5, width: 3 },
  { name: "Rectangle 2", length: 7, width: 4 },
  { name: "Rectangle 3", length: 6, width: 2 }
]);
```

Query example

```
db.shapes.aggregate([
  {
    $project: {
      name: 1,
      area: {
        $let: {
          vars: {
            length: "$length",
            width: "$width"
          },
          in: {
            $multiply: ["$$length", "$$width"]
          }
        }
      }
    }
  }
])
```

Output

```
[
  {
    "_id": ObjectId("6161e5b1a3eba3c7f2960d03"),
    "name": "Rectangle 1",
    "area": 15
  },
  {
    "_id": ObjectId("6161e5b1a3eba3c7f2960d04"),
    "name": "Rectangle 2",
    "area": 28
  },
  {
    "_id": ObjectId("6161e5b1a3eba3c7f2960d05"),
    "name": "Rectangle 3",
    "area": 12
  }
]
```

Code examples

To view a code example for using the `$let` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function calculateRectangleAreas() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('shapes');

  const result = await shapes.aggregate([
    {
      $project: {
        name: 1,
        area: {
          $let: {
            vars: {
              length: '$length',
              width: '$width'
            },
            in: {
              $multiply: ['$$length', '$$width']
            }
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

calculateRectangleAreas();
```

Python

```
from pymongo import MongoClient

def calculate_rectangle_areas():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.shapes

    result = list(shapes.aggregate([
        {
            '$project': {
                'name': 1,
                'area': {
                    '$let': {
                        'vars': {
                            'length': '$length',
                            'width': '$width'
                        },
                        'in': {
                            '$multiply': ['$$length', '$$width']
                        }
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

calculate_rectangle_areas()
```

$limit

The `$limit` operator in Amazon DocumentDB is used to restrict the number of documents returned by a query. It is similar to the MongoDB `$limit` operator, but there are some specific considerations when using it with Amazon DocumentDB.

In Amazon DocumentDB, the `$limit` operator is useful for pagination, where you want to retrieve a subset of the total matching documents. It allows you to control the number of documents returned in each response, improving performance and reducing the amount of data transferred over the network.

Parameters

- `limit`: The maximum number of documents to return. This must be a non-negative integer value.

Example (MongoDB Shell)

The following example demonstrates how to use the `$limit` operator to return a maximum of one document that matches the given filter.

Create sample documents

```
db.test.insertMany([
  { "_id": 1, "star_rating": 4, "comments": "apple is red" },
  { "_id": 2, "star_rating": 5, "comments": "comfortable couch" },
  { "_id": 3, "star_rating": 3, "comments": "apples, oranges - healthy fruit" },
  { "_id": 4, "star_rating": 5, "comments": "this is a great couch" },
  { "_id": 5, "star_rating": 5, "comments": "interesting couch" }
]);
```

Query example

```
db.test.createIndex({ comments: "text" });

db.test.find({
  $and: [
    { star_rating: 5 },
    { $text: { $search: "couch" } }
  ]
}).limit(1);
```

Output

```
[ { _id: 2, star_rating: 5, comments: 'comfortable couch' } ]
```

Code examples

To view a code example for using the `$limit` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function limitExample() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
      await client.connect();

      const db = client.db('test');
      const collection = db.collection('test');

      await collection.createIndex({ comments: 'text' });

      const query = {
        $and: [
          { star_rating: 5 },
          { $text: { $search: "couch" } }
        ]
      };

      const result = await collection.find(query).limit(1).toArray();

      console.log(result);

    } catch (err) {
        console.error("Error:", err);
    } finally {
        await client.close();
    }

}

limitExample();
```

Python

```
from pymongo import MongoClient

def limit_example():
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']
        collection = db['test']

        collection.create_index([('comments', 'text')])

        query = {
            '$and': [
                {'star_rating': 5},
                {'$text': {'$search': 'couch'}}
            ]
        }

        result = collection.find(query).limit(1)

        for doc in result:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

limit_example()
```

$listSearchIndexes

New from version 8.0.1.

The `$listSearchIndexes` aggregation stage in Amazon DocumentDB returns information about existing search indexes on a collection. It must be the first stage in an aggregation pipeline.

Parameters

- `name`: (optional) The name of the search index to return information about. If omitted, all search indexes on the collection are returned.

Output fields

Each returned document contains the following fields:

- `name`: The name of the search index.
- `status`: The build state of the index. One of `READY` (built and valid), `BUILDING` (a concurrent build is in progress), or `FAILED` (the index build did not complete successfully).
- `queryable`: A boolean indicating whether the index can currently be used to serve queries. This is `false` for hidden indexes (which remain `READY`) and for `FAILED` indexes. A hidden index is the case where `status` is `READY` but `queryable` is `false`.
- `latestDefinitionVersion`: A document containing `version` (the index format version) and `createdAt` (the time the index was created).

Syntax

```
db.collection.aggregate([
  { $listSearchIndexes: {} }
])

// Or filter by name:
db.collection.aggregate([
  { $listSearchIndexes: { name: "mySearchIndex" } }
])
```

Example (MongoDB Shell)

The following example shows how to use the `$listSearchIndexes` stage to list all search indexes on a collection.

Query example

```
db.movies.aggregate([
  { $listSearchIndexes: {} }
```

```
]);
```

Output

```
[
  {
    "name": "default",
    "status": "READY",
    "queryable": true,
    "latestDefinitionVersion": {
      "version": 2,
      "createdAt": ISODate("2026-05-11T21:12:57.974Z")
    }
  }
]
```

A hidden index remains READY but reports `queryable: false`, because it cannot be used to serve queries while hidden:

```
[
  {
    "name": "myHiddenIndex",
    "status": "READY",
    "queryable": false,
    "latestDefinitionVersion": {
      "version": 2,
      "createdAt": ISODate("2026-05-11T21:12:57.974Z")
    }
  }
]
```

To filter by a specific index name:

```
db.movies.aggregate([
  { $listSearchIndexes: { name: "default" } }
]);
```

Code examples

To view a code example for using the `$listSearchIndexes` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('movies');

    // List all search indexes
    const allIndexes = await collection.aggregate([
      { $listSearchIndexes: {} }
    ]).toArray();
    console.log('All search indexes:', allIndexes);

    // List a specific search index by name
    const namedIndex = await collection.aggregate([
      { $listSearchIndexes: { name: "default" } }
    ]).toArray();
    console.log('Named index:', namedIndex);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
```

```
        db = client['test']
        collection = db['movies']

        # List all search indexes
        all_indexes = list(collection.aggregate([
            { '$listSearchIndexes': {} }
        ]))
        print('All search indexes:', all_indexes)

        # List a specific search index by name
        named_index = list(collection.aggregate([
            { '$listSearchIndexes': { 'name': 'default' } }
        ]))
        print('Named index:', named_index)

    finally:
        client.close()

example()
```

$literal

The `$literal` operator in Amazon DocumentDB is used to represent a literal value within an aggregation pipeline stage. It allows you to include a specific value, such as a number, string, or boolean, without interpreting it as a field reference or expression.

This operator is particularly useful when you need to include a literal value as part of a more complex aggregation pipeline, such as when building dynamic query filters or performing calculations.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates the use of the `$literal` operator to include a literal value in an aggregation pipeline. The `$literal` operator is used to include the value 18 as a literal value in the $gt expression. This allows the aggregation pipeline to compare the age field against the literal value 18 to determine if the person is an adult.

Create sample documents

```
db.collection.insertMany([
  { "name": "John Doe", "age": 30, "city": "New York" },
  { "name": "Jane Doe", "age": 25, "city": "Los Angeles" },
  { "name": "Bob Smith", "age": 35, "city": "Chicago" }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      name: 1,
      age: 1,
      city: 1,
      isAdult: { $gt: ["$age", { $literal: 18 }] }
    }
  }
]);
```

Output

```
[
  {
    "_id": ObjectId("601234567890abcdef012345"),
    "name": "John Doe",
    "age": 30,
    "city": "New York",
    "isAdult": true
  },
  {
    "_id": ObjectId("601234567890abcdef012346"),
    "name": "Jane Doe",
    "age": 25,
    "city": "Los Angeles",
    "isAdult": true
  },
  {
    "_id": ObjectId("601234567890abcdef012347"),
    "name": "Bob Smith",
    "age": 35,
    "city": "Chicago",
    "isAdult": true
```

```
  }
]
```

Code examples

To view a code example for using the `$literal` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        age: 1,
        city: 1,
        isAdult: { $gt: ["$age", { $literal: 18 }] }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.collection

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'age': 1,
                'city': 1,
                'isAdult': { '$gt': ["$age", { '$literal': 18 }] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$ln

New from version 4.0.

The `$ln` operator in Amazon DocumentDB calculates the natural logarithm (base e) of a specified number. It returns the logarithm of the number to the base e.

Parameters

- `expression`: The number for which the natural logarithm will be calculated.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$log` operator to calculate the natural logarithm of a number.

Create sample documents

```
db.numbers.insertMany([
```

```
  { _id: 1, value: 10 },
  { _id: 2, value: 100 },
  { _id: 3, value: 1000 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    _id: 1,
    naturalLog: { $ln: "$value" }
  }}
]);
```

Output

```
[
  { "_id" : 1, "naturalLog" : 2.302585092994046 },
  { "_id" : 2, "naturalLog" : 4.605170185988092 },
  { "_id" : 3, "naturalLog" : 6.907755278982137 }
]
```

Code examples

To view a code example for using the `$ln` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
```

```
      _id: 1,
      naturalLog: { $ln: "$value" }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():

    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client.test
    collection = db.numbers

    result = list(collection.aggregate([
        { "$project": {
            "_id": 1,
            "naturalLog": { "$ln": "$value" }
        }}
    ]))

    print(result)
    client.close()

example()
```

$log

New from version 4.0.

The `$log` operator in Amazon DocumentDB calculates the natural logarithm of a number. It returns the base-e logarithm of the specified number.

Parameters

- `expression`: The number for which the natural logarithm will be calculated.
- `base`: Base value to calculate log.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$log` operator to calculate the natural logarithm of a number.

Create sample documents

```
db.numbers.insertMany([
  { _id: 1, value: 10 },
  { _id: 2, value: 100 },
  { _id: 3, value: 1000 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    _id: 1,
    naturalLog: { $log: ["$value", 10] }
  }}
]);
```

Output

```
[
  { "_id" : 1, "naturalLog" : 1 },
  { "_id" : 2, "naturalLog" : 2 },
  { "_id" : 3, "naturalLog" : 2.9999999999999996 }
]
```

Code examples

To view a code example for using the `$log` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      _id: 1,
      naturalLog: { $log: ["$value", 10] }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client.test
    collection = db.numbers

    result = list(collection.aggregate([
        { "$project": {
            "_id": 1,
            "naturalLog": { "$log": ["$value", 10] }
        }}
    ]))
```

```
    print(result)
    client.close()

example()
```

$log10

New from version 4.0.

The `$log10` operator in Amazon DocumentDB is used to calculate the base-10 logarithm of a number. It is useful for performing logarithmic calculations on numeric fields within the aggregation pipeline.

Parameters

- `expression`: The numeric expression for which the base-10 logarithm is to be calculated.

Example (MongoDB Shell)

The following example demonstrates how to use the `$log10` operator to calculate the base-10 logarithm of a numeric field.

Create sample documents

```
db.numbers.insertMany([
  { _id: 1, value: 1 },
  { _id: 2, value: 10 },
  { _id: 3, value: 100 },
  { _id: 4, value: 1000 }
]);
```

Query example

```
db.numbers.aggregate([
  {
    $project: {
      _id: 1,
      log10Value: { $log10: "$value" }
    }
```

```
  }
]);
```

Output

```
[
  { "_id": 1, "log10Value": 0 },
  { "_id": 2, "log10Value": 1 },
  { "_id": 3, "log10Value": 2 },
  { "_id": 4, "log10Value": 3 }
]
```

Code examples

To view a code example for using the `$log10` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  let client;

  try {
    client = await MongoClient.connect('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

    const db = client.db('test');
    const collection = db.collection('numbers');

    const result = await collection.aggregate([
      {
        $project: {
          _id: 1,
          log10Value: { $log10: "$value" }
        }
      }
    ]).toArray();

    console.log(result);
  } catch (error) {
```

```
    console.error("An error occurred:", error);
  } finally {
    if (client) {
      await client.close();
    }
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client.test
        collection = db.numbers

        result = list(collection.aggregate([
            {
                '$project': {
                    '_id': 1,
                    'log10Value': { '$log10': '$value' }
                }
            }
        ]))

        print(result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$lt

The `$lt` aggregation operator compares two values and returns `true` if the first value is less than the second, otherwise returns `false`.

Parameters

- `expression1`: The first value to compare.
- `expression2`: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the `$lt` operator to identify low stock items.

Create sample documents

```
db.warehouse.insertMany([
  { _id: 1, item: "Bolts", stock: 5 },
  { _id: 2, item: "Nuts", stock: 25 },
  { _id: 3, item: "Screws", stock: 8 }
]);
```

Query example

```
db.warehouse.aggregate([
  {
    $project: {
      item: 1,
      stock: 1,
      lowStock: { $lt: ["$stock", 10] }
    }
  }
]);
```

Output

```
[
  { _id: 1, item: 'Bolts', stock: 5, lowStock: true },
  { _id: 2, item: 'Nuts', stock: 25, lowStock: false },
  { _id: 3, item: 'Screws', stock: 8, lowStock: true }
]
```

Code examples

To view a code example for using the `$lt` aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('warehouse');

  const result = await collection.aggregate([
    {
      $project: {
        item: 1,
        stock: 1,
        lowStock: { $lt: ["$stock", 10] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['warehouse']
```

```
    result = list(collection.aggregate([
        {
            '$project': {
                'item': 1,
                'stock': 1,
                'lowStock': { '$lt': ['$stock', 10] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$lte

The `$lte` aggregation operator compares two values and returns `true` if the first value is less than or equal to the second, otherwise returns `false`.

Parameters

- `expression1`: The first value to compare.
- `expression2`: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the `$lte` operator to identify budget-friendly items.

Create sample documents

```
db.menu.insertMany([
  { _id: 1, dish: "Salad", price: 8 },
  { _id: 2, dish: "Pasta", price: 12 },
  { _id: 3, dish: "Soup", price: 6 }
]);
```

Query example

```
db.menu.aggregate([
```

```
  {
    $project: {
      dish: 1,
      price: 1,
      affordable: { $lte: ["$price", 10] }
    }
  }
]);
```

Output

```
[
  { _id: 1, dish: 'Salad', price: 8, affordable: true },
  { _id: 2, dish: 'Pasta', price: 12, affordable: false },
  { _id: 3, dish: 'Soup', price: 6, affordable: true }
]
```

Code examples

To view a code example for using the `$lte` aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('menu');

  const result = await collection.aggregate([
    {
      $project: {
        dish: 1,
        price: 1,
        affordable: { $lte: ["$price", 10] }
      }
    }
  ]).toArray();
```

```
  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['menu']

    result = list(collection.aggregate([
        {
            '$project': {
                'dish': 1,
                'price': 1,
                'affordable': { '$lte': ['$price', 10] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$lookup

The `$lookup` aggregation stage in Amazon DocumentDB allows you to perform a left outer join between two collections. This operation lets you combine data from multiple collections based on matching field values. It is particularly useful when you need to incorporate data from related collections into your query results.

Parameters

- `from`: The name of the collection to perform the join with.

- `localField`: The field from the input documents to match against the `foreignField`.
- `foreignField`: The field from the documents in the `from` collection to match against the `localField`.
- `as`: The name of the new field to add to the output documents containing the matching documents from the `from` collection.
- `pipeline`: Optional if `localField` and `foreignField` are specified. An aggregation pipeline to run on the `from` collection. The pipeline returns documents from the `from` collection. To return all documents, specify an empty pipeline `[]`. The pipeline cannot include the `$out` or `$merge` stages. See Join conditions and correlated subqueries (MongoDB Shell).
- `let`: Optional. A document that defines variables to use in the `pipeline` stages. Use the variable expressions to access the fields from the input documents that are joined to the `pipeline`. To reference a variable in the pipeline stages, use the "$$<variable>" syntax. See Join conditions and correlated subqueries (MongoDB Shell).

Example (MongoDB Shell)

The following example demonstrates a simple `$lookup` operation that joins data from the `orders` collection into the `customers` collection.

Create sample documents

```
db.customers.insertMany([
  { _id: 1, name: "Alice" },
  { _id: 2, name: "Bob" },
  { _id: 3, name: "Charlie" }
]);

db.orders.insertMany([
  { _id: 1, customer_id: 1, total: 50 },
  { _id: 2, customer_id: 1, total: 100 },
  { _id: 3, customer_id: 2, total: 75 }
]);
```

Query example

```
db.customers.aggregate([
  {
    $lookup: {
      from: "orders",
```

```
      localField: "_id",
      foreignField: "customer_id",
      as: "orders"
    }
  }
]);
```

Output

```
[
  {
    _id: 1,
    name: 'Alice',
    orders: [
      { _id: 2, customer_id: 1, total: 100 },
      { _id: 1, customer_id: 1, total: 50 }
    ]
  },
  { _id: 3, name: 'Charlie', orders: [] },
  {
    _id: 2,
    name: 'Bob',
    orders: [ { _id: 3, customer_id: 2, total: 75 } ]
  }
]
```

Join conditions and correlated subqueries (MongoDB Shell)

New from version 8.0.2.

In addition to a single equality match, `$lookup` in Amazon DocumentDB can run an aggregation pipeline on the joined collection. This supports uncorrelated subqueries, which run the same subquery for every input document, and correlated subqueries, which use variables defined from each input document's fields to return results correlated to that document.

Syntax

To run a correlated subquery, define variables from the input document's fields with `let`, and reference them in the `pipeline` stages:

```
{
  $lookup: {
    from: <collection to join>,
```

```
    let: { <var_1>: <expression>, ..., <var_n>: <expression> },
    pipeline: [ <pipeline to run on the joined collection> ],
    as: <output array field>
  }
}
```

Parameters

- `let`: A document that defines variables from the fields of the input documents. To reference a variable in the pipeline stages, use the `"$$<variable>"` syntax. The variables can be accessed by the stages in the `pipeline`, including additional `$lookup` stages nested in the `pipeline`. A `$match` stage requires the use of the `$expr` operator to access the variables. Omit `let` to run an uncorrelated subquery.
- `pipeline`: The aggregation pipeline to run on the `from` collection. The pipeline cannot access fields of the input documents directly; define variables with `let` and reference the variables instead. To return all documents, specify an empty pipeline `[]`.

The `$eq`, `$gt`, `$gte`, `$lt`, and `$lte` comparison operators placed in an `$expr` operator inside a `$match` stage can use an index on the `from` collection when they compare a field of the joined collection with a `let` variable or a constant.

Create sample documents

```
db.customers.insertMany([
  { _id: 1, name: "Alice", min_total: 60 },
  { _id: 2, name: "Bob", min_total: 50 },
  { _id: 3, name: "Charlie", min_total: 20 }
]);

db.orders.insertMany([
  { _id: 1, customer_id: 1, total: 50 },
  { _id: 2, customer_id: 1, total: 100 },
  { _id: 3, customer_id: 2, total: 75 }
]);
```

Query example

The following example joins each customer with their orders, and keeps only the orders whose `total` is greater than or equal to that customer's own `min_total` value. The join condition and the per-customer filter both reference `let` variables inside `$expr`:

```
db.customers.aggregate([
  {
    $lookup: {
      from: "orders",
      let: { customer_id: "$_id", min_total: "$min_total" },
      pipeline: [
        {
          $match: {
            $expr: {
              $and: [
                { $eq: [ "$customer_id", "$$customer_id" ] },
                { $gte: [ "$total", "$$min_total" ] }
              ]
            }
          }
        },
        { $project: { _id: 0, total: 1 } }
      ],
      as: "qualifying_orders"
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Alice', min_total: 60, qualifying_orders: [ { total: 100 } ] },
  { _id: 2, name: 'Bob', min_total: 50, qualifying_orders: [ { total: 75 } ] },
  { _id: 3, name: 'Charlie', min_total: 20, qualifying_orders: [] }
]
```

For each customer, the subquery is evaluated with that customer's `_id` and `min_total` values, so each document receives its own correlated result.

Correlated subqueries using concise syntax (MongoDB Shell)

New from version 8.0.2.

Amazon DocumentDB also supports the concise correlated subquery syntax, which combines the equality match on `localField` and `foreignField` with a `pipeline`. The concise syntax removes the requirement to express the equality match inside an `$expr` operator in a `$match` stage.

Syntax

```
{
  $lookup: {
    from: <collection to join>,
    localField: <field from the input documents>,
    foreignField: <field from the documents of the "from" collection>,
    let: { <var_1>: <expression>, ..., <var_n>: <expression> },
    pipeline: [ <pipeline to run> ],
    as: <output array field>
  }
}
```

Query example

The following example returns the same results as the previous correlated subquery example. The equality match between `_id` and `customer_id` is expressed with `localField` and `foreignField` instead of an `$expr` condition:

```
db.customers.aggregate([
  {
    $lookup: {
      from: "orders",
      localField: "_id",
      foreignField: "customer_id",
      let: { min_total: "$min_total" },
      pipeline: [
        { $match: { $expr: { $gte: [ "$total", "$$min_total" ] } } },
        { $project: { _id: 0, total: 1 } }
      ],
      as: "qualifying_orders"
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Alice', min_total: 60, qualifying_orders: [ { total: 100 } ] },
  { _id: 2, name: 'Bob', min_total: 50, qualifying_orders: [ { total: 75 } ] },
  { _id: 3, name: 'Charlie', min_total: 20, qualifying_orders: [] }
]
```

Supported stages in correlated subqueries

When a `$lookup` pipeline uses `let`, the following stages can reference the `"$$<variable>"` variables:

- `$match` (using the `$expr` operator)
- `$project`
- `$addFields`
- `$set`
- `$group`
- `$bucket`
- `$bucketAuto`
- `$sortByCount`
- `$lookup` (nested)
- `$graphLookup`

> **Note**
>
> New from version 8.0.3.
> When a `$facet` or `$unionWith` stage appears in the correlated `$lookup` pipeline, the stages inside its sub-pipeline can also reference these variables.

Stages that do not evaluate variable expressions, such as `$sort`, `$skip`, `$limit`, `$count`, `$sample`, `$unwind`, and `$unset`, can also be used in the pipeline but cannot reference the variables.

The following stages are not supported in a `$lookup` pipeline that uses `let`: `$redact`, `$replaceRoot`, `$replaceWith`, `$geoNear`, `$setWindowFields`, `$fill`, and `$match` with `$text`.

Restrictions

- The `pipeline` cannot include the `$out` or `$merge` stages.

Code examples

To view a code example for using the $lookup command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  await client.connect();

  const db = client.db('test');

  const result = await db.collection('customers').aggregate([
    {
      $lookup: {
        from: 'orders',
        localField: '_id',
        foreignField: 'customer_id',
        as: 'orders'
      }
    }
  ]).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    db = client.test

    collection = db.customers

    pipeline = [
        {
            "$lookup": {
                "from": "orders",
                "localField": "_id",
                "foreignField": "customer_id",
                "as": "orders"
            }
        }
    ]

    result = collection.aggregate(pipeline)

    for doc in result:
        print(doc)

    client.close()

example()
```

Correlated subquery code examples

To view a code example for using the `$lookup` stage with a correlated subquery (using `let` and `pipeline`), choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');

  const customers = db.collection('customers');
```

```
  await customers.insertMany([
    { _id: 1, name: "Alice", min_total: 60 },
    { _id: 2, name: "Bob", min_total: 50 },
    { _id: 3, name: "Charlie", min_total: 20 }
  ]);
  const orders = db.collection('orders');
  await orders.insertMany([
    { _id: 1, customer_id: 1, total: 50 },
    { _id: 2, customer_id: 1, total: 100 },
    { _id: 3, customer_id: 2, total: 75 }
  ]);

  // Correlated subquery: filter each customer's orders with a per-customer
 threshold
  const result = await customers.aggregate([
    {
      $lookup: {
        from: 'orders',
        localField: '_id',
        foreignField: 'customer_id',
        let: { min_total: '$min_total' },
        pipeline: [
          { $match: { $expr: { $gte: [ '$total', '$$min_total' ] } } },
          { $project: { _id: 0, total: 1 } }
        ],
        as: 'qualifying_orders'
      }
    }
  ]).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client.test

    customers = db.customers
    customers.insert_many([
        { "_id": 1, "name": "Alice", "min_total": 60 },
        { "_id": 2, "name": "Bob", "min_total": 50 },
        { "_id": 3, "name": "Charlie", "min_total": 20 }
    ])
    orders = db.orders
    orders.insert_many([
        { "_id": 1, "customer_id": 1, "total": 50 },
        { "_id": 2, "customer_id": 1, "total": 100 },
        { "_id": 3, "customer_id": 2, "total": 75 }
    ])

    # Correlated subquery: filter each customer's orders with a per-customer
 threshold
    pipeline = [
        {
            "$lookup": {
                "from": "orders",
                "localField": "_id",
                "foreignField": "customer_id",
                "let": { "min_total": "$min_total" },
                "pipeline": [
                    { "$match": { "$expr": { "$gte": [ "$total", "$
$min_total" ] } } },
                    { "$project": { "_id": 0, "total": 1 } }
                ],
                "as": "qualifying_orders"
            }
        }
    ]

    result = customers.aggregate(pipeline)

    for doc in result:
        print(doc)

    client.close()
```

```
example()
```

$ltrim

New from version 4.0.

Not supported by Elastic cluster.

The `$ltrim` operator in Amazon DocumentDB is used to remove leading characters from a string. By default, it removes leading whitespace characters, but you can also specify a set of characters to remove by passing the chars argument.

Parameters

- `input`: The input string from which to remove leading whitespace characters.
- `chars`: (optional) To remove specific characters.

Example (MongoDB Shell)

The following example demonstrates the usage of `$ltrim` to remove specified characters (" *") from the beginning of a string.

Create sample documents

```
db.collection.insertMany([
  { name: " *John Doe", age: 30 },
  { name: "Jane Doe*", age: 25 },
  { name: "  Bob Smith  ", age: 35 }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      _id: 0,
      name: {
        $ltrim: { input: "$name", chars: " *" }
      },
```

```
      age: 1
    }
  }
]);
```

Output

```
[
  { "name": "John Doe", "age": 30 },
  { "name": "Jane Doe ", "age": 25 },
  { "name": "Bob Smith  ", "age": 35 }
]
```

Code examples

To view a code example for using the `$ltrim` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();

    const db = client.db('test');
    const collection = db.collection('collection');

    const pipeline = [
      {
        $project: {
          _id: 0,
          name: {
            $ltrim: {
              input: '$name',
              chars: ' *'
            }
          },
```

```
          age: 1
        }
      }
    ];

    const result = await collection.aggregate(pipeline).toArray();

    console.dir(result, { depth: null });

  } finally {
    await client.close();
  }
}

example().catch(console.error);
```

Python

```
from pymongo import MongoClient

def example():
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client.test
        collection = db.collection

        pipeline = [
            {
                "$project": {
                    "_id": 0,
                    "name": {
                        "$ltrim": {
                            "input": "$name",
                            "chars": " *"
                        }
                    },
                    "age": 1
                }
            }
        ]
```

```
        results = collection.aggregate(pipeline)

        for doc in results:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

example()
```

$map

The $map operator in Amazon DocumentDB allows you to apply a specified expression to each element in an array and return a new array with the transformed elements. This operator is particularly useful for manipulating and transforming data within arrays, which can help simplify your application code and improve query performance by pushing the array processing to the database level.

Parameters

- input: The array to be transformed.
- as: (optional) The name of the variable to be used within the in expression to represent the current element being processed.
- in: The expression to be applied to each element in the input array.

Example (MongoDB Shell)

The following example demonstrates how to use the $map operator to transform an array of numbers, doubling each value.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, numbers: [1, 2, 3, 4, 5] },
  { _id: 2, numbers: [10, 20, 30, 40, 50] }
```

```
])
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      doubledNumbers: { $map: { input: "$numbers", as: "num", in: { $multiply: ["$
$num", 2] } } }
    }
  }
])
```

Output

```
[
  { _id: 1, doubledNumbers: [2, 4, 6, 8, 10] },
  { _id: 2, doubledNumbers: [20, 40, 60, 80, 100] }
]
```

Code examples

To view a code example for using the $map command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.aggregate([
    {
      $project: {
        doubledNumbers: { $map: { input: "$numbers", as: "num", in: { $multiply: ["$
$num", 2] } } }
      }
```

```
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.collection

    result = list(collection.aggregate([
        {
            '$project': {
                'doubledNumbers': { '$map': { 'input': '$numbers', 'as': 'num',
 'in': { '$multiply': ['$$num', 2] } } }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$match

The $match pipeline stage in Amazon DocumentDB is used to filter the input documents to only those that match the specified query criteria. It is one of the most commonly used pipeline stages in aggregation operations. The $match stage is applied before any other pipeline stages, allowing you to efficiently reduce the number of documents that need to be processed by the subsequent stages.

Parameters

- `query`: A document that expresses the selection criteria for the operation. The query document uses the same syntax as the `find()` method.

Example (MongoDB Shell)

The following example demonstrates the use of the `$match` stage to filter documents based on a specific field value.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, name: "John", age: 25, city: "New York" },
  { _id: 2, name: "Jane", age: 30, city: "Los Angeles" },
  { _id: 3, name: "Bob", age: 35, city: "Chicago" },
  { _id: 4, name: "Alice", age: 40, city: "Miami" }
]);
```

Query example

```
db.collection.aggregate([
  { $match: { age: { $gt: 30 } } },
  { $project: { _id: 1, name: 1, city: 1 } }
]);
```

Output

```
[
  { "_id": 3, "name": "Bob", "city": "Chicago" },
  { "_id": 4, "name": "Alice", "city": "Miami" }
]
```

The `$match` stage filters the documents to include only those where the `age` field is greater than 30.

Code examples

To view a code example for using the `$match` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.aggregate([
    { $match: { age: { $gt: 30 } } },
    { $project: { _id: 1, name: 1, city: 1 } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    db = client['test']
    collection = db['collection']

    result = list(collection.aggregate([
        { '$match': { 'age': { '$gt': 30 } } },
        { '$project': { '_id': 1, 'name': 1, 'city': 1 } }
    ]))

    print(result)
    client.close()
```

```
example()
```

$max

The $max aggregation operator returns the maximum value of a specified field across a set of documents. This operator is useful for finding the highest value in a set of documents.

Parameters

- expression: The expression to use to calculate the maximum value.

Example (MongoDB Shell)

The following example demonstrates how to use the $max operator to find the maximum score in a collection of student documents. The $group stage groups all documents together, and the $max operator is used to calculate the maximum value of the score field across all documents.

Create sample documents

```
db.students.insertMany([
  { name: "John", score: 85 },
  { name: "Jane", score: 92 },
  { name: "Bob", score: 78 },
  { name: "Alice", score: 90 }
])
```

Query example

```
db.students.aggregate([
  { $group: { _id: null, maxScore: { $max: "$score" } } },
  { $project: { _id: 0, maxScore: 1 } }
])
```

Output

```
[ { maxScore: 92 } ]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The $max operator can also be used as a window operator in the $setWindowFields stage. In this context, it returns the maximum value for the documents in each window. You specify the operator under the output field, and optionally define the window boundaries with a window document.

> **Note**
>
> When used as a window operator in $setWindowFields, $max is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by ticker, sorts each partition by hour, and returns the highest price seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        highestSoFar: {
          $max: "$price",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "highestSoFar": 50 },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "highestSoFar": 50 },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "highestSoFar": 60 },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "highestSoFar": 60 },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "highestSoFar": 30 },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "highestSoFar": 35 },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "highestSoFar": 35 }
]
```

Each document is augmented with `highestSoFar`, the running maximum of `price` within its partition up to and including the current document.

Code examples

To view a code example for using the `$max` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  // Accumulator usage: maximum value across a group
  const students = db.collection('students');
  await students.insertMany([
    { name: "John", score: 85 },
    { name: "Jane", score: 92 },
    { name: "Bob", score: 78 },
    { name: "Alice", score: 90 }
  ]);
  const accumulatorResult = await students.aggregate([
    { $group: { _id: null, maxScore: { $max: "$score" } } }
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);
```

```
  // Window operator usage: running maximum within each partition
  const stockPrices = db.collection('stockPrices');
  await stockPrices.insertMany([
    { _id: 1, ticker: "ABC", hour: 1, price: 50 },
    { _id: 2, ticker: "ABC", hour: 2, price: 45 },
    { _id: 3, ticker: "ABC", hour: 3, price: 60 },
    { _id: 4, ticker: "ABC", hour: 4, price: 40 },
    { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
    { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
    { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
  ]);
  const windowResult = await stockPrices.aggregate([
    {
      $setWindowFields: {
        partitionBy: "$ticker",
        sortBy: { hour: 1 },
        output: {
          highestSoFar: {
            $max: "$price",
            window: { documents: ["unbounded", "current"] }
          }
        }
      }
    }
  ]).toArray();
  console.log('Window result:', windowResult);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
```

```
    # Accumulator usage: maximum value across a group
    students = db.students
    students.insert_many([
        { "name": "John", "score": 85 },
        { "name": "Jane", "score": 92 },
        { "name": "Bob", "score": 78 },
        { "name": "Alice", "score": 90 }
    ])
    accumulator_result = list(students.aggregate([
        { "$group": { "_id": None, "maxScore": { "$max": "$score" } } }
    ]))
    print('Accumulator result:', accumulator_result)

    # Window operator usage: running maximum within each partition
    stock_prices = db.stockPrices
    stock_prices.insert_many([
        { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50 },
        { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45 },
        { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60 },
        { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40 },
        { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30 },
        { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35 },
        { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25 }
    ])
    window_result = list(stock_prices.aggregate([
        {
            "$setWindowFields": {
                "partitionBy": "$ticker",
                "sortBy": { "hour": 1 },
                "output": {
                    "highestSoFar": {
                        "$max": "$price",
                        "window": { "documents": ["unbounded", "current"] }
                    }
                }
            }
        }
    ]))
    print('Window result:', window_result)

    client.close()

example()
```

$maxN

New from version 8.0.1.

The `$maxN` operator in Amazon DocumentDB returns the N largest values. When used as an accumulator in a `$group` stage, it returns an array of the N maximum values in each group. When used as an array expression operator, it returns the N maximum elements of an array.

Parameters

- `input`: An expression that resolves to the array or field from which to return the maximum `n` values.
- `n`: An expression that resolves to a positive integer specifying how many maximum values to return.

Behavior

Array expression operator behavior

- You cannot specify a value of `n` less than `1`.
- `$maxN` filters out `null` values found in the `input` array.
- If the specified `n` is greater than or equal to the number of elements in the `input` array, `$maxN` returns all elements in the `input` array.
- If `input` resolves to a non-array value, the aggregation operation errors.
- If `input` contains both numeric and string elements, the string elements are sorted before numeric elements according to the BSON comparison order.

Accumulator behavior

- When used as an accumulator, `n` must be a positive integral expression that is either a constant or depends on the `_id` value for `$group`.
- `$maxN` filters out null and missing values.
- If the group contains fewer than `n` elements, `$maxN` returns all elements in the group.
- `$maxN` compares input data following the BSON comparison order to determine the appropriate output type. When the input data contains multiple data types, the `$maxN` output type is the highest in the comparison order.

Output ordering

$maxN returns values in no particular sort order. If guaranteeing a particular sort order is a requirement, use $topN instead, or wrap the result with $sortArray.

Example (MongoDB Shell)

The following example shows how to use the $maxN accumulator to retrieve the two highest scores for each subject.

Create sample documents

```
db.scores.insertMany([
  { subject: "math", score: 85 },
  { subject: "math", score: 72 },
  { subject: "math", score: 93 },
  { subject: "math", score: 68 },
  { subject: "science", score: 90 },
  { subject: "science", score: 78 },
  { subject: "science", score: 65 },
  { subject: "science", score: 88 }
]);
```

Query example

```
db.scores.aggregate([
  { $group: { _id: "$subject", highestTwo: { $maxN: { input: "$score", n: 2 } } } }
]);
```

Output

```
[
  { "_id": "math", "highestTwo": [85, 93] },
  { "_id": "science", "highestTwo": [88, 90] }
]
```

Expression usage example (MongoDB Shell)

The $maxN operator can also be used as an expression within a $project stage to return the N largest elements from an array field.

Create sample documents

```
db.readings.insertMany([
  { _id: 1, sensor: "A", values: [45, 12, 78, 3, 56] },
  { _id: 2, sensor: "B", values: [90, 23, 67, 11, 44] }
]);
```

Query example

```
db.readings.aggregate([
  { $project: {
      highestThree: { $maxN: { input: "$values", n: 3 } }
    }}
]);
```

Output

```
[
  { "_id": 1, "highestThree": [45, 56, 78] },
  { "_id": 2, "highestThree": [44, 90, 67] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$maxN` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of up to the n largest values for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$maxN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
```

```
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the two highest prices seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        highestTwoSoFar: {
          $maxN: { input: "$price", n: 2 },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "highestTwoSoFar": [50] },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "highestTwoSoFar": [50, 45] },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "highestTwoSoFar": [60, 50] },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "highestTwoSoFar": [60, 50] },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "highestTwoSoFar": [30] },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "highestTwoSoFar": [35, 30] },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "highestTwoSoFar": [35, 30] }
]
```

Each document is augmented with `highestTwoSoFar`, the two largest `price` values within its window. As noted in the **Output ordering** section, the order of values within the returned array is not guaranteed.

Code examples

To view a code example for using the $maxN operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in $group), expression usage (in $project), and window operator usage (in $setWindowFields):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: N largest values per group
    const scores = db.collection('scores');
    await scores.insertMany([
      { subject: "math", score: 85 },
      { subject: "math", score: 72 },
      { subject: "math", score: 93 },
      { subject: "math", score: 68 },
      { subject: "science", score: 90 },
      { subject: "science", score: 78 },
      { subject: "science", score: 65 },
      { subject: "science", score: 88 }
    ]);
    const accumulatorResult = await scores.aggregate([
      { $group: { _id: "$subject", highestTwo: { $maxN: { input: "$score", n:
 2 } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Expression usage: N largest elements from an array field
    const readings = db.collection('readings');
    await readings.insertMany([
      { _id: 1, sensor: "A", values: [45, 12, 78, 3, 56] },
      { _id: 2, sensor: "B", values: [90, 23, 67, 11, 44] }
    ]);
```

```
    const expressionResult = await readings.aggregate([
      { $project: { highestThree: { $maxN: { input: "$values", n: 3 } } } }
    ]).toArray();
    console.log('Expression result:', expressionResult);

    // Window operator usage: N largest values over a window in each partition
    const stockPrices = db.collection('stockPrices');
    await stockPrices.insertMany([
      { _id: 1, ticker: "ABC", hour: 1, price: 50 },
      { _id: 2, ticker: "ABC", hour: 2, price: 45 },
      { _id: 3, ticker: "ABC", hour: 3, price: 60 },
      { _id: 4, ticker: "ABC", hour: 4, price: 40 },
      { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
      { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
      { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
    ]);
    const windowResult = await stockPrices.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$ticker",
          sortBy: { hour: 1 },
          output: {
            highestTwoSoFar: {
              $maxN: { input: "$price", n: 2 },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: N largest values per group
        scores = db['scores']
        scores.insert_many([
            { 'subject': 'math', 'score': 85 },
            { 'subject': 'math', 'score': 72 },
            { 'subject': 'math', 'score': 93 },
            { 'subject': 'math', 'score': 68 },
            { 'subject': 'science', 'score': 90 },
            { 'subject': 'science', 'score': 78 },
            { 'subject': 'science', 'score': 65 },
            { 'subject': 'science', 'score': 88 }
        ])
        accumulator_result = list(scores.aggregate([
            { '$group': { '_id': '$subject', 'highestTwo': { '$maxN': { 'input':
 '$score', 'n': 2 } } } }
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: N largest elements from an array field
        readings = db['readings']
        readings.insert_many([
            { '_id': 1, 'sensor': 'A', 'values': [45, 12, 78, 3, 56] },
            { '_id': 2, 'sensor': 'B', 'values': [90, 23, 67, 11, 44] }
        ])
        expression_result = list(readings.aggregate([
            { '$project': { 'highestThree': { '$maxN': { 'input': '$values', 'n':
 3 } } } }
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: N largest values over a window in each partition
        stock_prices = db['stockPrices']
        stock_prices.insert_many([
            { '_id': 1, 'ticker': 'ABC', 'hour': 1, 'price': 50 },
            { '_id': 2, 'ticker': 'ABC', 'hour': 2, 'price': 45 },
            { '_id': 3, 'ticker': 'ABC', 'hour': 3, 'price': 60 },
```

```
            { '_id': 4, 'ticker': 'ABC', 'hour': 4, 'price': 40 },
            { '_id': 5, 'ticker': 'XYZ', 'hour': 1, 'price': 30 },
            { '_id': 6, 'ticker': 'XYZ', 'hour': 2, 'price': 35 },
            { '_id': 7, 'ticker': 'XYZ', 'hour': 3, 'price': 25 }
        ])
        window_result = list(stock_prices.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$ticker',
                    'sortBy': { 'hour': 1 },
                    'output': {
                        'highestTwoSoFar': {
                            '$maxN': { 'input': '$price', 'n': 2 },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$median

New from version 8.0.1.

The `$median` operator in Amazon DocumentDB calculates the median value of numeric data. As an accumulator, it computes the median of numeric values across documents within a group in the `$group` stage of an aggregation pipeline. As an expression, it calculates the median of an array of numbers.

Parameters

- `input`: An expression that resolves to a numeric value or an array of numeric values.
- `method`: A string specifying the calculation method. Currently only `"approximate"` is supported, which uses the t-digest algorithm.

Behavior

The `"approximate"` method uses the t-digest algorithm to calculate an approximate median. The result is an existing value from the dataset rather than an interpolation between values. Precision improves as the number of data points increases.

Example (MongoDB Shell)

The following example shows how to use the `$median` operator to calculate the median test score per class.

Create sample documents

```
db.students.insertMany([
  { class: "A", score: 72 },
  { class: "A", score: 85 },
  { class: "A", score: 90 },
  { class: "A", score: 68 },
  { class: "A", score: 95 },
  { class: "B", score: 80 },
  { class: "B", score: 75 },
  { class: "B", score: 92 },
  { class: "B", score: 88 },
  { class: "B", score: 70 }
]);
```

Query example

```
db.students.aggregate([
  { $group: {
      _id: "$class",
      medianScore: { $median: { input: "$score", method: "approximate" } }
    }}
]);
```

Output

```
[
  { "_id": "A", "medianScore": 85 },
  { "_id": "B", "medianScore": 80 }
]
```

Expression usage example (MongoDB Shell)

The `$median` operator can also be used as an expression within a `$project` stage to compute the median of an array field.

Create sample documents

```
db.surveys.insertMany([
  { _id: 1, ratings: [3, 5, 7, 9, 2] },
  { _id: 2, ratings: [10, 20, 30, 40, 50] },
  { _id: 3, ratings: [1, 1, 2, 3, 5] }
]);
```

Query example

```
db.surveys.aggregate([
  { $project: {
      medianRating: { $median: { input: "$ratings", method: "approximate" } }
    }}
]);
```

Output

```
[
  { "_id": 1, "medianRating": 5 },
  { "_id": 2, "medianRating": 30 },
  { "_id": 3, "medianRating": 2 }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$median` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it computes the median of the numeric values for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$median` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.temperatures.insertMany([
  { _id: 1, city: "A", reading: 60 },
  { _id: 2, city: "A", reading: 65 },
  { _id: 3, city: "A", reading: 70 },
  { _id: 4, city: "B", reading: 80 },
  { _id: 5, city: "B", reading: 85 },
  { _id: 6, city: "B", reading: 90 }
]);
```

Query example

The following example partitions the documents by `city` and computes the median `reading` across all documents in each partition.

```
db.temperatures.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$city",
      output: {
        medianReading: {
          $median: { input: "$reading", method: "approximate" },
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "city": "A", "reading": 60, "medianReading": 65 },
  { "_id": 2, "city": "A", "reading": 65, "medianReading": 65 },
  { "_id": 3, "city": "A", "reading": 70, "medianReading": 65 },
```

```
  { "_id": 4, "city": "B", "reading": 80, "medianReading": 85 },
  { "_id": 5, "city": "B", "reading": 85, "medianReading": 85 },
  { "_id": 6, "city": "B", "reading": 90, "medianReading": 85 }
]
```

Each document is augmented with `medianReading`, the approximate median of `reading` across all documents in its partition.

Code examples

To view a code example for using the `$median` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: median across grouped documents
    const students = db.collection('students');
    await students.insertMany([
      { class: "A", score: 72 },
      { class: "A", score: 85 },
      { class: "A", score: 90 },
      { class: "A", score: 68 },
      { class: "A", score: 95 },
      { class: "B", score: 80 },
      { class: "B", score: 75 },
      { class: "B", score: 92 },
      { class: "B", score: 88 },
      { class: "B", score: 70 }
    ]);
    const accumulatorResult = await students.aggregate([
```

```
    { $group: {
        _id: "$class",
        medianScore: { $median: { input: "$score", method: "approximate" } }
      }}
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Expression usage: median of an array field
  const surveys = db.collection('surveys');
  await surveys.insertMany([
    { _id: 1, ratings: [3, 5, 7, 9, 2] },
    { _id: 2, ratings: [10, 20, 30, 40, 50] },
    { _id: 3, ratings: [1, 1, 2, 3, 5] }
  ]);
  const expressionResult = await surveys.aggregate([
    { $project: {
        medianRating: { $median: { input: "$ratings", method: "approximate" } }
      }}
  ]).toArray();
  console.log('Expression result:', expressionResult);

  // Window operator usage: median across each partition
  const temperatures = db.collection('temperatures');
  await temperatures.insertMany([
    { _id: 1, city: "A", reading: 60 },
    { _id: 2, city: "A", reading: 65 },
    { _id: 3, city: "A", reading: 70 },
    { _id: 4, city: "B", reading: 80 },
    { _id: 5, city: "B", reading: 85 },
    { _id: 6, city: "B", reading: 90 }
  ]);
  const windowResult = await temperatures.aggregate([
    {
      $setWindowFields: {
        partitionBy: "$city",
        output: {
          medianReading: {
            $median: { input: "$reading", method: "approximate" },
            window: { documents: ["unbounded", "unbounded"] }
          }
        }
      }
    }
  ]).toArray();
```

```
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: median across grouped documents
        students = db['students']
        students.insert_many([
            { 'class': 'A', 'score': 72 },
            { 'class': 'A', 'score': 85 },
            { 'class': 'A', 'score': 90 },
            { 'class': 'A', 'score': 68 },
            { 'class': 'A', 'score': 95 },
            { 'class': 'B', 'score': 80 },
            { 'class': 'B', 'score': 75 },
            { 'class': 'B', 'score': 92 },
            { 'class': 'B', 'score': 88 },
            { 'class': 'B', 'score': 70 }
        ])
        accumulator_result = list(students.aggregate([
            { '$group': {
                '_id': '$class',
                'medianScore': { '$median': { 'input': '$score', 'method':
 'approximate' } }
            }}
        ]))
        print('Accumulator result:', accumulator_result)
```

```
        # Expression usage: median of an array field
        surveys = db['surveys']
        surveys.insert_many([
            { '_id': 1, 'ratings': [3, 5, 7, 9, 2] },
            { '_id': 2, 'ratings': [10, 20, 30, 40, 50] },
            { '_id': 3, 'ratings': [1, 1, 2, 3, 5] }
        ])
        expression_result = list(surveys.aggregate([
            { '$project': {
                'medianRating': { '$median': { 'input': '$ratings', 'method':
'approximate' } }
            }}
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: median across each partition
        temperatures = db['temperatures']
        temperatures.insert_many([
            { '_id': 1, 'city': 'A', 'reading': 60 },
            { '_id': 2, 'city': 'A', 'reading': 65 },
            { '_id': 3, 'city': 'A', 'reading': 70 },
            { '_id': 4, 'city': 'B', 'reading': 80 },
            { '_id': 5, 'city': 'B', 'reading': 85 },
            { '_id': 6, 'city': 'B', 'reading': 90 }
        ])
        window_result = list(temperatures.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$city',
                    'output': {
                        'medianReading': {
                            '$median': { 'input': '$reading', 'method':
'approximate' },
                            'window': { 'documents': ['unbounded', 'unbounded'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()
```

```
example()
```

$meta

The $meta aggregation operator accesses metadata associated with documents in an aggregation pipeline. It is commonly used to retrieve text search scores and sort results by relevance.

Parameters

- textScore: Retrieves the text search score indicating document relevance to the search query.

Example (MongoDB Shell)

The following example demonstrates using the $meta operator in an aggregation pipeline to retrieve and sort by text search scores.

Create sample documents

```
db.articles.createIndex({ content: "text" });

db.articles.insertMany([
  { _id: 1, title: "Python Programming", content: "Python is a versatile programming
 language used for web development." },
  { _id: 2, title: "Python Guide", content: "Learn Python programming with Python
 tutorials and Python examples." },
  { _id: 3, title: "Java Basics", content: "Java is another popular programming
 language." }
]);
```

Query example

```
db.articles.aggregate([
  { $match: { $text: { $search: "Python" } } },
  { $addFields: { score: { $meta: "textScore" } } },
  { $sort: { score: -1 } }
]);
```

Output

```
[
  {
    _id: 2,
    title: 'Python Guide',
    content: 'Learn Python programming with Python tutorials and Python examples.',
    score: 1.5
  },
  {
    _id: 1,
    title: 'Python Programming',
    content: 'Python is a versatile programming language used for web development.',
    score: 0.75
  }
]
```

Code examples

To view a code example for using the $meta aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('articles');

  const result = await collection.aggregate([
    { $match: { $text: { $search: "Python" } } },
    { $addFields: { score: { $meta: "textScore" } } },
    { $sort: { score: -1 } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['articles']

    result = list(collection.aggregate([
        { '$match': { '$text': { '$search': 'Python' } } },
        { '$addFields': { 'score': { '$meta': 'textScore' } } },
        { '$sort': { 'score': -1 } }
    ]))

    print(result)
    client.close()

example()
```

$merge

Introduced in 8.0

The `$merge` aggregation stage in Amazon DocumentDB is used to merge the results of the previous pipeline stage into a target collection. This is useful for updating or inserting documents in a target collection based on the data from the input documents.

The `$merge` stage allows you to perform various actions based on the matching condition between the input documents and the target collection, such as:

```
- Insert new documents
- Update existing documents
- Delete documents
- Fail the operation if there are any conflicts
```

Parameters

- `into`: (required) The name of the target collection to merge the input documents into.

- on: (required) The field(s) to use as the matching condition between the input documents and the target collection.
- `whenMatched`: (optional) The action to perform when the input document matches an existing document in the target collection. Supported values are: `"merge"`, `"replace"`, `"keepExisting"`, and `"fail"`. Amazon DocumentDB does not support specifying an aggregation pipeline as the `whenMatched` value.
- `whenNotMatched`: (optional) The action to perform when the input document does not match any document in the target collection. Supported values are: `"insert"` and `"fail"`.
- `bypassDocumentValidation`: (optional) When set to `true`, documents written to the target collection by the `$merge` stage bypass any schema validation rules (`$jsonSchema`) defined on that collection. The default is `false`, which enforces the target collection's validation rules. This is supported in Amazon DocumentDB 8.0.

Example (MongoDB Shell)

The following example demonstrates how to use the `$merge` stage to update a `users` collection with new data from an input pipeline.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "John Doe", email: "john@example.com" },
  { _id: 2, name: "Jane Smith", email: "jane@example.com" }
]);

db.inputData.insertMany([
  { _id: 1, name: "John Doe", email: "john@example.com", phone: "123-456-7890" },
  { _id: 3, name: "Bob Johnson", email: "bob@example.com", phone: "987-654-3210" }
]);
```

Query example

```
db.inputData.aggregate([
  {
    $merge: {
      into: "users",
      on: "_id",
      whenMatched: "merge",
      whenNotMatched: "insert"
```

```
    }
  }
])
```

Output

After running the $merge pipeline, the users collection will contain the following documents:

```
[
  { _id: 1, name: "John Doe", email: "john@example.com", phone: "123-456-7890" },
  { _id: 2, name: "Jane Smith", email: "jane@example.com" },
  { _id: 3, name: "Bob Johnson", email: "bob@example.com", phone: "987-654-3210" }
]
```

Code examples

To view a code example for using the $merge command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the $merge operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  await db.collection('inputData').aggregate([
    {
      $merge: {
        into: 'users',
        on: '_id',
        whenMatched: 'merge',
        whenNotMatched: 'insert'
      }
    }
  ]).toArray();

  const users = await db.collection('users').find({}).toArray();
```

```
  console.log(users);

  await client.close();
}

example();
```

Python

Here's an example of using the $merge operator in a Python application:

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']

    # Assumes collections 'users' and 'inputData' already exist with sample data
    db.inputData.aggregate([
        {
            '$merge': {
                'into': 'users',
                'on': '_id',
                'whenMatched': 'merge',
                'whenNotMatched': 'insert'
            }
        }
    ])

    users = list(db.users.find({}))
    print(users)

    client.close()

example()
```

$mergeObjects

The `$mergeObjects` operator in Amazon DocumentDB is used to combine multiple documents or objects into a single document. This operator is particularly useful when you need to merge the

contents of two or more documents or objects, potentially overwriting values from one object with those from another.

Parameters

- `expression1`: The first object to be merged.
- `expression2`: (optional) The second object to be merged.
- `expression3`: (optional) Additional objects to be merged.

Example (MongoDB Shell)

The following example demonstrates how to use the `$mergeObjects` operator to combine two objects.

Create sample documents

```
db.collection.insertMany([
  { "_id": 1, "name": "John", "address": { "city": "New York", "state": "NY" } },
  { "_id": 2, "name": "Jane", "address": { "city": "Los Angeles", "state": "CA" } }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      "combinedAddress": {
        $mergeObjects: ["$address", { "country": "USA" }]
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "combinedAddress": {
      "city": "New York",
```

```
      "state": "NY",
      "country": "USA"
    }
  },
  {
    "_id": 2,
    "combinedAddress": {
      "city": "Los Angeles",
      "state": "CA",
      "country": "USA"
    }
  }
]
```

Code examples

To view a code example for using the `$mergeObjects` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.aggregate([
    {
      $project: {
        "combinedAddress": {
          $mergeObjects: ["$address", { "country": "USA" }]
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['collection']

    result = list(collection.aggregate([
        {
            '$project': {
                "combinedAddress": {
                    "$mergeObjects": ["$address", { "country": "USA" }]
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$millisecond

The `$millisecond` operator in Amazon DocumentDB is used to extract the millisecond portion of a date value.

Parameters

None

Example (MongoDB Shell)

This example demonstrates how to use the `$millisecond` operator to extract the millisecond portion of a date value.

Create sample documents

```
db.events.insert([
  {
    "name": "Event 1",
    "timestamp": ISODate("2023-04-21T10:30:15.123Z")
  },
  {
    "name": "Event 2",
    "timestamp": ISODate("2023-04-21T10:30:15.456Z")
  },
  {
    "name": "Event 3",
    "timestamp": ISODate("2023-04-21T10:30:15.789Z")
  }
])
```

Query example

```
db.events.aggregate([
  {
    $project: {
      name: 1,
      milliseconds: { $millisecond: "$timestamp" }
    }
  }
])
```

Output

```
[
  {
    "_id": ObjectId("644332a42054ed1b0d15f0c1"),
    "name": "Event 1",
    "milliseconds": 123
  },
  {
    "_id": ObjectId("644332a42054ed1b0d15f0c2"),
    "name": "Event 2",
    "milliseconds": 456
  },
  {
```

```
    "_id": ObjectId("644332a42054ed1b0d15f0c3"),
    "name": "Event 3",
    "milliseconds": 789
  }
]
```

Code examples

To view a code example for using the $millisecond command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    {
      $project: {
        name: 1,
        milliseconds: { $millisecond: '$timestamp' }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient
```

```
client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
events = db['events']

result = list(events.aggregate([
    {
        '$project': {
            'name': 1,
            'milliseconds': { '$millisecond': '$timestamp' }
        }
    }
]))

print(result)

client.close()
```

$min

The $min aggregation operator returns the minimum value of a specified field across a set of documents. This operator is useful for finding the lowest value in a set of documents.

Parameters

- expression: The expression to evaluate. This can be a field path, a variable, or any expression that resolves to a value.

Example (MongoDB Shell)

The following example demonstrates the usage of the $min operator to find the minimum value of the age field across multiple documents.

Create sample documents

```
db.users.insertMany([
  { name: "John", age: 35 },
  { name: "Jane", age: 28 },
  { name: "Bob", age: 42 },
  { name: "Alice", age: 31 }
```

```
]);
```

Query example

```
db.users.aggregate([
  { $group: { _id: null, minAge: { $min: "$age" } } },
  { $project: { _id: 0, minAge: 1 } }
])
```

Output

```
[ { minAge: 28 } ]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$min` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the minimum value for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$min` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the lowest price seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        lowestSoFar: {
          $min: "$price",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "lowestSoFar": 50 },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "lowestSoFar": 45 },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "lowestSoFar": 45 },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "lowestSoFar": 40 },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "lowestSoFar": 30 },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "lowestSoFar": 30 },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "lowestSoFar": 25 }
]
```

Each document is augmented with `lowestSoFar`, the running minimum of `price` within its partition up to and including the current document.

Code examples

To view a code example for using the `$min` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  // Accumulator usage: minimum value across a group
  const users = db.collection('users');
  await users.insertMany([
    { name: "John", age: 35 },
    { name: "Jane", age: 28 },
    { name: "Bob", age: 42 },
    { name: "Alice", age: 31 }
  ]);
  const accumulatorResult = await users.aggregate([
    { $group: { _id: null, minAge: { $min: "$age" } } }
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Window operator usage: running minimum within each partition
  const stockPrices = db.collection('stockPrices');
  await stockPrices.insertMany([
    { _id: 1, ticker: "ABC", hour: 1, price: 50 },
    { _id: 2, ticker: "ABC", hour: 2, price: 45 },
    { _id: 3, ticker: "ABC", hour: 3, price: 60 },
    { _id: 4, ticker: "ABC", hour: 4, price: 40 },
    { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
    { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
    { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
  ]);
  const windowResult = await stockPrices.aggregate([
    {
      $setWindowFields: {
        partitionBy: "$ticker",
        sortBy: { hour: 1 },
        output: {
          lowestSoFar: {
            $min: "$price",
            window: { documents: ["unbounded", "current"] }
          }
        }
      }
    }
```

```
  ]).toArray();
  console.log('Window result:', windowResult);

  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']

    # Accumulator usage: minimum value across a group
    users = db.users
    users.insert_many([
        { "name": "John", "age": 35 },
        { "name": "Jane", "age": 28 },
        { "name": "Bob", "age": 42 },
        { "name": "Alice", "age": 31 }
    ])
    accumulator_result = list(users.aggregate([
        { "$group": { "_id": None, "minAge": { "$min": "$age" } } }
    ]))
    print('Accumulator result:', accumulator_result)

    # Window operator usage: running minimum within each partition
    stock_prices = db.stockPrices
    stock_prices.insert_many([
        { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50 },
        { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45 },
        { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60 },
        { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40 },
        { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30 },
        { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35 },
        { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25 }
    ])
    window_result = list(stock_prices.aggregate([
```

```
        {
            "$setWindowFields": {
                "partitionBy": "$ticker",
                "sortBy": { "hour": 1 },
                "output": {
                    "lowestSoFar": {
                        "$min": "$price",
                        "window": { "documents": ["unbounded", "current"] }
                    }
                }
            }
        }
    ]))
    print('Window result:', window_result)

    client.close()

example()
```

$minN

New from version 8.0.1.

The $minN operator in Amazon DocumentDB returns the N smallest values. When used as an accumulator in a $group stage, it returns an array of the N minimum values in each group. When used as an array expression operator, it returns the N minimum elements of an array.

Parameters

- input: An expression that resolves to the array or field from which to return the minimum n values.
- n: An expression that resolves to a positive integer specifying how many minimum values to return.

Behavior

Array expression operator behavior

- You cannot specify a value of n less than 1.
- $minN filters out null values found in the input array.

- If the specified n is greater than or equal to the number of elements in the `input` array, `$minN` returns all elements in the `input` array.
- If `input` resolves to a non-array value, the aggregation operation errors.
- If `input` contains both numeric and string elements, the numeric elements are sorted before string elements according to the BSON comparison order.

Accumulator behavior

- When used as an accumulator, n must be a positive integral expression that is either a constant or depends on the `_id` value for `$group`.
- `$minN` filters out null and missing values.
- If the group contains fewer than n elements, `$minN` returns all elements in the group.
- `$minN` compares input data following the BSON comparison order to determine the appropriate output type. When the input data contains multiple data types, the `$minN` output type is the lowest in the comparison order.

Output ordering

`$minN` returns values in no particular sort order. If guaranteeing a particular sort order is a requirement, use `$bottomN` instead, or wrap the result with `$sortArray`.

Example (MongoDB Shell)

The following example shows how to use the `$minN` accumulator to retrieve the two lowest scores for each subject.

Create sample documents

```
db.scores.insertMany([
  { subject: "math", score: 85 },
  { subject: "math", score: 72 },
  { subject: "math", score: 93 },
  { subject: "math", score: 68 },
  { subject: "science", score: 90 },
  { subject: "science", score: 78 },
  { subject: "science", score: 65 },
  { subject: "science", score: 88 }
]);
```

Query example

```
db.scores.aggregate([
  { $group: { _id: "$subject", lowestTwo: { $minN: { input: "$score", n: 2 } } } }
]);
```

Output

```
[
  { "_id": "math", "lowestTwo": [72, 68] },
  { "_id": "science", "lowestTwo": [78, 65] }
]
```

Expression usage example (MongoDB Shell)

The $minN operator can also be used as an expression within a $project stage to return the N smallest elements from an array field.

Create sample documents

```
db.readings.insertMany([
  { _id: 1, sensor: "A", values: [45, 12, 78, 3, 56] },
  { _id: 2, sensor: "B", values: [90, 23, 67, 11, 44] }
]);
```

Query example

```
db.readings.aggregate([
  { $project: {
      lowestThree: { $minN: { input: "$values", n: 3 } }
    }}
]);
```

Output

```
[
  { "_id": 1, "lowestThree": [45, 12, 3] },
  { "_id": 2, "lowestThree": [44, 23, 11] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$minN` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of up to the `n` smallest values for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$minN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.stockPrices.insertMany([
  { _id: 1, ticker: "ABC", hour: 1, price: 50 },
  { _id: 2, ticker: "ABC", hour: 2, price: 45 },
  { _id: 3, ticker: "ABC", hour: 3, price: 60 },
  { _id: 4, ticker: "ABC", hour: 4, price: 40 },
  { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
  { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
  { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
]);
```

Query example

The following example partitions the documents by `ticker`, sorts each partition by `hour`, and returns the two lowest prices seen from the start of the partition through the current document.

```
db.stockPrices.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$ticker",
      sortBy: { hour: 1 },
      output: {
        lowestTwoSoFar: {
          $minN: { input: "$price", n: 2 },
          window: { documents: ["unbounded", "current"] }
        }
```

```
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "ticker": "ABC", "hour": 1, "price": 50, "lowestTwoSoFar": [50] },
  { "_id": 2, "ticker": "ABC", "hour": 2, "price": 45, "lowestTwoSoFar": [45, 50] },
  { "_id": 3, "ticker": "ABC", "hour": 3, "price": 60, "lowestTwoSoFar": [45, 50] },
  { "_id": 4, "ticker": "ABC", "hour": 4, "price": 40, "lowestTwoSoFar": [40, 45] },
  { "_id": 5, "ticker": "XYZ", "hour": 1, "price": 30, "lowestTwoSoFar": [30] },
  { "_id": 6, "ticker": "XYZ", "hour": 2, "price": 35, "lowestTwoSoFar": [30, 35] },
  { "_id": 7, "ticker": "XYZ", "hour": 3, "price": 25, "lowestTwoSoFar": [25, 30] }
]
```

Each document is augmented with `lowestTwoSoFar`, the two smallest `price` values within its window. As noted in the **Output ordering** section, the order of values within the returned array is not guaranteed.

Code examples

To view a code example for using the `$minN` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: N smallest values per group
    const scores = db.collection('scores');
```

```
    await scores.insertMany([
      { subject: "math", score: 85 },
      { subject: "math", score: 72 },
      { subject: "math", score: 93 },
      { subject: "math", score: 68 },
      { subject: "science", score: 90 },
      { subject: "science", score: 78 },
      { subject: "science", score: 65 },
      { subject: "science", score: 88 }
    ]);
    const accumulatorResult = await scores.aggregate([
      { $group: { _id: "$subject", lowestTwo: { $minN: { input: "$score", n:
2 } } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Expression usage: N smallest elements from an array field
    const readings = db.collection('readings');
    await readings.insertMany([
      { _id: 1, sensor: "A", values: [45, 12, 78, 3, 56] },
      { _id: 2, sensor: "B", values: [90, 23, 67, 11, 44] }
    ]);
    const expressionResult = await readings.aggregate([
      { $project: { lowestThree: { $minN: { input: "$values", n: 3 } } } }
    ]).toArray();
    console.log('Expression result:', expressionResult);

    // Window operator usage: N smallest values over a window in each partition
    const stockPrices = db.collection('stockPrices');
    await stockPrices.insertMany([
      { _id: 1, ticker: "ABC", hour: 1, price: 50 },
      { _id: 2, ticker: "ABC", hour: 2, price: 45 },
      { _id: 3, ticker: "ABC", hour: 3, price: 60 },
      { _id: 4, ticker: "ABC", hour: 4, price: 40 },
      { _id: 5, ticker: "XYZ", hour: 1, price: 30 },
      { _id: 6, ticker: "XYZ", hour: 2, price: 35 },
      { _id: 7, ticker: "XYZ", hour: 3, price: 25 }
    ]);
    const windowResult = await stockPrices.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$ticker",
          sortBy: { hour: 1 },
          output: {
```

```
            lowestTwoSoFar: {
              $minN: { input: "$price", n: 2 },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: N smallest values per group
        scores = db['scores']
        scores.insert_many([
            { 'subject': 'math', 'score': 85 },
            { 'subject': 'math', 'score': 72 },
            { 'subject': 'math', 'score': 93 },
            { 'subject': 'math', 'score': 68 },
            { 'subject': 'science', 'score': 90 },
            { 'subject': 'science', 'score': 78 },
            { 'subject': 'science', 'score': 65 },
            { 'subject': 'science', 'score': 88 }
        ])
        accumulator_result = list(scores.aggregate([
```

```
            { '$group': { '_id': '$subject', 'lowestTwo': { '$minN': { 'input':
'$score', 'n': 2 } } } }
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: N smallest elements from an array field
        readings = db['readings']
        readings.insert_many([
            { '_id': 1, 'sensor': 'A', 'values': [45, 12, 78, 3, 56] },
            { '_id': 2, 'sensor': 'B', 'values': [90, 23, 67, 11, 44] }
        ])
        expression_result = list(readings.aggregate([
            { '$project': { 'lowestThree': { '$minN': { 'input': '$values', 'n':
3 } } } }
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: N smallest values over a window in each partition
        stock_prices = db['stockPrices']
        stock_prices.insert_many([
            { '_id': 1, 'ticker': 'ABC', 'hour': 1, 'price': 50 },
            { '_id': 2, 'ticker': 'ABC', 'hour': 2, 'price': 45 },
            { '_id': 3, 'ticker': 'ABC', 'hour': 3, 'price': 60 },
            { '_id': 4, 'ticker': 'ABC', 'hour': 4, 'price': 40 },
            { '_id': 5, 'ticker': 'XYZ', 'hour': 1, 'price': 30 },
            { '_id': 6, 'ticker': 'XYZ', 'hour': 2, 'price': 35 },
            { '_id': 7, 'ticker': 'XYZ', 'hour': 3, 'price': 25 }
        ])
        window_result = list(stock_prices.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$ticker',
                    'sortBy': { 'hour': 1 },
                    'output': {
                        'lowestTwoSoFar': {
                            '$minN': { 'input': '$price', 'n': 2 },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)
```

```
    finally:
        client.close()

example()
```

$minute

The `$minute` aggregation pipeline stage in Amazon DocumentDB extracts the minute value from a date or timestamp field.

This operator is useful when you need to perform date and time-based calculations or grouping within your aggregation pipeline.

Parameters

- `expression`: The date or timestamp field from which to extract the minute value.

Example (MongoDB Shell)

The following example demonstrates how to use the `$minute` operator to group the documents by the minute value extracted from the timestamp field and count the number of documents in each group.

Create sample documents

```
db.events.insertMany([
  { timestamp: new Date("2023-04-15T10:30:25.000Z") },
  { timestamp: new Date("2023-04-15T10:30:35.000Z") },
  { timestamp: new Date("2023-04-15T10:31:05.000Z") },
  { timestamp: new Date("2023-04-15T10:31:45.000Z") },
  { timestamp: new Date("2023-04-15T10:32:15.000Z") }
]);
```

Query example

```
db.events.aggregate([
  {
    $group: {
      _id: {
        minute: { $minute: "$timestamp" }
```

```
      },
      count: { $count: {} }
    }
  },
  { $sort: { "_id.minute": 1 } }
]);
```

Output

```
[
  { "_id": { "minute": 30 }, "count": 2 },
  { "_id": { "minute": 31 }, "count": 2 },
  { "_id": { "minute": 32 }, "count": 1 }
]
```

Code examples

To view a code example for using the `$minute` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('events');

  await collection.insertMany([
    { timestamp: new Date("2023-04-15T10:30:25.000Z") },
    { timestamp: new Date("2023-04-15T10:30:35.000Z") },
    { timestamp: new Date("2023-04-15T10:31:05.000Z") },
    { timestamp: new Date("2023-04-15T10:31:45.000Z") },
    { timestamp: new Date("2023-04-15T10:32:15.000Z") }
  ]);

  const result = await collection.aggregate([
    {
      $group: {
        _id: {
```

```
          minute: { $minute: "$timestamp" }
        },
        count: { $count: {} }
      }
    },
    { $sort: { "_id.minute": 1 } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
from datetime import datetime

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['events']

    collection.insert_many([
        {'timestamp': datetime(2023, 4, 15, 10, 30, 25)},
        {'timestamp': datetime(2023, 4, 15, 10, 30, 35)},
        {'timestamp': datetime(2023, 4, 15, 10, 31, 5)},
        {'timestamp': datetime(2023, 4, 15, 10, 31, 45)},
        {'timestamp': datetime(2023, 4, 15, 10, 32, 15)}
    ])

    result = list(collection.aggregate([
        {
            '$group': {
                '_id': {
                    'minute': {'$minute': '$timestamp'}
                },
                'count': {'$count': {}}
            }
        },
```

```
        {'$sort': {'_id.minute': 1}}
    ]))

    print(result)
    client.close()

example()
```

$mod

The $mod operator is an arithmetic operator that performs a modulo operation on a number. It returns the remainder of dividing one number by another. This operator is commonly used to determine if a number is odd or even, or to distribute items into a finite number of groups.

Parameters

- `expression1`: The dividend expression.
- `expression2`: The divisor expression.

Example (MongoDB Shell)

This example demonstrates how to use the $mod operator to determine the number of leftover widgets when shipping in packages of 100.

Create sample documents

```
db.widgets.insertMany([
  { "_id" : 1, "widget" : "A", "count" : 80372 },
  { "_id" : 2, "widget" : "B", "count" : 409282 },
  { "_id" : 3, "widget" : "C", "count" : 60739 }
])
```

Query example

```
db.widgets.aggregate([
  { $addFields: { leftOver: { $mod: [ "$count", 100 ] } } }
])
```

Output

```
[
  { "_id" : 1, "widget" : "A", "count" : 80372, "leftOver" : 72 },
  { "_id" : 2, "widget" : "B", "count" : 409282, "leftOver" : 82 },
  { "_id" : 3, "widget" : "C", "count" : 60739, "leftOver" : 39 }
]
```

The output shows the remainder of the count divided by 100 for each document, which represents the number of leftover widgets when shipping in packages of 100.

Code examples

To view a code example for using the $mod command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const widgets = db.collection('widgets');

  await widgets.insertMany([
    { "_id" : 1, "widget" : "A", "count" : 80372 },
    { "_id" : 2, "widget" : "B", "count" : 409282 },
    { "_id" : 3, "widget" : "C", "count" : 60739 }
  ]);

  const result = await widgets.aggregate([
    { $addFields: { leftOver: { $mod: [ "$count", 100 ] } } }
  ]).toArray();

  console.log(result);
  client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
widgets = db['widgets']

widgets.insert_many([
    { "_id" : 1, "widget" : "A", "count" : 80372 },
    { "_id" : 2, "widget" : "B", "count" : 409282 },
    { "_id" : 3, "widget" : "C", "count" : 60739 }
])

result = list(widgets.aggregate([
    { "$addFields": { "leftOver": { "$mod": [ "$count", 100 ] } } }
]))

print(result)
client.close()
```

$month

The `$month` operator in Amazon DocumentDB returns the month of a date as a number between 1 and 12. This operator is useful for extracting the month component from a date field and performing date-based aggregations and analyses.

Parameters

- `date_expression`: This is the expression or field that contains the date or timestamp from which you want to extract the month.

Example (MongoDB Shell)

The following example demonstrates how to use the `$month` operator to extract the month from a date field and group the data by month.

Create sample documents

```
db.sales.insert([
  { product: "abc123", price: 10.99, date: new Date("2022-01-15") },
  { product: "def456", price: 15.50, date: new Date("2022-02-28") },
  { product: "ghi789", price: 8.25, date: new Date("2022-03-10") },
  { product: "jkl012", price: 12.75, date: new Date("2022-04-05") },
  { product: "mno345", price: 18.99, date: new Date("2022-05-20") }
]);
```

Query example

```
db.sales.aggregate([
  { $group: {
      _id: { month: { $month: "$date" } },
      totalSales: { $sum: "$price" }
    }},
  { $sort: { "_id.month": 1 } }
]);
```

Output

```
[
  { _id: { month: 1 }, totalSales: 10.99 },
  { _id: { month: 2 }, totalSales: 15.5 },
  { _id: { month: 3 }, totalSales: 8.25 },
  { _id: { month: 4 }, totalSales: 12.75 },
  { _id: { month: 5 }, totalSales: 18.99 }
]
```

Code examples

To view a code example for using the `$month` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function groupSalesByMonth() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('sales');

    const pipeline = [
      {
        $group: {
          _id: { month: { $month: "$date" } },
          totalSales: { $sum: "$price" }
        }
      },
      {
        $sort: { "_id.month": 1 }
      }
    ];

    const results = await collection.aggregate(pipeline).toArray();

    console.dir(results, { depth: null });

  } finally {
    await client.close();
  }
}

groupSalesByMonth().catch(console.error);
```

Python

```
from pymongo import MongoClient

def group_sales_by_month():

    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client.test
        collection = db.sales
```

```
        pipeline = [
            {
                "$group": {
                    "_id": { "$month": "$date" },
                    "totalSales": { "$sum": "$price" }
                }
            },
            {
                "$sort": { "_id": 1 }
            }
        ]

        results = collection.aggregate(pipeline)

        for doc in results:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

group_sales_by_month()
```

$multiply

The `$multiply` operator in Amazon DocumentDB is used to multiply the values of two or more fields or expressions. This operator is particularly useful for performing arithmetic operations on numeric fields within documents. It can be employed in various stages of the aggregation pipeline, such as `$project` and `$addFields`, to create new fields or modify existing ones.

Parameters

- `expression1`: The first numeric expression to be multiplied.
- `expression2`: The second numeric expression to be multiplied.
- `[expression3, ...]`: (optional) Additional numeric expressions to be multiplied.

Example (MongoDB Shell)

The following example demonstrates using `$multiply` to calculate `bonus_miles` by multiplying `base_miles` and `bonus_rate` for customers who used a credit card for the trip.

Sample documents

```
db.miles.insertMany([
{ "_id": 1, "customer_name": "Arnav Desai", "member_since":
 ISODate("1997-03-01T00:00:00Z"), "base_miles": 2500, "bonus_rate": 1.8, "credit_card":
 true, "trip_cost": 250 },
{ "_id": 2, "customer_name": "Jorge Souza", "member_since":
 ISODate("2004-01-10T00:00:00Z"), "base_miles": 1890, "bonus_rate": 1.4, "credit_card":
 true, "trip_cost": 189 },
{ "_id": 3, "customer_name": "Saanvi Sarkar", "member_since":
 ISODate("1999-11-22T00:00:00Z"), "base_miles": 3250, "bonus_rate": 1.8, "credit_card":
 false, "trip_cost": 325 },
{ "_id": 4, "customer_name": "Paulo Santos", "member_since":
 ISODate("2021-06-19T00:00:00Z"), "base_miles": 2980, "bonus_rate": 1.2, "credit_card":
 true, "trip_cost": 298 },
{ "_id": 5, "customer_name": "Wang Xiulan", "member_since":
 ISODate("1995-12-04T00:00:00Z"), "base_miles": 1350, "bonus_rate": 1.9, "credit_card":
 false, "trip_cost": 135 }
]);
```

Query example

```
db.miles.aggregate([
  {
    $match: { credit_card: true }
  },
  {
    $project: {
      customer_name: 1,
      base_miles: 1,
      bonus_rate:1,
      credit_card: 1,
      total_miles: {
        $multiply: ["$base_miles", "$bonus_rate"]
      }
    }
  }
]);
```

Output

```
[
  { _id: 1, customer_name: 'Arnav Desai', base_miles: 12500, bonus_rate: 1.8,
 credit_card: true, total_miles: 22500 },
  { _id: 3, customer_name: 'Saanvi Sarkar',base_miles: 15200, bonus_rate: 1.8,
 credit_card: true, total_miles: 27360 },
  { _id: 4, customer_name: 'Paulo Santos', base_miles: 3400, bonus_rate: 1.1,
 credit_card: true, total_miles: 3740 }
]
```

Code examples

To view a code example for using the `$multiply` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function multiplyBonusMiles() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('miles');

  const result = await collection.aggregate([
    { $match: { credit_card: true } },
    {
      $project: {
        customer_name: 1,
        base_miles: 1,
        bonus_rate: 1,
        credit_card: 1,
        total_miles: {
          $multiply: ["$base_miles", "$bonus_rate"]
        }
      }
    }
  ]).toArray();

  console.log(result);
```

```
  await client.close();
}

multiplyBonusMiles();
```

Python

```
from pymongo import MongoClient

def multiply_bonus_miles():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['miles']

    result = list(collection.aggregate([
        {'$match': {'credit_card': True}},
        {
            '$project': {
                'customer_name': 1,
                'base_miles': 1,
                'bonus_rate': 1,
                'credit_card': 1,
                'total_miles': {
                    '$multiply': ['$base_miles', '$bonus_rate']
                }
            }
        }
    ]))

    print(result)
    client.close()

multiply_bonus_miles()
```

$natural

The `$natural` operator in Amazon DocumentDB is used to sort the documents in their natural order, which is the order in which they were inserted into the collection. This is in contrast to the default sorting behavior, which is to sort the documents based on the values of the specified fields.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the `$natural` operator to sort the documents in a collection in their natural order.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "name": "María García", "age": 28 },
  { "_id": 2, "name": "Arnav Desai", "age": 32 },
  { "_id": 3, "name": "Li Juan", "age": 25 },
  { "_id": 4, "name": "Carlos Salazar", "age": 41 },
  { "_id": 5, "name": "Sofia Martínez", "age": 35 }
]);
```

Query example

```
db.people.find({}, { "_id": 1, "name": 1 }).sort({ "$natural": 1 });
```

Output

```
[
  { "_id": 1, "name": "María García" },
  { "_id": 2, "name": "Arnav Desai" },
  { "_id": 3, "name": "Li Juan" },
  { "_id": 4, "name": "Carlos Salazar" },
  { "_id": 5, "name": "Sofia Martínez" }
]
```

The query sorts the documents in the collection in their natural order, which is the order in which they were inserted.

Code examples

To view a code example for using the `$natural` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const documents = await collection.find({}, { projection: { _id: 1, name: 1 } })
    .sort({ $natural: 1 })
    .toArray();

  console.log(documents);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['people']

    documents = list(collection.find({}, {'_id': 1, 'name': 1}).sort('$natural', 1))
    print(documents)

    client.close()

example()
```

$ne

The $ne aggregation operator compares two values and returns `true` if they are not equal, otherwise returns `false`.

Parameters

- `expression1`: The first value to compare.
- `expression2`: The second value to compare.

Example (MongoDB Shell)

The following example demonstrates using the $ne operator to identify orders with status changes.

Create sample documents

```
db.orders.insertMany([
  { _id: 1, orderId: "A123", status: "shipped", expectedStatus: "shipped" },
  { _id: 2, orderId: "B456", status: "pending", expectedStatus: "shipped" },
  { _id: 3, orderId: "C789", status: "delivered", expectedStatus: "delivered" }
]);
```

Query example

```
db.orders.aggregate([
  {
    $project: {
      orderId: 1,
      status: 1,
      expectedStatus: 1,
      needsAttention: { $ne: ["$status", "$expectedStatus"] }
    }
  }
]);
```

Output

```
[
  { _id: 1, orderId: 'A123', status: 'shipped', expectedStatus: 'shipped',
 needsAttention: false },
```

```
  { _id: 2, orderId: 'B456', status: 'pending', expectedStatus: 'shipped',
 needsAttention: true },
  { _id: 3, orderId: 'C789', status: 'delivered', expectedStatus: 'delivered',
 needsAttention: false }
]
```

Code examples

To view a code example for using the $ne aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('orders');

  const result = await collection.aggregate([
    {
      $project: {
        orderId: 1,
        status: 1,
        expectedStatus: 1,
        needsAttention: { $ne: ["$status", "$expectedStatus"] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['orders']

    result = list(collection.aggregate([
        {
            '$project': {
                'orderId': 1,
                'status': 1,
                'expectedStatus': 1,
                'needsAttention': { '$ne': ['$status', '$expectedStatus'] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$not

The $not aggregation operator performs a logical NOT operation on an expression. It returns `true` if the expression evaluates to `false`, and `false` if the expression evaluates to `true`.

Parameters

- `expression`: The expression to negate.

Example (MongoDB Shell)

The following example demonstrates using the $not operator to invert boolean values.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "Alice", active: true },
  { _id: 2, name: "Bob", active: false },
  { _id: 3, name: "Charlie", active: true }
```

```
]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      name: 1,
      active: 1,
      inactive: { $not: ["$active"] }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Alice', active: true, inactive: false },
  { _id: 2, name: 'Bob', active: false, inactive: true },
  { _id: 3, name: 'Charlie', active: true, inactive: false }
]
```

Code examples

To view a code example for using the $not aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
```

```
        active: 1,
        inactive: { $not: ["$active"] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['users']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'active': 1,
                'inactive': { '$not': ['$active'] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$objectToArray

The `$objectToArray` aggregation operator in Amazon DocumentDB converts an object (or document) into an array. The input to the operator is a document, and the output consists of an

array element for each field-value pair in the input document. This operator is useful when you need to work with the individual fields of a document as an array, such as when you want to find the document with the maximum or minimum value for a particular field.

Parameters

- `expression`: The document expression to convert into an array.

Example (MongoDB Shell)

The following example demonstrates how to use the `$objectToArray` operator to find the document with the maximum inventory for a video rental store chain.

Create sample documents

```
db.videos.insertMany([
  {
    "_id": 1,
    "name": "Live Soft",
    "inventory": {
      "Des Moines": 1000,
      "Ames": 500
    }
  },
  {
    "_id": 2,
    "name": "Top Pilot",
    "inventory": {
      "Mason City": 250,
      "Des Moines": 1000
    }
  },
  {
    "_id": 3,
    "name": "Romancing the Rock",
    "inventory": {
      "Mason City": 250,
      "Ames": 500
    }
  },
  {
    "_id": 4,
    "name": "Bravemind",
```

```
    "inventory": {
      "Mason City": 250,
      "Des Moines": 1000,
      "Ames": 500
    }
  }
]);
```

Query example

```
db.videos.aggregate([
  {
    $project: {
      name: 1,
      videos: {
        $objectToArray: "$inventory"
      }
    }
  },
  {
    $unwind: "$videos"
  },
  {
    $group: {
      _id: "$name",
      maxInventory: {
        $max: "$videos.v"
      }
    }
  }
]);
```

Output

```
[
  {
    "_id": "Bravemind",
    "maxInventory": 1000
  },
  {
    "_id": "Live Soft",
    "maxInventory": 1000
  },
```

```
  {
    "_id": "Romancing the Rock",
    "maxInventory": 500
  },
  {
    "_id": "Top Pilot",
    "maxInventory": 1000
  }
]
```

Code examples

To view a code example for using the `$objectToArray` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findMaxInventory() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const videos = db.collection('videos');

  const result = await videos.aggregate([
    {
      $project: {
        name: 1,
        videos: {
          $objectToArray: "$inventory"
        }
      }
    },
    {
      $unwind: "$videos"
    },
    {
      $group: {
        _id: "$name",
        maxInventory: {
          $max: "$videos.v"
```

```
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

findMaxInventory();
```

Python

```
from pymongo import MongoClient

def find_max_inventory():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    videos = db['videos']

    result = list(videos.aggregate([
        {
            '$project': {
                'name': 1,
                'videos': {
                    '$objectToArray': '$inventory'
                }
            }
        },
        {
            '$unwind': '$videos'
        },
        {
            '$group': {
                '_id': '$name',
                'maxInventory': {
                    '$max': '$videos.v'
                }
            }
        }
    ]))
```

```
    print(result)
    client.close()

find_max_inventory()
```

$or

The $or aggregation operator evaluates multiple expressions and returns true if at least one expression evaluates to true. It returns false only if all expressions are false.

Parameters

- expressions: An array of expressions to evaluate.

Example (MongoDB Shell)

The following example demonstrates using the $or operator to check if products meet any of multiple criteria.

Create sample documents

```
db.items.insertMany([
  { _id: 1, name: "Widget", price: 150, onSale: false },
  { _id: 2, name: "Gadget", price: 45, onSale: false },
  { _id: 3, name: "Tool", price: 200, onSale: true }
]);
```

Query example

```
db.items.aggregate([
  {
    $project: {
      name: 1,
      price: 1,
      onSale: 1,
      goodDeal: {
        $or: [
          { $lt: ["$price", 50] },
          { $eq: ["$onSale", true] }
```

```
        ]
      }
    }
  }
]);
```

Output

```
[
  { _id: 1, name: 'Widget', price: 150, onSale: false, goodDeal: false },
  { _id: 2, name: 'Gadget', price: 45, onSale: false, goodDeal: true },
  { _id: 3, name: 'Tool', price: 200, onSale: true, goodDeal: true }
]
```

Code examples

To view a code example for using the `$or` aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('items');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        price: 1,
        onSale: 1,
        goodDeal: {
          $or: [
            { $lt: ["$price", 50] },
            { $eq: ["$onSale", true] }
          ]
        }
```

```
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['items']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'price': 1,
                'onSale': 1,
                'goodDeal': {
                    '$or': [
                        { '$lt': ['$price', 50] },
                        { '$eq': ['$onSale', True] }
                    ]
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$out

The $out operator in Amazon DocumentDB is used to write the result of an aggregation pipeline to a specified collection.

$out should be the last stage in the pipeline.

Parameters

- output_collection: The name of the output collection to write the aggregation results to.

Note: If the collection already exists, it will be replaced with the results of the aggregation stage.

Example (MongoDB Shell)

The following example demonstrates how to use the $out operator in Amazon DocumentDB to write the results of an aggregation pipeline to a new collection.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Wireless Headphones", category: "Electronics", price: 100.0 },
  { _id: 2, name: "Smartphone", category: "Electronics", price: 200.0 },
  { _id: 3, name: "JavaScript Guide", category: "Books", price: 50.0 },
  { _id: 4, name: "Database Design Handbook", category: "Books", price: 75.0 }
]);
```

Query example

```
db.products.aggregate([
  { $group: { _id: "$category", totalPrice: { $sum: "$price" } } },
  { $out: "product_categories" }
])
```

Output

None (the results are written to the output collection).

The aggregation pipeline groups the products by category and calculates the total price of the items for each category. The $out operator writes the results to a new collection named "product_categories".

To view the results in the output collection:

```
db.product_categories.find()
[
{ "_id" : "Books", "totalPrice" : 125 },
{ "_id" : "Electronics", "totalPrice" : 300 }
]
```

Code examples

To view a code example for using the $out command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function demo_out_operator() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const products = db.collection('products');

  // Execute aggregation with $out - results are stored in 'product_categories'
 collection
  await products.aggregate([
    { $group: { _id: "$category", totalPrice: { $sum: "$price" } } },
    { $out: "product_categories" }
  ]).toArray();

  // Retrieve the results from the output collection (limited to 20 records)
  const productCategories = db.collection('product_categories');
  const results = await productCategories.find({}).limit(20).toArray();

  console.log('Results stored in product_categories collection:', results);
  await client.close();
}

demo_out_operator();
```

Python

```
from pymongo import MongoClient

def demo_out_operator():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    products = db['products']

    # Execute aggregation with $out - results are stored in 'product_categories'
 collection
    list(products.aggregate([
        { '$group': { '_id': '$category', 'totalPrice': { '$sum': '$price' } } },
        { '$out': 'product_categories' }
    ]))

    # Retrieve the results from the output collection (limited to 20 records)
    product_categories = db['product_categories']
    results = list(product_categories.find({}).limit(20))

    print('Results stored in product_categories collection:', results)
    client.close()

demo_out_operator()
```

$percentile

New from version 8.0.1.

The `$percentile` operator in Amazon DocumentDB calculates specified percentile values for numeric data. As an accumulator, it computes percentile values across documents within a group in the `$group` stage of an aggregation pipeline. As an expression, it calculates percentiles for an array of numbers.

Parameters

- `input`: An expression that resolves to a numeric value or an array of numeric values.
- p: An array of percentile values between 0 and 1, where each value represents a percentile to calculate. For example, `[0.25, 0.5, 0.75]` calculates the 25th, 50th, and 75th percentiles.

- `method`: A string specifying the calculation method. Currently only `"approximate"` is supported.

Behavior

The `"approximate"` method uses the t-digest algorithm to calculate approximate, percentile-based metrics. With small datasets, distinct percentile values may resolve to the same result. For example, with only 5 values per group, p90 and p99 may both return the maximum value in the group. Precision improves as the number of data points increases.

Example (MongoDB Shell)

The following example shows how to use the `$percentile` operator to calculate the 25th and 75th percentile of scores per class.

Create sample documents

```
db.students.insertMany([
  { class: "A", score: 72 },
  { class: "A", score: 85 },
  { class: "A", score: 90 },
  { class: "A", score: 68 },
  { class: "A", score: 95 },
  { class: "B", score: 80 },
  { class: "B", score: 75 },
  { class: "B", score: 92 },
  { class: "B", score: 88 },
  { class: "B", score: 70 }
]);
```

Query example

```
db.students.aggregate([
  { $group: {
      _id: "$class",
      percentiles: { $percentile: { input: "$score", p: [0.25, 0.75], method:
 "approximate" } }
    }}
]);
```

Output

```
[
  { "_id": "A", "percentiles": [72, 90] },
  { "_id": "B", "percentiles": [75, 88] }
]
```

Expression usage example (MongoDB Shell)

The `$percentile` operator can also be used as an expression within a `$project` stage to compute percentiles of an array field.

Create sample documents

```
db.surveys.insertMany([
  { _id: 1, responses: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] },
  { _id: 2, responses: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] }
]);
```

Query example

```
db.surveys.aggregate([
  { $project: {
      quartiles: { $percentile: { input: "$responses", p: [0.25, 0.5, 0.75], method:
 "approximate" } }
    }}
]);
```

Output

```
[
  { "_id": 1, "quartiles": [6, 10, 16] },
  { "_id": 2, "quartiles": [5, 9, 15] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$percentile` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it computes the specified percentile values for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$percentile` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.sensorData.insertMany([
  { _id: 1, sensor: "A", reading: 72 },
  { _id: 2, sensor: "A", reading: 85 },
  { _id: 3, sensor: "A", reading: 90 },
  { _id: 4, sensor: "A", reading: 68 },
  { _id: 5, sensor: "A", reading: 95 },
  { _id: 6, sensor: "B", reading: 80 },
  { _id: 7, sensor: "B", reading: 75 },
  { _id: 8, sensor: "B", reading: 92 },
  { _id: 9, sensor: "B", reading: 88 },
  { _id: 10, sensor: "B", reading: 70 }
]);
```

Query example

The following example partitions the documents by `sensor` and computes the 25th and 75th percentiles of `reading` across all documents in each partition.

```
db.sensorData.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$sensor",
      output: {
        readingPercentiles: {
          $percentile: { input: "$reading", p: [0.25, 0.75], method: "approximate" },
          window: { documents: ["unbounded", "unbounded"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "sensor": "A", "reading": 72, "readingPercentiles": [72, 90] },
  { "_id": 2, "sensor": "A", "reading": 85, "readingPercentiles": [72, 90] },
  { "_id": 3, "sensor": "A", "reading": 90, "readingPercentiles": [72, 90] },
  { "_id": 4, "sensor": "A", "reading": 68, "readingPercentiles": [72, 90] },
  { "_id": 5, "sensor": "A", "reading": 95, "readingPercentiles": [72, 90] },
  { "_id": 6, "sensor": "B", "reading": 80, "readingPercentiles": [75, 88] },
  { "_id": 7, "sensor": "B", "reading": 75, "readingPercentiles": [75, 88] },
  { "_id": 8, "sensor": "B", "reading": 92, "readingPercentiles": [75, 88] },
  { "_id": 9, "sensor": "B", "reading": 88, "readingPercentiles": [75, 88] },
  { "_id": 10, "sensor": "B", "reading": 70, "readingPercentiles": [75, 88] }
]
```

Each document is augmented with `readingPercentiles`, the approximate 25th and 75th percentiles of `reading` across all documents in its partition.

Code examples

To view a code example for using the `$percentile` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: percentiles across grouped documents
    const students = db.collection('students');
    await students.insertMany([
      { class: "A", score: 72 },
      { class: "A", score: 85 },
      { class: "A", score: 90 },
```

```
    { class: "A", score: 68 },
    { class: "A", score: 95 },
    { class: "B", score: 80 },
    { class: "B", score: 75 },
    { class: "B", score: 92 },
    { class: "B", score: 88 },
    { class: "B", score: 70 }
  ]);
  const accumulatorResult = await students.aggregate([
    { $group: {
        _id: "$class",
        percentiles: { $percentile: { input: "$score", p: [0.25, 0.75], method:
"approximate" } }
      }}
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Expression usage: percentiles of an array field
  const surveys = db.collection('surveys');
  await surveys.insertMany([
    { _id: 1, responses: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] },
    { _id: 2, responses: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] }
  ]);
  const expressionResult = await surveys.aggregate([
    { $project: {
        quartiles: { $percentile: { input: "$responses", p: [0.25, 0.5, 0.75],
method: "approximate" } }
      }}
  ]).toArray();
  console.log('Expression result:', expressionResult);

  // Window operator usage: percentiles across each partition
  const sensorData = db.collection('sensorData');
  await sensorData.insertMany([
    { _id: 1, sensor: "A", reading: 72 },
    { _id: 2, sensor: "A", reading: 85 },
    { _id: 3, sensor: "A", reading: 90 },
    { _id: 4, sensor: "A", reading: 68 },
    { _id: 5, sensor: "A", reading: 95 },
    { _id: 6, sensor: "B", reading: 80 },
    { _id: 7, sensor: "B", reading: 75 },
    { _id: 8, sensor: "B", reading: 92 },
    { _id: 9, sensor: "B", reading: 88 },
    { _id: 10, sensor: "B", reading: 70 }
```

```
    ]);
    const windowResult = await sensorData.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$sensor",
          output: {
            readingPercentiles: {
              $percentile: { input: "$reading", p: [0.25, 0.75], method:
 "approximate" },
              window: { documents: ["unbounded", "unbounded"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: percentiles across grouped documents
        students = db['students']
        students.insert_many([
            { 'class': 'A', 'score': 72 },
            { 'class': 'A', 'score': 85 },
            { 'class': 'A', 'score': 90 },
            { 'class': 'A', 'score': 68 },
```

```
            { 'class': 'A', 'score': 95 },
            { 'class': 'B', 'score': 80 },
            { 'class': 'B', 'score': 75 },
            { 'class': 'B', 'score': 92 },
            { 'class': 'B', 'score': 88 },
            { 'class': 'B', 'score': 70 }
        ])
        accumulator_result = list(students.aggregate([
            { '$group': {
                '_id': '$class',
                'percentiles': { '$percentile': { 'input': '$score', 'p': [0.25,
0.75], 'method': 'approximate' } }
            }}
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: percentiles of an array field
        surveys = db['surveys']
        surveys.insert_many([
            { '_id': 1, 'responses': [2, 4, 6, 8, 10, 12, 14, 16, 18, 20] },
            { '_id': 2, 'responses': [1, 3, 5, 7, 9, 11, 13, 15, 17, 19] }
        ])
        expression_result = list(surveys.aggregate([
            { '$project': {
                'quartiles': { '$percentile': { 'input': '$responses', 'p': [0.25,
0.5, 0.75], 'method': 'approximate' } }
            }}
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: percentiles across each partition
        sensor_data = db['sensorData']
        sensor_data.insert_many([
            { '_id': 1, 'sensor': 'A', 'reading': 72 },
            { '_id': 2, 'sensor': 'A', 'reading': 85 },
            { '_id': 3, 'sensor': 'A', 'reading': 90 },
            { '_id': 4, 'sensor': 'A', 'reading': 68 },
            { '_id': 5, 'sensor': 'A', 'reading': 95 },
            { '_id': 6, 'sensor': 'B', 'reading': 80 },
            { '_id': 7, 'sensor': 'B', 'reading': 75 },
            { '_id': 8, 'sensor': 'B', 'reading': 92 },
            { '_id': 9, 'sensor': 'B', 'reading': 88 },
            { '_id': 10, 'sensor': 'B', 'reading': 70 }
        ])
```

```
        window_result = list(sensor_data.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$sensor',
                    'output': {
                        'readingPercentiles': {
                            '$percentile': { 'input': '$reading', 'p': [0.25, 0.75],
 'method': 'approximate' },
                            'window': { 'documents': ['unbounded', 'unbounded'] }
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$pow

Introduced in 8.0

The $pow operator in Amazon DocumentDB allows you to raise a number to a power. This can be useful for performing exponential calculations within your aggregation pipeline.

Parameters

- <number> (required): The number to be raised to a power.
- <exponent> (required): The power to which the number should be raised.

Example (MongoDB Shell)

The following example demonstrates how to use the $pow operator to calculate the square of a number.

Create sample documents

```
db.numbers.insertMany([
```

```
  { "_id": 1, "value": 2 },
  { "_id": 2, "value": 3 },
  { "_id": 3, "value": 4 }
]);
```

Query example

```
db.numbers.aggregate([
  { $addFields: { "square": { $pow: ["$value", 2] } } }
])
```

Output

```
[
  { "_id": 1, "value": 2, "square": 4 },
  { "_id": 2, "value": 3, "square": 9 },
  { "_id": 3, "value": 4, "square": 16 }
]
```

Code examples

To view a code example for using the $pow command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the $pow operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $addFields: { "square": { $pow: ["$value", 2] } } }
  ]).toArray();
```

```
  console.log(result);
  await client.close();
}

example();
```

Python

Here's an example of using the $pow operator in a Python application:

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['numbers']

    result = list(collection.aggregate([
        { "$addFields": { "square": { "$pow": ["$value", 2] } } }
    ]))

    print(result)
    client.close()

example()
```

$push

The $push aggregation operator returns an array of all values from a specified expression for each group. It is typically used within the $group stage to accumulate values into an array.

Parameters

- expression: The expression to evaluate for each document in the group.

Example (MongoDB Shell)

The following example demonstrates using the $push operator to collect all product names for each category.

Create sample documents

```
db.sales.insertMany([
  { _id: 1, category: "Electronics", product: "Laptop", amount: 1200 },
  { _id: 2, category: "Electronics", product: "Mouse", amount: 25 },
  { _id: 3, category: "Furniture", product: "Desk", amount: 350 },
  { _id: 4, category: "Furniture", product: "Chair", amount: 150 },
  { _id: 5, category: "Electronics", product: "Keyboard", amount: 75 }
]);
```

Query example

```
db.sales.aggregate([
  {
    $group: {
      _id: "$category",
      products: { $push: "$product" }
    }
  }
]);
```

Output

```
[
  { _id: 'Furniture', products: [ 'Desk', 'Chair' ] },
  { _id: 'Electronics', products: [ 'Laptop', 'Mouse', 'Keyboard' ] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$push` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns an array of all values of the specified expression for the documents in each window, in sort order. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$push` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.purchases.insertMany([
  { _id: 1, category: "Electronics", day: 1, product: "Laptop" },
  { _id: 2, category: "Electronics", day: 2, product: "Mouse" },
  { _id: 3, category: "Electronics", day: 3, product: "Keyboard" },
  { _id: 4, category: "Furniture", day: 1, product: "Desk" },
  { _id: 5, category: "Furniture", day: 2, product: "Chair" }
]);
```

Query example

The following example partitions the documents by `category`, sorts each partition by `day`, and returns a running list of `product` values from the start of the partition through the current document.

```
db.purchases.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$category",
      sortBy: { day: 1 },
      output: {
        productsSoFar: {
          $push: "$product",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "category": "Electronics", "day": 1, "product": "Laptop",
 "productsSoFar": [ "Laptop" ] },
  { "_id": 2, "category": "Electronics", "day": 2, "product": "Mouse", "productsSoFar":
 [ "Laptop", "Mouse" ] },
  { "_id": 3, "category": "Electronics", "day": 3, "product": "Keyboard",
 "productsSoFar": [ "Laptop", "Mouse", "Keyboard" ] },
  { "_id": 4, "category": "Furniture", "day": 1, "product": "Desk", "productsSoFar":
 [ "Desk" ] },
```

```
  { "_id": 5, "category": "Furniture", "day": 2, "product": "Chair", "productsSoFar":
 [ "Desk", "Chair" ] }
]
```

Each document is augmented with `productsSoFar`, the array of all `product` values within its partition from the start through the current document, in sort order.

Code examples

To view a code example for using the `$push` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  // Accumulator usage: collect all values per group
  const sales = db.collection('sales');
  await sales.insertMany([
    { _id: 1, category: "Electronics", product: "Laptop", amount: 1200 },
    { _id: 2, category: "Electronics", product: "Mouse", amount: 25 },
    { _id: 3, category: "Furniture", product: "Desk", amount: 350 },
    { _id: 4, category: "Furniture", product: "Chair", amount: 150 },
    { _id: 5, category: "Electronics", product: "Keyboard", amount: 75 }
  ]);
  const accumulatorResult = await sales.aggregate([
    {
      $group: {
        _id: "$category",
        products: { $push: "$product" }
      }
    }
  ]).toArray();
  console.log('Accumulator result:', accumulatorResult);

  // Window operator usage: running list of values within each partition
```

```
  const purchases = db.collection('purchases');
  await purchases.insertMany([
    { _id: 1, category: "Electronics", day: 1, product: "Laptop" },
    { _id: 2, category: "Electronics", day: 2, product: "Mouse" },
    { _id: 3, category: "Electronics", day: 3, product: "Keyboard" },
    { _id: 4, category: "Furniture", day: 1, product: "Desk" },
    { _id: 5, category: "Furniture", day: 2, product: "Chair" }
  ]);
  const windowResult = await purchases.aggregate([
    {
      $setWindowFields: {
        partitionBy: "$category",
        sortBy: { day: 1 },
        output: {
          productsSoFar: {
            $push: "$product",
            window: { documents: ["unbounded", "current"] }
          }
        }
      }
    }
  ]).toArray();
  console.log('Window result:', windowResult);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']

    # Accumulator usage: collect all values per group
    sales = db['sales']
    sales.insert_many([
```

```
        { '_id': 1, 'category': 'Electronics', 'product': 'Laptop', 'amount':
1200 },
        { '_id': 2, 'category': 'Electronics', 'product': 'Mouse', 'amount': 25 },
        { '_id': 3, 'category': 'Furniture', 'product': 'Desk', 'amount': 350 },
        { '_id': 4, 'category': 'Furniture', 'product': 'Chair', 'amount': 150 },
        { '_id': 5, 'category': 'Electronics', 'product': 'Keyboard', 'amount': 75 }
    ])
    accumulator_result = list(sales.aggregate([
        {
            '$group': {
                '_id': '$category',
                'products': { '$push': '$product' }
            }
        }
    ]))
    print('Accumulator result:', accumulator_result)

    # Window operator usage: running list of values within each partition
    purchases = db['purchases']
    purchases.insert_many([
        { '_id': 1, 'category': 'Electronics', 'day': 1, 'product': 'Laptop' },
        { '_id': 2, 'category': 'Electronics', 'day': 2, 'product': 'Mouse' },
        { '_id': 3, 'category': 'Electronics', 'day': 3, 'product': 'Keyboard' },
        { '_id': 4, 'category': 'Furniture', 'day': 1, 'product': 'Desk' },
        { '_id': 5, 'category': 'Furniture', 'day': 2, 'product': 'Chair' }
    ])
    window_result = list(purchases.aggregate([
        {
            '$setWindowFields': {
                'partitionBy': '$category',
                'sortBy': { 'day': 1 },
                'output': {
                    'productsSoFar': {
                        '$push': '$product',
                        'window': { 'documents': ['unbounded', 'current'] }
                    }
                }
            }
        }
    ]))
    print('Window result:', window_result)

    client.close()
```

```
example()
```

$project

The $project operator in Amazon DocumentDB allows you to selectively include or exclude fields from output documents, pass values to the next pipeline stage, and compute new fields from input document values.

Parameters

- field: The field to include or exclude from the output documents, it can be a field path (e.g., "a.b.c").
- 1 or true: Includes the field in the output.
- 0 or false: Excludes the field from the output.

Example (MongoDB Shell)

The following example demonstrates the usage of the $project operator on students collection

Create sample documents

```
db.students.insertMany([
  { "_id": 1, "name": "Alejandro Rosalez", "math": 85, "science": 92, "grade": "A" },
  { "_id": 2, "name": "Carlos Salazar", "math": 78, "science": 84, "grade": "B" },
  { "_id": 3, "name": "Nikhil Jayashankar", "math": 95, "science": 89, "grade": "A" },
  { "_id": 4, "name": "Shirley Rodriguez", "math": 72, "science": 76, "grade": "B" }
  ]);
```

This query includes only the name and math fields in the output. The _id field is included by default unless explicitly excluded.

```
db.students.aggregate([
  { $project: { "name": 1, "math": 1 } }
])
```

Output

```
{ _id: 1, name: "Alejandro Rosalez", math: 85 }
```

```
{ _id: 2, name: "Carlos Salazar", math: 78 }
{ _id: 3, name: "Nikhil Jayashankar", math: 95 }
{ _id: 4, name: "Shirley Rodriguez", math: 72 }
```

This query excludes the `grade` and `_id` fields from the output, showing all other fields (`name`, `math`, `science`).

```
db.students.aggregate([
  { $project: { "grade": 0, "_id": 0 } }
])
```

Output

```
{ name: "Alejandro Rosalez", math: 85, science: 92 }
{ name: "Carlos Salazar", math: 78, science: 84 }
{ name: "Nikhil Jayashankar", math: 95, science: 89 }
{ name: "Shirley Rodriguez", math: 72, science: 76 }
```

Code examples

To view a code example for using the `$project` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('students');

  const result = await collection.aggregate([
    { $project: { "name": 1, "math": 1 } }
  ]).toArray();
  console.log(result);

  await client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['students']

    result = list(collection.aggregate([
        { '$project': { 'name': 1, 'math': 1 } }
    ]))
    print(result)

    client.close()

example()
```

$radiansToDegrees

New from version 8.0.1.

The `$radiansToDegrees` operator in Amazon DocumentDB converts a value from radians to degrees.

Parameters

- `expression`: An expression that resolves to a numeric value in radians.

Example (MongoDB Shell)

The following example shows how to use the `$radiansToDegrees` operator to convert radian values to degrees.

Create sample documents

```
db.angles.insertMany([
  { "_id": 1, "radians": 0 },
  { "_id": 2, "radians": 1.5707963267948966 },
  { "_id": 3, "radians": 3.141592653589793 },
  { "_id": 4, "radians": 6.283185307179586 }
]);
```

Query example

```
db.angles.aggregate([
  { $project: {
    "degrees": { $radiansToDegrees: "$radians" }
  }}
]);
```

Output

```
[
  { "_id": 1, "degrees": 0 },
  { "_id": 2, "degrees": 90 },
  { "_id": 3, "degrees": 180 },
  { "_id": 4, "degrees": 360 }
]
```

Code examples

To view a code example for using the `$radiansToDegrees` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('angles');

  const result = await collection.aggregate([
```

```
    { $project: {
      "degrees": { $radiansToDegrees: "$radians" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['angles']

    result = list(collection.aggregate([
        { '$project': {
            'degrees': { '$radiansToDegrees': '$radians' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$rand

New from version 8.0

The $rand operator in Amazon DocumentDB is used to generate a random number between 0 and 1.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the $rand operator to randomly select two documents from the temp collection.

Create sample documents

```
db.items.insertMany([
  { "name": "pencil", "quantity": 110 },
  { "name": "pen", "quantity": 159 }
])
```

Query example

```
db.items.aggregate([
  {
    $project: {
      randomValue: { $rand: {} }
    }
  }
])
```

Output

```
[
  {
    _id: ObjectId('6924a5edd66dcae121d29517'),
    randomValue: 0.8615243955294392
  },
  {
    _id: ObjectId('6924a5edd66dcae121d29518'),
    randomValue: 0.22815483022099903
  }
]
```

Code examples

To view a code example for using the $rand command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('items');

  const result = await collection.aggregate([
    {
      $project: {
        randomValue: { $rand: {} }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
  client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
  db = client['test']
  collection = db['items']

  result = list(collection.aggregate([
      {
        "$project": {
          "randomValue": { "$rand": {} }
        }
      }
```

```
  ]))

  print(result)

  client.close()

example()
```

$range

The $range aggregation operator in Amazon DocumentDB is used to create an array of consecutive numbers within a specified range. This operator is particularly useful for generating sequences of numbers, such as mile markers for aid stations in a race, as demonstrated in the following examples.

Parameters

- start: The starting value for the range.
- end: The ending value for the range.
- step: (optional) The step value to use when generating the range. If not provided, the default step value is 1.

Example (MongoDB Shell)

In this example, we'll use the $range operator to generate the mile markers for water stations in a bicycle race.

Create sample documents

```
db.races.insertMany([
  { _id: 0, race: "STP", distance: 206 },
  { _id: 1, race: "RSVP", distance: 160 },
  { _id: 2, race: "Chilly Hilly", distance: 33 },
  { _id: 3, race: "Flying Wheels", distance: 100 }
]);
```

Query example

```
db.races.aggregate([
```

```
  {
    $project: {
      race: 1,
      "waterStations": { $range: [20, "$distance", 20] }
    }
  }
]);
```

Output

```
[
  {
    _id: 0,
    race: 'STP',
    waterStations: [
       20,  40,  60,  80,
      100, 120, 140, 160,
      180, 200
    ]
  },
  {
    _id: 1,
    race: 'RSVP',
    waterStations: [
       20,  40,  60, 80,
      100, 120, 140
    ]
  },
  { _id: 2, race: 'Chilly Hilly', waterStations: [ 20 ] },
  { _id: 3, race: 'Flying Wheels', waterStations: [ 20, 40, 60, 80 ] }
]
```

Code examples

To view a code example for using the `$range` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('races');

    const pipeline = [
      {
        $project: {
          race: 1,
          waterStations: { $range: [20, "$distance", 20] }
        }
      }
    ];

    const results = await collection.aggregate(pipeline).toArray();

    console.dir(results, { depth: null });

  } finally {
    await client.close();
  }
}

example().catch(console.error);
```

Python

```
from pymongo import MongoClient

def example():

  client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

  try:
      db = client.test
      collection = db.races
```

```
        pipeline = [
            {
                "$project": {
                    "race": 1,
                    "waterStations": { "$range": [20, "$distance", 20] }
                }
            }
        ]

        results = collection.aggregate(pipeline)

        for doc in results:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

example()
```

$rank

New from version 8.0.2.

The `$rank` window operator in Amazon DocumentDB returns the position of each document within its partition, according to the `$setWindowFields` stage `sortBy` order. Documents with the same `sortBy` value receive the same rank, and the next distinct value skips the intervening positions, leaving gaps (for example, `1, 2, 2, 4`). It is specified under the `output` field of the `$setWindowFields` stage.

It is a window operator that is used only in the `$setWindowFields` stage.

Parameters

- `$rank` takes an empty document `{}` as its value.
- It does not accept a `window` frame.
- It requires exactly one `sortBy` key in the `$setWindowFields` stage.

Example (MongoDB Shell)

The following example partitions the documents by `game`, sorts each partition by `score` in descending order, and assigns each player a rank.

Create sample documents

```
db.players.insertMany([
  { _id: 1, game: "A", player: "Alice", score: 90 },
  { _id: 2, game: "A", player: "Bob", score: 85 },
  { _id: 3, game: "A", player: "Carol", score: 85 },
  { _id: 4, game: "A", player: "Dave", score: 70 },
  { _id: 5, game: "B", player: "Erin", score: 95 },
  { _id: 6, game: "B", player: "Frank", score: 80 }
]);
```

Query example

```
db.players.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$game",
      sortBy: { score: -1 },
      output: {
        rank: { $rank: {} }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "game": "A", "player": "Alice", "score": 90, "rank": 1 },
  { "_id": 2, "game": "A", "player": "Bob", "score": 85, "rank": 2 },
  { "_id": 3, "game": "A", "player": "Carol", "score": 85, "rank": 2 },
  { "_id": 4, "game": "A", "player": "Dave", "score": 70, "rank": 4 },
  { "_id": 5, "game": "B", "player": "Erin", "score": 95, "rank": 1 },
  { "_id": 6, "game": "B", "player": "Frank", "score": 80, "rank": 2 }
]
```

Within each `game` partition, players are ranked by descending `score`. Bob and Carol tie at 85 and share rank 2, so the next rank skips to 4 for Dave.

Code examples

To view a code example for using the `$rank` operator in the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const players = db.collection('players');

    await players.insertMany([
      { _id: 1, game: "A", player: "Alice", score: 90 },
      { _id: 2, game: "A", player: "Bob", score: 85 },
      { _id: 3, game: "A", player: "Carol", score: 85 },
      { _id: 4, game: "A", player: "Dave", score: 70 },
      { _id: 5, game: "B", player: "Erin", score: 95 },
      { _id: 6, game: "B", player: "Frank", score: 80 }
    ]);

    const result = await players.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$game",
          sortBy: { score: -1 },
          output: {
            rank: { $rank: {} }
          }
        }
      }
    ]).toArray();
```

```
    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        players = db['players']

        players.insert_many([
            { '_id': 1, 'game': 'A', 'player': 'Alice', 'score': 90 },
            { '_id': 2, 'game': 'A', 'player': 'Bob', 'score': 85 },
            { '_id': 3, 'game': 'A', 'player': 'Carol', 'score': 85 },
            { '_id': 4, 'game': 'A', 'player': 'Dave', 'score': 70 },
            { '_id': 5, 'game': 'B', 'player': 'Erin', 'score': 95 },
            { '_id': 6, 'game': 'B', 'player': 'Frank', 'score': 80 }
        ])

        result = list(players.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$game',
                    'sortBy': { 'score': -1 },
                    'output': {
                        'rank': { '$rank': {} }
                    }
                }
            }
        ]))

        print(result)
```

```
    finally:
        client.close()

example()
```

$redact

The `$redact` aggregation stage in Amazon DocumentDB is used to selectively include or exclude content from the output documents based on the values of specified fields. This is particularly useful in scenarios where you need to control the visibility of sensitive data based on access levels or user permissions.

Parameters

- `$cond`: An expression that evaluates to either `$$KEEP`, `$$PRUNE`, or `$$DESCEND` for each field in the document.
- `$$KEEP`: Retains the current field in the output document.
- `$$PRUNE`: Removes the current field from the output document.
- `$$DESCEND`: Recursively applies the `$redact` stage to the current field, which is an object or array.

Example (MongoDB Shell)

In this example, we'll use the `$redact` stage to filter orders based on their status, showing only orders with specific status values.

Create sample documents

```
db.orders.insert([
  { "_id": 1, "status": "shipped", "customer": "Carlos Salazar", "total": 150.00,
 "date": "2025-01-15" },
  { "_id": 2, "status": "processing", "customer": "Saanvi Sarkar", "total": 89.99,
 "date": "2025-01-20" },
  { "_id": 3, "status": "cancelled", "customer": "Zhang Wei", "total": 220.50, "date":
 "2025-01-18" }
])
```

Query example

```
db.orders.aggregate([
  {
    $redact: {
      $cond: {
        if: { $in: ["$status", ["shipped", "processing"]] },
        then: "$$KEEP",
        else: "$$PRUNE"
      }
    }
  }
])
```

Output

```
[
  { _id: 1, status: 'shipped', customer: 'Carlos Salazar', total: 150, date:
 '2025-01-15' },
  { _id: 2, status: 'processing', customer: 'Saanvi Sarkar', total: 89.99, date:
 '2025-01-20' }
]
```

In this example, the `$redact` stage checks the value of the `status` field in each document. If the `status` is "shipped" or "processing", the document is kept ($$KEEP). Otherwise, the document is pruned ($$PRUNE).

Code examples

To view a code example for using the `$redact` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('orders');

  const result = await collection.aggregate([
```

```
    {
      $redact: {
        $cond: {
          if: { $in: ['$status', ['shipped', 'processing']] },
          then: '$$KEEP',
          else: '$$PRUNE'
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['orders']

    result = list(collection.aggregate([
        {
            '$redact': {
                '$cond': {
                    'if': { '$in': ['$status', ['shipped', 'processing']] },
                    'then': '$$KEEP',
                    'else': '$$PRUNE'
                }
            }
        }
    ]))

    print(result)
    client.close()
```

```
example()
```

$reduce

The `$reduce` aggregation operator in Amazon DocumentDB is used to apply a function of two arguments cumulatively to the elements of an array to reduce the array to a single value. This operator is particularly useful for performing complex calculations or transformations on array data within the aggregation pipeline.

Parameters

- `input`: The array to be reduced.
- `initialValue`: The initial value to be used in the reduction operation.
- `in`: The expression to be evaluated on each element of the `input` array. This expression should return a value that will be used in the next iteration of the reduction.

Example (MongoDB Shell)

The following example demonstrates how to use the `$reduce` operator to calculate the sum of all elements in an array.

Create sample documents

```
db.orders.insertMany([
  { "_id": 1, "items": [1, 2, 3, 4, 5] },
  { "_id": 2, "items": [10, 20, 30] },
  { "_id": 3, "items": [5, 15, 25, 35] },
  { "_id": 4, "items": [100, 200] }
])
```

Query example

```
db.orders.aggregate([
  {
    $project: {
      total: {
        $reduce: {
          input: "$items",
```

```
          initialValue: 0,
          in: { $add: ["$$value", "$$this"] }
        }
      }
    }
  }
])
```

Output

```
[
  { "_id": 1, "total": 15 },
  { "_id": 2, "total": 60 },
  { "_id": 3, "total": 80 },
  { "_id": 4, "total": 300 }
]
```

The `$reduce` operator iterates over the `items` array, adding each element to the `initialValue` of 0. The result is the sum of all elements in the array.

Code examples

To view a code example for using the `$reduce` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$reduce` operator in a Node.js application:

```
const { MongoClient } = require("mongodb");

async function main() {
  const client = await MongoClient.connect("mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false");
  const db = client.db("test");
  const orders = db.collection("orders");

  const result = await orders.aggregate([
    {
      $project: {
        total: {
```

```
          $reduce: {
            input: "$items",
            initialValue: 0,
            in: { $add: ["$$value", "$$this"] }
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

Here's an example of using the $reduce operator in a python application:

```
from pymongo import MongoClient

def main():
    client = MongoClient("mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false")
    db = client["test"]
    orders = db["orders"]

    result = list(orders.aggregate([
        {
            "$project": {
                "total": {
                    "$reduce": {
                        "input": "$items",
                        "initialValue": 0,
                        "in": { "$add": ["$$value", "$$this"] }
                    }
                }
            }
        }
    ]))

    print(result)
```

```
    client.close()

if __name__ == "__main__":
    main()
```

$regexFind

New from version 5.0.

Not supported by Elastic cluster.

The `$regexFind` operator in Amazon DocumentDB is used to perform regular expression matching on string fields within documents. It allows you to search for and extract specific substrings that match a given regular expression pattern.

Parameters

- `input`: The string field or expression to search.
- `regex`: The regular expression pattern to match.
- `options`: (optional) An object that specifies optional parameters for the regular expression, such as case-sensitivity and multi-line matching. Supported options are `i` (case-insensitive) and `m` (multi-line).

Example (MongoDB Shell)

The following example demonstrates how to use the `$regexFind` operator to search for documents where the `name` field matches a specific regular expression pattern.

Create sample documents

```
db.users.insertMany([
  { "_id": 1, name: "John Doe", email: "john@example.com" },
  { "_id": 2, name: "Diego Ramirez", email: "diego@example.com" },
  { "_id": 3, name: "Alejandro Rosalez", email: "alejandro@example.com" },
  { "_id": 4, name: "Shirley Rodriguez", email: "shirley@example.com" }
]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      names: {
        $regexFind: { input: '$name', regex: 'j', options: 'i' }
      }
    }
  },
  { $match: {names: {$ne: null}}}
])
```

This query will return all documents where the name field contains the letter "j" (case-insensitive).

Output

```
[
  { _id: 1, names: { match: 'J', idx: 0, captures: [] } }
]
```

Note: If your query is using Amazon DocumentDB planner version 1, you must use a hint to utilize an index. Without a hint, the query may perform a collection scan. To check your planner version and learn more about using hints, see Query planner v2.

Code examples

To view a code example for using the `$regexFind` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const results = await users.aggregate([
    { $project: { names: { $regexFind: { input: "$name", regex: "john", options:
 "i" }}}},
```

```
    { $match: {names: {$ne: null}}}
  ]).toArray();


  console.log(results);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
users = db['users']

results = list(users.aggregate([
    { "$project": { "names": { "$regexFind": { "input": "$name", "regex": "john",
 "options": "i" }}}},
    { "$match": {"names": {"$ne": None}}}
]))

print(results)

client.close()
```

$regexFindAll

Introduced in 5.0

The `$regexFindAll` operator in Amazon DocumentDB is used to perform regular expression matching on string fields within documents. It allows you to search for and extract specific substrings that match a given regular expression pattern, returning all matches of the regular expression.

Parameters

- `input`: The string field or expression to search.
- `regex`: The regular expression pattern to match.
- `options`: (optional) An object that specifies optional parameters for the regular expression, such as case-sensitivity and multi-line matching. Supported options are `i` (case-insensitive) and `m` (multi-line).

Example (MongoDB Shell)

The following example demonstrates how to use the `$regexFindAll` operator to extract all letter sequences from the `email` field.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "John Doe", email: "john@example.com", phone: "555-1234" },
  { _id: 2, name: "Jane Roe", email: "jane@example.com", phone: "555-5678" },
  { _id: 3, name: "Carlos Salazar", email: "carlos@example.com", phone: "555-3456" },
  { _id: 4, name: "Saanvi Sarkar", email: "saanvi@example.com", phone: "555-7890" }

]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      name: 1,
      emailMatches: {
        $regexFindAll: { input: '$email', regex: '[a-z]+', options: 'i' }
      }
    }
  }
])
```

Output

```
[
  {
    _id: 1,
    name: 'John Doe',
```

```
    emailMatches: [
      { match: 'john', idx: 0, captures: [] },
      { match: 'example', idx: 5, captures: [] },
      { match: 'com', idx: 13, captures: [] }
    ]
  },
  {
    _id: 2,
    name: 'Jane Roe',
    emailMatches: [
      { match: 'jane', idx: 0, captures: [] },
      { match: 'example', idx: 5, captures: [] },
      { match: 'com', idx: 13, captures: [] }
    ]
  },
  {
    _id: 3,
    name: 'Carlos Salazar',
    emailMatches: [
      { match: 'carlos', idx: 0, captures: [] },
      { match: 'example', idx: 7, captures: [] },
      { match: 'com', idx: 15, captures: [] }
    ]
  },
  {
    _id: 4,
    name: 'Saanvi Sarkar',
    emailMatches: [
      { match: 'saanvi', idx: 0, captures: [] },
      { match: 'example', idx: 7, captures: [] },
      { match: 'com', idx: 15, captures: [] }
    ]
  }
]
```

Note: If your query is using Amazon DocumentDB planner version 1, you must use a hint to utilize an index. Without a hint, the query may perform a collection scan. To check your planner version and learn more about using hints, see Query planner v2.

Code examples

To view a code example for using the `$regexFindAll` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$regexFind` operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const results = await users.aggregate([
    {
      $project: {
        name: 1,
        emailMatches: {
          $regexFindAll: { input: "$email", regex: "[a-z]+", options: "i" }
        }
      }
    }
  ]).toArray();

  console.log(JSON.stringify(results, null, 2));

  await client.close();
}

main();
```

Python

Here's an example of using the `$regexFind` operator in a Python application:

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
users = db['users']
```

```
results = list(users.aggregate([
    {
        "$project": {
            "name": 1,
            "emailMatches": {
                "$regexFindAll": {
                    "input": "$email",
                    "regex": "[a-z]+",
                    "options": "i"
                }
            }
        }
    }
]))

print(results)

client.close()
```

$regexMatch

New from version 5.0. Not supported by Elastic cluster.

The `$regexMatch` operator in Amazon DocumentDB is used to perform regular expression matching on string fields. It returns a boolean value (`true` or `false`) indicating whether the input string matches the specified pattern.

Parameters

- `input`: The string to test against the regular expression.
- `regex`: The regular expression pattern to match.
- `options`: (Optional) Flags to modify the regular expression behavior, such as case-insensitive matching (`i`) or multiline matching (`m`).

Example (MongoDB Shell)

The following example demonstrates how to use the `$regexMatch` operator to check if names start with the letter 'M'. The operator returns `true` or `false` for each document.

Create sample documents

```
db.users.insertMany([
  { "_id":1, name: "María García", email: "maría@example.com" },
  { "_id":2, name: "Arnav Desai", email: "arnav@example.com" },
  { "_id":3, name: "Martha Rivera", email: "martha@example.com" },
  { "_id":4, name: "Richard Roe", email: "richard@example.com" },

]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      name: 1,
      startsWithM: {
        $regexMatch: {
          input: "$name",
          regex: "^M",
          options: "i"
        }
      }
    }
  }
]);
```

Output

```
{ _id: 1, name: 'María García', startsWithM: true },
{ _id: 2, name: 'Arnav Desai', startsWithM: false },
{ _id: 3, name: 'Martha Rivera', startsWithM: true },
{ _id: 4, name: 'Richard Roe', startsWithM: false }
```

Code examples

To view a code example for using the `$regexMatch` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function checkNamePattern() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const result = await collection.aggregate([
    {
      $project: {
        name: 1,
        startsWithM: {
          $regexMatch: {
            input: "$name",
            regex: "^M",
            options: "i"
          }
        }
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

checkNamePattern();
```

Python

```
from pymongo import MongoClient

def check_name_pattern():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.users

    result = list(collection.aggregate([
        {
            '$project': {
```

```
                'name': 1,
                'startsWithM': {
                    '$regexMatch': {
                        'input': '$name',
                        'regex': '^M',
                        'options': 'i'
                    }
                }
            }
        }
    ]))

    print(result)

    client.close()

check_name_pattern()
```

$replaceAll

Introduced in 5.0

The `$replaceAll` operator in Amazon DocumentDB is used to replace all occurrences of a specified string pattern within a field with a new string. This operator can be useful for tasks such as data normalization, text cleaning, and string manipulation.

Parameters

- `input`: The field or expression containing the string to be replaced.
- `find`: The string pattern to search for and replace.
- `replacement`: The string to replace the matched occurrences with.

Example (MongoDB Shell)

The following example demonstrates how to use the `$replaceAll` operator in an aggregation pipeline to replace all occurrences of the string "Chocolatier" with "Chocolate Co." in the "brandName" field of a "products" collection.

Create sample documents

```
db.products.insertMany([
  {
    "_id": 1,
    "productId": "PROD-0Y9GL0",
    "brandName": "Gordon's Chocolatier",
    "category": "CPG",
    "rating": {
      "average": 4.8
    }
  },
  {
    "_id": 2,
    "productId": "PROD-1X2YZ3",
    "brandName": "Premium Chocolatier",
    "category": "CPG",
    "rating": {
      "average": 4.5
    }
  },
  {
    "_id": 3,
    "productId": "PROD-Y2E9H5",
    "name": "Nutrition Co. - Original Corn Flakes Cereal",
    "category": "Breakfast Cereals",
    "price": 8.5
  }
]);
```

Query example

```
db.products.aggregate([
  {
    $addFields: {
      "brandName": {
        $replaceAll: {
          input: "$brandName",
          find: "Chocolatier",
          replacement: "Chocolate Co."
        }
      }
    }
  }
```

```
])
```

Output

```
[
  {
    _id: 1,
    productId: 'PROD-0Y9GL0',
    brandName: "Gordon's Chocolate Co.",
    category: 'CPG',
    rating: { average: 4.8 }
  },
  {
    _id: 2,
    productId: 'PROD-1X2YZ3',
    brandName: 'Premium Chocolate Co.',
    category: 'CPG',
    rating: { average: 4.5 }
  },
  {
    _id: 3,
    productId: 'PROD-Y2E9H5',
    name: 'Nutrition Co. - Original Corn Flakes Cereal',
    category: 'Breakfast Cereals',
    price: 8.5,
    brandName: null
  }
]
```

Code examples

To view a code example for using the `$replaceAll` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function replaceAll() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const collection = db.collection('products');

  const results = await collection.aggregate([
    {
      $addFields: {
        "brandName": {
          $replaceAll: {
            input: "$brandName",
            find: "Chocolatier",
            replacement: "Chocolate Co."
          }
        }
      }
    }
  ]).toArray();

  console.log(results);

  await client.close();
}

replaceAll();
```

Python

```
from pymongo import MongoClient

def replace_all():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.products

    results = list(collection.aggregate([
        {
            "$addFields": {
                "brandName": {
                    "$replaceAll": {
                        "input": "$brandName",
                        "find": "Chocolatier",
                        "replacement": "Chocolate Co."
                    }
```

```
                }
            }
        }
    ]))

    print(results)

    client.close()

replace_all()
```

$replaceOne

Introduced in 5.0

The `$replaceOne` operator in Amazon DocumentDB is a string expression operator used within aggregation pipelines to replace the first occurrence of a specified substring within a string with a replacement string. This operator is case-sensitive and only replaces the first match found.

Parameters

- `input`: The string (field) on which to perform the find.
- `find`: The string to search for within the input.
- `replacement`: The string to replace the first occurrence of the find in the input(field).

Example (MongoDB Shell)

The following example demonstrates how to use the `$replaceOne` operator within an aggregation pipeline to replace substrings in product names.

Create sample documents

```
db.products.insertMany([
  { "_id":1, "productId": "PROD-0Y9GL0", "name": "Gordon's Extra Creamy Milk Chocolate
 - Pack of 4", "category": "Confectionery", "price": 24.99 },
  { "_id":2, "productId": "PROD-Y2E9H5", "name": "Nutrition Co. - Original Corn Flakes
 Cereal", "category": "Breakfast Cereals", "price": 8.50 },
  { "_id":3, "productId": "PROD-Z3F8K2", "name": "Gordon's Dark Chocolate (90% Cocoa)
 Pack - Pack of 4", "category": "Confectionery", "price": 28.99 }
```

```
]);
```

Aggregation example

```
db.products.aggregate([
  {
    $addFields: {
      standardizedName: {
        $replaceOne: {
          input: "$name",
          find: "Pack",
          replacement: "Package"
        }
      }
    }
  }
]);
```

Output

The output shows that only the first occurrence of "Pack" in each product name was replaced with "Package".

```
[
  {
    _id: 1,
    productId: 'PROD-0Y9GL0',
    name: "Gordon's Extra Creamy Milk Chocolate - Pack of 4",
    category: 'Confectionery',
    price: 24.99,
    standardizedName: "Gordon's Extra Creamy Milk Chocolate - Package of 4"
  },
  {
    _id: 2,
    productId: 'PROD-Y2E9H5',
    name: 'Nutrition Co. - Original Corn Flakes Cereal',
    category: 'Breakfast Cereals',
    price: 8.5,
    standardizedName: 'Nutrition Co. - Original Corn Flakes Cereal'
  },
  {
    _id: 3,
```

```
  productId: 'PROD-Z3F8K2',
  name: "Gordon's Dark Chocolate (90% Cocoa) Pack - Pack of 4",
  category: 'Confectionery',
  price: 28.99,
  standardizedName: "Gordon's Dark Chocolate (90% Cocoa) Package - Pack of 4"
}
```

Code examples

To view a code example for using the $replaceOne command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function replaceOne() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const pipeline = [
    {
      $addFields: {
        standardizedName: {
          $replaceOne: {
            input: '$name',
            find: 'Pack',
            replacement: 'Package'
          }
        }
      }
    }
  ];

  const result = await collection.aggregate(pipeline).toArray();
  console.log(result);

  await client.close();
}
```

```
replaceOne();
```

Python

```
from pymongo import MongoClient

def replaceOne():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    pipeline = [
        {
            '$addFields': {
                'standardizedName': {
                    '$replaceOne': {
                        'input': '$name',
                        'find': 'Pack',
                        'replacement': 'Package'
                    }
                }
            }
        }
    ]

    result = list(collection.aggregate(pipeline))
    print(result)

    client.close()

replaceOne()
```

$replaceRoot

The `$replaceRoot` operator is used to replace the root document with the specified embedded document. This is useful when you want to promote a nested document to the top level or restructure your data output.

Parameters

- newRoot: The new root document that will replace the existing root document.

Example (MongoDB Shell)

This example shows how to extract shipping address information from customer orders, which is useful for generating shipping labels or address lists.

Create sample documents

```
db.orders.insertMany([
  {
    "_id":1, "orderId": "ORD-2024-001", "customerId": "CUST-12345", "orderDate":
 "2024-01-15", "shippingAddress": { "name": "María García", "street": "123
 Main St", "city": "Seattle", "state": "WA", "zipCode": "98101", "country":
 "USA" },"totalAmount": 149.99 },
  { "_id":2, "orderId": "ORD-2024-002", "customerId": "CUST-67890", "orderDate":
 "2024-01-16", "shippingAddress": { "name": "Arnav Desai", "street": "456 Oak
 Ave", "city": "Portland", "state": "OR", "zipCode": "97201", "country": "USA" },
 "totalAmount": 89.50 } ])
```

Query example

```
db.orders.aggregate([
  {
    $replaceRoot: {
      newRoot: "$shippingAddress"
    }
  }
])
```

Output

```
{
    name: 'María García',
    street: '123 Main St',
    city: 'Seattle',
    state: 'WA',
    zipCode: '98101',
    country: 'USA'
  },
  {
    name: 'Arnav Desai',
```

```
    street: '456 Oak Ave',
    city: 'Portland',
    state: 'OR',
    zipCode: '97201',
    country: 'USA'
  }
```

Code examples

To view a code example for using the `$replaceRoot` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function extractShippingAddresses() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('orders');

  const result = await collection.aggregate([
    {
      $replaceRoot: {
        newRoot: "$shippingAddress"
      }
    }
  ]).toArray();

  console.log(result);

  await client.close();
}

extractShippingAddresses();
```

Python

```
from pymongo import MongoClient

def extract_shipping_addresses():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['orders']

    result = list(collection.aggregate([
        {
            "$replaceRoot": {
                "newRoot": "$shippingAddress"
            }
        }
    ]))

    print(result)

    client.close()

extract_shipping_addresses()
```

$replaceWith

New from version 8.0

Not supported by Elastic cluster.

The `$replaceWith` aggregation stage in Amazon DocumentDB is used to replace the input document with a new document. All existing fields on the input document, including the _id field, are replaced by the new document. `$replaceWith` is commonly used to flatten documents, or promote an embedded document to the top level.

Parameters

- `<replacement>` (required): The new document that will replace the existing document.

Example (MongoDB Shell)

The following example demonstrates how to use the `$replaceWith` operator to replace an existing document in an Amazon DocumentDB collection.

Create sample documents

```
db.restaurants.insertMany([
  {
    "restaurantId": "REST-0Y9GL0",
    "name": "Biryani Adda",
    "cuisine": "Indian",
    "ratings": [ 3, 4, 3, 2, 2, 4, 1, 5, 5, 5 ]
  },
  {
    "restaurantId": "REST-8L2PX9",
    "name": "The Burger Spot",
    "cuisine": "American",
    "ratings": [ 2, 3, 4, 5, 3, 1, 1, 2, 4 ]
  }
]);
```

Query example

```
db.restaurants.aggregate([
  { $replaceWith: {
      name: "$name",
      cuisine: "$cuisine",
      rating: { $avg: "$ratings" }
    }
  }
]);
```

Output

```
[
  {
    name: 'Biryani Adda',
    cuisine: 'Indian',
    rating: 3.4
  },
  {
    name: 'The Burger Spot',
    cuisine: 'American',
    rating: 2.7777777777777777
  }
]
```

Code examples

To view a code example for using the `$replaceWith` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function replaceDoc() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('restaurants');

  const result = await collection.aggregate([
    { $replaceWith: {
        name: "description_index",
        cuisine: 2,
        rating: { $avg: "$ratings" }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

replaceDoc();
```

Python

```
from pymongo import MongoClient


def replace_document():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.restaurants
```

```
    result = list(collection.aggregate([
        {
            '$replaceWith': {
                'name': "$name",
                'cuisine': "$cuisine",
                'rating': { '$avg': "$ratings"}
            }
        }
    ]))

    print(result)
    client.close()

replace_document()
```

$reverseArray

The `$reverseArray` operator in Amazon DocumentDB is used to reverse the elements of an array in the specified order. This operator is useful when you need to reorder the elements of an array in the reverse direction.

Parameters

- `expression`: The array expression to reverse.

Example (MongoDB Shell)

The following example demonstrates how to use the `$reverseArray` operator to reverse the order of elements in an array.

Create sample documents

```
db.miles.insertMany([
  { "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "flight_miles" : [ 1205, 2560, 880 ]},
  { "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 2560, 890, 2780]},
  { "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 880]}
]);
```

Query example

```
db.miles.aggregate([
  {
    $project: {
      _id: 1,
      member_since: 1,
      credit_card: 1,
      reversed_flight_miles: { $reverseArray: "$flight_miles" }
    }
  }
]);
```

Output

```
{ "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "reversed_flight_miles" : [ 880, 2560, 1205 ] }
{ "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "reversed_flight_miles" : [ 2780, 890, 2560, 1205 ] }
{ "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "reversed_flight_miles" : [ 880, 1205 ] }
```

In this example, the `$reverseArray` operator is used to reverse the order of the `flight_miles` array. The resulting `reversed_flight_miles` field in the output shows the elements of the array in the reversed order.

Code examples

To view a code example for using the `$reverseArray` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$reverseArray` operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function reverseArray() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const collection = db.collection('miles');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        member_since: 1,
        credit_card: 1,
        reversed_flight_miles: { $reverseArray: '$flight_miles' }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

reverseArray();
```

Python

Here's an example of using the `$reverseArray` operator in a Python application:

```
from pymongo import MongoClient

def reverse_array():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.miles

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'member_since': 1,
                'credit_card': 1,
                'reversed_flight_miles': { '$reverseArray': '$flight_miles' }
            }
        }
    ]))

    print(result)
```

```
    client.close()

reverse_array()
```

$round

New from version 8.0.1.

The $round operator in Amazon DocumentDB rounds a number to a specified decimal place. When a value falls exactly halfway between the two nearest values at that place, $round rounds to the nearest even digit. For example, 2.5 rounds to 2, and 3.5 rounds to 4.

Parameters

- number: An expression that resolves to a number.
- place: Optional. An integer expression between -20 and 100 that specifies the number of decimal places to round to. A negative value rounds to the left of the decimal. Defaults to 0.

Example (MongoDB Shell)

The following example shows how to use the $round operator to round numeric values to one decimal place.

Rounding with a positive place value

```
db.measurements.insertMany([
  {_id: 1, value: 3.456},
  {_id: 2, value: 7.891},
  {_id: 3, value: 12.345}
]);
```

Query example

```
db.measurements.aggregate([
  { $project: { rounded: { $round: ["$value", 1] } } }
]);
```

Output

```
[
```

```
  {_id: 1, rounded: 3.5},
  {_id: 2, rounded: 7.9},
  {_id: 3, rounded: 12.3}
]
```

Rounding halfway values to the nearest even digit

When a value falls exactly halfway between the two nearest values at the target place, `$round` rounds to the nearest even digit rather than always rounding up. The following example rounds halfway values to whole integers.

```
db.halfway.insertMany([
  {_id: 1, value: 10.5},
  {_id: 2, value: 11.5},
  {_id: 3, value: 12.5},
  {_id: 4, value: 13.5}
]);
```

```
db.halfway.aggregate([
  { $project: { rounded: { $round: ["$value", 0] } } }
]);
```

Output

```
[
  {_id: 1, rounded: 10},
  {_id: 2, rounded: 12},
  {_id: 3, rounded: 12},
  {_id: 4, rounded: 14}
]
```

Because `10.5` is exactly halfway between `10` and `11`, it rounds to the nearest even value `10`. Similarly, `11.5` and `12.5` both round to the nearest even value `12`, and `13.5` rounds to `14`.

Rounding with a negative place value

When `place` is negative, `$round` rounds to the left of the decimal. The following example rounds values using the first digit to the left of the decimal.

```
db.samples.insertMany([
```

```
  {_id: 1, value: 19.25},
  {_id: 2, value: 28.73},
  {_id: 3, value: 34.32}
]);
```

```
db.samples.aggregate([
  { $project: { rounded: { $round: ["$value", -1] } } }
]);
```

Output

```
[
  {_id: 1, rounded: 20},
  {_id: 2, rounded: 30},
  {_id: 3, rounded: 30}
]
```

Code examples

To view a code example for using the $round operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('measurements');
    const result = await collection.aggregate([
      { $project: { rounded: { $round: ["$value", 1] } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['measurements']
        result = list(collection.aggregate([
            {'$project': {'rounded': {'$round': ['$value', 1]}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$rtrim

New from version 4.0.

Not supported by Elastic cluster.

The `$rtrim` operator in Amazon DocumentDB is used to remove trailing characters from a string. By default, it removes trailing whitespace characters, but you can also specify a set of characters to remove by passing the chars argument.

Parameters

- `input`: The input string from which to remove trailing whitespace characters.
- `chars`: (optional) To remove specific characters.

Example (MongoDB Shell)

The following example demonstrates how to use the `$rtrim` operator to remove trailing specified characters (" *") from a string.

Create sample documents

```
db.collection.insert([
  { "name": "John Doe*  ", "age": 30 },
  { "name": "Jane Smith  ", "age": 25 },
  { "name": "Bob Johnson", "age": 35 }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      _id: 0,
      name: { $rtrim: { input: "$name", chars: " *" } },
      age: 1
    }
  }
]);
```

Output

```
[
  { age: 30, name: 'John Doe' },
  { age: 25, name: 'Jane Smith' },
  { age: 35, name: 'Bob Johnson' }
]
```

Code examples

To view a code example for using the `$rtrim` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient, Bson } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  try {

    const db = client.db('test');
    const collection = db.collection('collection');

    const pipeline = [
      {
        $project: {
          _id: 0,
          name: { $rtrim: { input: "$name", chars: " *" } },
          age: 1
        }
      }
    ];

    const results = await collection.aggregate(pipeline).toArray();
    console.dir(results, { depth: null });

  } catch (err) {
    console.error('Error occurred:', err);
  } finally {
    await client.close();
  }
}

example().catch(console.error);
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:

        db = client.test
        collection = db.collection

        pipeline = [
```

```
            {
                "$project": {
                    "_id": 0,
                    "name": { "$rtrim": { "input": "$name", "chars": " *" } },
                    "age": 1
                }
            }
        ]

        results = collection.aggregate(pipeline)

        for doc in results:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

example()
```

$sample

The `$sample` aggregation stage in Amazon DocumentDB is used to randomly select a specified number of documents from a collection. This is useful for tasks such as data analysis, testing, and generating samples for further processing.

Parameters

- `size`: The number of documents to randomly select.

Example (MongoDB Shell)

The following example demonstrates how to use the `$sample` stage to randomly select two documents from the `temp` collection.

Create sample documents

```
db.temp.insertMany([
```

```
  { "_id": 1, "temperature": 97.1, "humidity": 0.60, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 2, "temperature": 98.2, "humidity": 0.59, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 3, "temperature": 96.8, "humidity": 0.61, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 4, "temperature": 97.9, "humidity": 0.61, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 5, "temperature": 97.5, "humidity": 0.60, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 6, "temperature": 98.0, "humidity": 0.59, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 7, "temperature": 97.2, "humidity": 0.60, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 8, "temperature": 98.1, "humidity": 0.59, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 9, "temperature": 96.9, "humidity": 0.62, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") },
  { "_id": 10, "temperature": 97.7, "humidity": 0.60, "timestamp":
 ISODate("2019-03-21T21:17:22.425Z") }
]);
```

Query example

```
db.temp.aggregate([
   { $sample: { size: 2 } }
])
```

Output

```
{ "_id" : 4, "temperature" : 97.9, "humidity" : 0.61, "timestamp" :
 ISODate("2019-03-21T21:17:22.425Z") }
{ "_id" : 9, "temperature" : 96.9, "humidity" : 0.62, "timestamp" :
 ISODate("2019-03-21T21:17:22.425Z") }
```

As the results show, 2 of the 10 documents were randomly sampled. You can now use these documents to determine an average or to perform min/max calculations.

Code examples

To view a code example for using the `$sample` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function sampleDocuments() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('temp');

  const result = await collection.aggregate([
    { $sample: { size: 2 } }
  ]).toArray();

  console.log(result);
  await client.close();
}

sampleDocuments();
```

Python

```
from pymongo import MongoClient

def sample_documents():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['temp']

    result = list(collection.aggregate([
        { '$sample': { 'size': 2 } }
    ]))

    print(result)
    client.close()

sample_documents()
```

$search

New from version 5.0.

The `$search` operator in Amazon DocumentDB is used to provide text search capabilities.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the `$search` operator to perform a text search query.

Create sample documents

```
db.textcollection.createIndex({"description": "text"});

db.textcollection.insertMany([
  { _id: 1, name: "John Doe", description: "This is a sample document about John
 Doe." },
  { _id: 2, name: "Jane Smith", description: "This is a sample document about Jane
 Smith." },
  { _id: 3, name: "Bob Johnson", description: "This is a sample document about Bob
 Johnson." },
  { _id: 4, name: "Jon Jeffries", description: "This is a sample document about Jon
 Jeffries." }
]);
```

Query example

```
db.textcollection.find(
  { $text: { $search: "John" } }
);
```

Output

```
[
  {
    _id: 1,
    name: 'John Doe',
    description: 'This is a sample document about John Doe.'
```

```
  }
]
```

Code examples

To view a code example for using the `$search` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findWithText() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('textcollection');

  const result = await collection.find(
    { $text: { $search: "John" } }
  ).sort({ score: { $meta: "textScore" } }).toArray();

  console.log(result);
  client.close();
}

findWithText();
```

Python

```
from pymongo import MongoClient

def find_with_text():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['textcollection']

    result = list(collection.find(
        { '$text': { '$search': 'John' } }
    ))
```

```
    print(result)
    client.close()

find_with_text()
```

$second

The $second operator in Amazon DocumentDB extracts the seconds component from a date or timestamp. It is used to retrieve the seconds value from a date or timestamp field.

Parameters

- expression: The date or timestamp field to extract the seconds value from. This expression can be a field path or any valid expression that resolves to a date or timestamp.

Example (MongoDB Shell)

The following example demonstrates how to use the $second operator to extract the seconds component from a date field.

Create sample documents

```
db.users.insertMany([
  { name: "John", dob: new Date("1990-05-15T12:30:45Z") },
  { name: "Jane", dob: new Date("1985-09-20T23:59:59Z") },
  { name: "Bob", dob: new Date("2000-01-01T00:00:00Z") }
]);
```

Query example

```
db.users.aggregate([{ $project: { name: 1, dobSeconds: { $second: "$dob" } } }])
```

Output

```
[
  { "_id" : ObjectId("6089a9c306a829d1f8b456a1"), "name" : "John", "dobSeconds" : 45 },
  { "_id" : ObjectId("6089a9c306a829d1f8b456a2"), "name" : "Jane", "dobSeconds" : 59 },
  { "_id" : ObjectId("6089a9c306a829d1f8b456a3"), "name" : "Bob", "dobSeconds" : 0 }
]
```

Code examples

To view a code example for using the $second command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const result = await users.aggregate([{ $project: { name: 1, dobSeconds:
 { $second: '$dob' } } }]).toArray();
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    users = db['users']

    result = list(users.aggregate([{'$project': {'name': 1, 'dobSeconds':
 {'$second': '$dob'}}}]))
    print(result)

    client.close()

example ()
```

$set

New from version 8.0

Not supported by Elastic cluster.

The `$set` aggregation stage in Amazon DocumentDB allows you to add new fields or update existing field values in documentation during an aggregation pipeline.

Parameters

- `expression`: The expression to evaluate. This can be any valid aggregation expression, including field references and arithmetic operations.

Example (MongoDB Shell)

The following example demonstrates the use of the `$set` aggregation stage to calculate totals by multiplying the `quantity` field by the `price` field.

Create sample documents

```
db.inventory.insertMany([
  { item: "pencil", quantity: 100, price: 0.24},
  { item: "pen", quantity: 204, price: 1.78 }
]);
```

Aggregation example

```
db.inventory.aggregate([
  {
    $set: {
      total: { $multiply: ["$quantity", "$price"] }
    }
  }
])
```

Output

```
[
  {
    _id: ObjectId('69248951d66dcae121d2950d'),
```

```
    item: 'pencil',
    quantity: 100,
    price: 0.24,
    total: 24
  },
  {
    _id: ObjectId('69248951d66dcae121d2950e'),
    item: 'pen',
    quantity: 204,
    price: 1.78,
    total: 363.12
  }
]
```

Code examples

To view a code example for using the `$set` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const inventory = db.collection('inventory');

  const result = await inventory.aggregate([
      {
        $set: {
          total: { $multiply: ["$quantity", "$price"] }
        }
      }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    inventory = db['inventory']

    result = list(inventory.aggregate([
      {
        "$set": {
          "total": { "$multiply": ["$quantity", "$price"] }
        }
      }
    ]))

    print(result)
    client.close()

example()
```

$setDifference

New from version 4.0.

The `$setDifference` operator in Amazon DocumentDB is used to compare two sets and return the elements that are in the first set but not in the second set. This operator is useful for finding the unique elements between two sets.

Parameters

- `firstSet` : The first set to compare.
- `secondSet` : The second set to compare.

Example (MongoDB Shell)

The following example demonstrates how to use the `$setDifference` operator to find the unique elements between two sets.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, fruits: ["apple", "banana", "cherry", "date"] },
  { _id: 2, fruits: ["banana", "cherry", "date", "elderberry"] }
]);
```

Query example

```
db.collection.aggregate([
  {
    $project: {
      uniqueFruits: { $setDifference: ["$fruits", ["banana", "cherry", "date"]] }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "uniqueFruits": ["apple"] },
  { "_id": 2, "uniqueFruits": ["elderberry"] }
]
```

The query performs the following steps:

1. It uses the `$project` stage to create a new field `uniqueFruits` for each document.

2. The `$setDifference` operator compares the `fruits` array with the array `["banana", "cherry", "date"]` and returns the unique elements in the `fruits` array.

Code examples

To view a code example for using the `$setDifference` command, choose the tab for the language that you want to use:

Node.js

Here's an example of how to use the `$setDifference` operator in a Node.js application:

```
const { MongoClient } = require('mongodb');
```

```
async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('mycollection');

  // Insert sample documents
  await collection.insertMany([
    { _id: 1, fruits: ["apple", "banana", "cherry", "date"] },
    { _id: 2, fruits: ["banana", "cherry", "date", "elderberry"] }
  ]);

 // Query using $setDifference
  const result = await collection.aggregate([
    {
      $project: {
        uniqueFruits: { $setDifference: ["$fruits", ["banana", "cherry", "date"]] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

Here's an example of how to use the `$setDifference` operator in a Python application:

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['mycollection']

    # Insert sample documents
    collection.insert_many([
```

```
        {'_id': 1, 'fruits': ["apple", "banana", "cherry", "date"]},
        {'_id': 2, 'fruits': ["banana", "cherry", "date", "elderberry"]}
    ])

    # Query using $setDifference
    result = list(collection.aggregate([
        {
            '$project': {
                'uniqueFruits': {'$setDifference': ['$fruits', ["banana", "cherry",
 "date"]]}
            }
        }
    ]))

    print(result)
    client.close()

if __name__ == '__main__':
    main()
```

$setEquals

The `$setEquals` operator in Amazon DocumentDB is used to determine if two sets are equal. It compares two arrays and returns `true` if they contain the same distinct elements, regardless of their order.

Parameters

- `expression1`: The first array to compare.
- `expression2`: The second array to compare.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$setEquals` operator to compare two sets of values.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, fruits: ["apple", "banana", "cherry"] },
  { _id: 2, fruits: ["banana", "apple", "cherry"] },
```

```
  { _id: 3, fruits: ["apple", "banana", "orange"] }
])
```

Query example

```
db.collection.find({
  $expr: {
    $setEquals: ["$fruits", ["apple", "banana", "cherry"]]
  }
})
```

Output

```
{ "_id" : 1, "fruits" : [ "apple", "banana", "cherry" ] }
{ "_id" : 2, "fruits" : [ "banana", "apple", "cherry" ] }
```

The query uses the `$setEquals` operator to compare the `fruits` field of each document with the array `["apple", "banana", "cherry"]`. The documents where the `fruits` field is equal to the comparison array are returned.

Code examples

To view a code example for using the `$setEquals` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('mycollection');

  // Insert sample documents
  await collection.insertMany([
    { _id: 1, fruits: ["apple", "banana", "cherry"] },
    { _id: 2, fruits: ["banana", "apple", "cherry"] },
    { _id: 3, fruits: ["apple", "banana", "orange"] }
  ]);
```

```
  // Query using $setEquals
  const result = await collection.find({
    $expr: {
      $setEquals: ["$fruits", ["apple", "banana", "cherry"]]
    }
  }).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['mycollection']

    # Insert sample documents
    collection.insert_many([
        {"_id": 1, "fruits": ["apple", "banana", "cherry"]},
        {"_id": 2, "fruits": ["banana", "apple", "cherry"]},
        {"_id": 3, "fruits": ["apple", "banana", "orange"]}
    ])

    # Query using $setEquals
    result = list(collection.find({
        "$expr": {
            "$setEquals": ["$fruits", ["apple", "banana", "cherry"]]
        }
    }))

    print(result)
    client.close()

example()
```

$setIntersection

The `$setIntersection` operator in Amazon DocumentDB is used to return the common elements between two or more arrays. This operator is particularly useful when working with sets of data, allowing you to find the intersection of multiple sets.

Parameters

- `array1`: The first array to intersect.
- `array2`: The second array to intersect.
- `arrayN`: (optional) Additional arrays to intersect.

Example (MongoDB Shell)

The following example demonstrates how to use the `$setIntersection` operator to find the common elements between two arrays.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, colors: ["red", "blue", "green"] },
  { _id: 2, colors: ["blue", "yellow", "orange"] },
  { _id: 3, colors: ["red", "green", "purple"] }
])
```

Query example

```
db.collection.aggregate([
  { $project: {
      _id: 1,
      commonColors: { $setIntersection: ["$colors", ["red", "blue", "green"]] }
    }
  }
])
```

Output

```
[
  { "_id": 1, "commonColors": ["red", "blue", "green"] },
```

```
  { "_id": 2, "commonColors": ["blue"] },
  { "_id": 3, "commonColors": ["red", "green"] }
]
```

Code examples

To view a code example for using the `$setIntersection` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('mycollection');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        commonColors: { $setIntersection: ["$colors", ["red", "blue", "green"]] }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    db = client['test']
    collection = db['mycollection']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'commonColors': { '$setIntersection': ["$colors", ["red", "blue",
 "green"]] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$setIsSubset

The `$setIsSubset` operator in Amazon DocumentDB is used to determine if a set of values is a subset of another set. It is useful for performing set-based comparisons and operations on array fields.

Parameters

- `field`: The field to apply the `$setIsSubset` operator to.
- `set`: The set to compare the field against.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$setIsSubset` operator to check if the `tags` field is a subset of the specified set.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Product A", tags: ["tag1", "tag2", "tag3"] },
  { _id: 2, name: "Product B", tags: ["tag1", "tag2"] },
  { _id: 3, name: "Product C", tags: ["tag2", "tag3"] }
```

```
]);
```

Query example

```
db.products.find({
  $expr: { $setIsSubset: [["tag1", "tag2"], "$tags"] }
})
```

Note: `$setIsSubset` is an aggregation operator and cannot be used directly in find() queries. In this example, `$expr` is used with `find()` to bridge the gap between query operators and aggregation expressions.

Output

```
[
  { "_id" : 1, "name" : "Product A", "tags" : [ "tag1", "tag2", "tag3" ] },
  { "_id" : 2, "name" : "Product B", "tags" : [ "tag1", "tag2" ] }
]
```

The query returns the documents where the `tags` field is a subset of the set `["tag1", "tag2"]`.

Code examples

To view a code example for using the `$setIsSubset` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const result = await collection.find({
    $expr: { $setIsSubset: [["tag1", "tag2"], "$tags"] }
  }).toArray();
```

```
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    result = list(collection.find({
        '$expr': {'$setIsSubset': [['tag1', 'tag2'], '$tags']}
    }))

    print(result)

    client.close()

example()
```

$setUnion

The `$setUnion` aggregation operator in Amazon DocumentDB is used to combine two or more sets of values and return a set that contains all the unique elements from the input sets. This operator is useful when you need to perform set-based operations on array fields in your documents.

Parameters

- `expression1`: An expression that resolves to an array.
- `expression2`: An expression that resolves to an array.
- `expressionN`: Additional expressions that resolve to arrays (optional).

Example (MongoDB Shell)

The following example demonstrates how to use the `$setUnion` operator to combine the unique elements from two array fields in a collection.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "Alice", hobbies: ["reading", "swimming"], skills: ["coding",
 "writing"] },
  { _id: 2, name: "Bob", hobbies: ["cooking", "gardening"], skills: ["coding",
 "photography"] },
  { _id: 3, name: "Charlie", hobbies: ["reading", "painting"], skills: ["gardening",
 "music"] }
]);
```

Query example

```
db.users.aggregate([
  {
    $project: {
      name: 1,
      allInterests: { $setUnion: ["$hobbies", "$skills"] }
    }
  }
]);
```

Output

```
[
  { "_id" : 1, "name" : "Alice", "allInterests" : [ "coding", "reading", "swimming",
 "writing" ] },
  { "_id" : 2, "name" : "Bob", "allInterests" : [ "coding", "cooking", "gardening",
 "photography" ] },
  { "_id" : 3, "name" : "Charlie", "allInterests" : [ "gardening", "music", "painting",
 "reading" ] }
]
```

In this example, the `$setUnion` operator is used to combine the unique elements from the `hobbies` and `skills` array fields for each user document. The resulting `allInterests` field contains the union of all the unique hobbies and skills for each user.

Code examples

To view a code example for using the `$setUnion` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const result = await users.aggregate([
    {
      $project: {
        _id: 1,
        name: 1,
        allInterests: { $setUnion: ["$hobbies", "$skills"] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    users = db['users']
```

```
    result = list(users.aggregate([
        {
            '$project': {
                '_id': 1,
                'name': 1,
                'allInterests': { '$setUnion': ["$hobbies", "$skills"] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$setWindowFields

New from version 8.0.2.

The `$setWindowFields` stage in Amazon DocumentDB performs operations on a specified span of documents (a *window*) in a partition, and returns the results based on the chosen window operator. The stage groups the input documents into partitions, optionally sorts them, and for each document computes one or more output fields over a window of surrounding documents.

Syntax

```
{
  $setWindowFields: {
    partitionBy: <expression>,
    sortBy: { <sortField1>: <sortOrder>, ... },
    output: {
      <outputField1>: {
        <windowOperator>: <expression>,
        window: {
          documents: [ <lowerBound>, <upperBound> ],
          range: [ <lowerBound>, <upperBound> ],
          unit: <time unit>
        }
      },
      ...
    }
  }
```

```
}
```

Parameters

- `partitionBy`: Optional. A single expression used to group the documents into partitions. This can be a field path (for example, `"$state"`) or another expression. If omitted, all input documents belong to a single partition.
- `sortBy`: Optional. A document that specifies the field or fields to sort the documents in each partition by, in the form `{ <field>: 1 }` for ascending order or `{ <field>: -1 }` for descending order. A `sortBy` is required when you use document-based or range-based window bounds, and when you use one of the ranking operators.
- `output`: Required. A document that specifies the fields to append to each document. You can specify multiple output fields, but each output field takes exactly one window operator, its argument, and an optional `window` document that defines the window boundaries: `{ <field>: { <windowOperator>: <expression>, window: { ... } } }`.

Window boundaries

Within each `output` field you can optionally define the window boundaries with a `window` document. A window can be defined by document position or by a range of values.

- `documents`: Specifies the window as a number of documents before and after the current document, in the form `[ <lowerBound>, <upperBound> ]`. Each bound is an integer offset (for example, `-1` or `2`), or one of the keywords `"unbounded"` and `"current"`. `"unbounded"` extends the window to the start of the partition when used as the lower bound, or to the end of the partition when used as the upper bound. `"current"` means the current document. A `documents` window requires a `sortBy` unless both bounds are `"unbounded"`.
- `range`: Specifies the window as a range of `sortBy` field values relative to the current document, in the form `[ <lowerBound>, <upperBound> ]`. Each bound is a number that is added to the `sortBy` value of the current document, or the `"unbounded"` keyword. A `range` window requires exactly one ascending `sortBy` field and does not accept the `"current"` keyword.
- `unit`: Optional. Used with `range` to interpret the numeric window bounds as a span of time when the `sortBy` field holds date values. Amazon DocumentDB supports the following units: `"year"`, `"quarter"`, `"month"`, `"week"`, `"day"`, `"hour"`, `"minute"`, `"second"`, and `"millisecond"`. When `unit` is specified, the `range` bounds must be integers.

When the `window` document is omitted, the default window is `documents: ["unbounded", "unbounded"]`, which spans the entire partition.

> **Note**
>
> When you use a `range` window over date values, we recommend a fixed-duration `unit` such as `"day"` (or smaller) rather than `"month"`, `"quarter"`, or `"year"`, whose calendar-dependent lengths can produce ambiguous window boundaries.

Supported window operators

Amazon DocumentDB supports the following 26 window operators in the `output` field of the `$setWindowFields` stage:

- `$sum`
- `$avg`
- `$count`
- `$push`
- `$addToSet`
- `$stdDevPop`
- `$stdDevSamp`
- `$covariancePop`
- `$covarianceSamp`
- `$first`
- `$last`
- `$firstN`
- `$lastN`
- `$top`
- `$bottom`
- `$topN`
- `$bottomN`
- `$min`

- $minN
- $max
- $maxN
- $median
- $percentile
- $rank
- $denseRank
- $documentNumber

> **Note**
>
> The $rand operator is not supported within a window operator in the $setWindowFields stage.

Empty windows

When a window contains no documents (for example, a range window with no values in range), the result depends on the operator: $count and $sum return 0; $push, $addToSet, $firstN, $lastN, $minN, $maxN, $topN, and $bottomN return an empty array; $percentile returns an array of null values (one per requested percentile); all other operators return null.

Example (MongoDB Shell)

The following example partitions the documents by region, sorts each partition by day, and computes a running total of revenue from the start of the partition through the current document.

Create sample documents

```
db.sales.insertMany([
  { _id: 1, region: "east", day: 1, revenue: 100 },
  { _id: 2, region: "east", day: 2, revenue: 150 },
  { _id: 3, region: "east", day: 3, revenue: 120 },
  { _id: 4, region: "west", day: 1, revenue: 200 },
  { _id: 5, region: "west", day: 2, revenue: 180 }
```

```
]);
```

Query example

```
db.sales.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$region",
      sortBy: { day: 1 },
      output: {
        runningRevenue: {
          $sum: "$revenue",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "region": "east", "day": 1, "revenue": 100, "runningRevenue": 100 },
  { "_id": 2, "region": "east", "day": 2, "revenue": 150, "runningRevenue": 250 },
  { "_id": 3, "region": "east", "day": 3, "revenue": 120, "runningRevenue": 370 },
  { "_id": 4, "region": "west", "day": 1, "revenue": 200, "runningRevenue": 200 },
  { "_id": 5, "region": "west", "day": 2, "revenue": 180, "runningRevenue": 380 }
]
```

Each document is augmented with `runningRevenue`, the cumulative sum of `revenue` within its partition up to and including the current document.

Code examples

To view a code example for using the `$setWindowFields` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const sales = db.collection('sales');

    await sales.insertMany([
      { _id: 1, region: "east", day: 1, revenue: 100 },
      { _id: 2, region: "east", day: 2, revenue: 150 },
      { _id: 3, region: "east", day: 3, revenue: 120 },
      { _id: 4, region: "west", day: 1, revenue: 200 },
      { _id: 5, region: "west", day: 2, revenue: 180 }
    ]);

    const result = await sales.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$region",
          sortBy: { day: 1 },
          output: {
            runningRevenue: {
              $sum: "$revenue",
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        sales = db['sales']

        sales.insert_many([
            { '_id': 1, 'region': 'east', 'day': 1, 'revenue': 100 },
            { '_id': 2, 'region': 'east', 'day': 2, 'revenue': 150 },
            { '_id': 3, 'region': 'east', 'day': 3, 'revenue': 120 },
            { '_id': 4, 'region': 'west', 'day': 1, 'revenue': 200 },
            { '_id': 5, 'region': 'west', 'day': 2, 'revenue': 180 }
        ])

        result = list(sales.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$region',
                    'sortBy': { 'day': 1 },
                    'output': {
                        'runningRevenue': {
                            '$sum': '$revenue',
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$sigmoid

New from version 8.0.1.

The $sigmoid operator in Amazon DocumentDB applies the logistic sigmoid function, 1 / (1 + e^(-x)), to a numeric input. The function maps any real number to a value between 0 and 1, producing an S-shaped curve. This is useful when you need to normalize values into the 0–1 range, such as converting raw scores into probabilities or relevance scores.

Parameters

- expression: An expression that resolves to a numeric value.

Example (MongoDB Shell)

The following example shows how to use the $sigmoid operator to convert logit values into probabilities.

Create sample documents

```
db.predictions.insertMany([
  {_id: 1, logit: -2},
  {_id: 2, logit: 0},
  {_id: 3, logit: 2}
]);
```

Query example

```
db.predictions.aggregate([
  { $project: { probability: { $sigmoid: "$logit" } } }
]);
```

Output

```
[
  {_id: 1, probability: 0.11920292202211755},
  {_id: 2, probability: 0.5},
  {_id: 3, probability: 0.8807970779778823}
]
```

Code examples

To view a code example for using the $sigmoid operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('predictions');
    const result = await collection.aggregate([
      { $project: { probability: { $sigmoid: "$logit" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['predictions']
        result = list(collection.aggregate([
            {'$project': {'probability': {'$sigmoid': '$logit'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$sin

New from version 8.0.1.

The `$sin` operator in Amazon DocumentDB returns the sine of a value that is measured in radians. Use it in the aggregation pipeline to perform trigonometric calculations on numeric fields.

Parameters

- `expression`: An expression that resolves to a number in radians.

Example (MongoDB Shell)

The following example shows how to use the `$sin` operator to calculate the sine of angle values in radians.

Create sample documents

```
db.angles.insertMany([
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 0.5235987755982988 },
  { "_id": 3, "angle": 1.5707963267948966 }
]);
```

Query example

```
db.angles.aggregate([
  { $project: {
    "sine": { $sin: "$angle" }
  }}
]);
```

Output

```
[
  { "_id": 1, "sine": 0 },
  { "_id": 2, "sine": 0.49999999999999994 },
  { "_id": 3, "sine": 1 }
]
```

Code examples

To view a code example for using the `$sin` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('angles');

  const result = await collection.aggregate([
    { $project: {
      "sine": { $sin: "$angle" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['angles']

    result = list(collection.aggregate([
        { '$project': {
            'sine': { '$sin': '$angle' }
        }}
```

```
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$sinh

New from version 8.0.1.

The $sinh operator in Amazon DocumentDB returns the hyperbolic sine of a value. Use it in the aggregation pipeline to perform hyperbolic trigonometric calculations on numeric fields.

Parameters

- expression: An expression that resolves to a number.

The return type is double by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Example (MongoDB Shell)

The following example shows how to use the $sinh operator to calculate the hyperbolic sine of numeric values.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 1 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $sinh: "$value" }
  }}
```

```
]);
```

Output

```
[
  { "_id": 1, "result": 0 },
  { "_id": 2, "result": 1.1752011936438014 },
  { "_id": 3, "result": -1.1752011936438014 }
]
```

Code examples

To view a code example for using the $sinh operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "result": { $sinh: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient
```

```
def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$sinh': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$skip

In Amazon DocumentDB, the `$skip` operator is used to offset the starting point of the query results, allowing you to retrieve a specific subset of the matching documents. This is particularly useful in pagination scenarios, where you want to retrieve subsequent pages of results.

Parameters

- `skip`: The number of documents to skip before returning the remaining documents.

Example (MongoDB Shell)

The following example demonstrates how to use the `$skip` operator to retrieve the second page of results (documents 11-20) from a collection.

Create sample documents

```
db.collection.insert([
  { "name": "Document 1" },
  { "name": "Document 2" },
  { "name": "Document 3" },
  { "name": "Document 4" },
```

```
  { "name": "Document 5" },
  { "name": "Document 6" },
  { "name": "Document 7" },
  { "name": "Document 8" },
  { "name": "Document 9" },
  { "name": "Document 10" },
  { "name": "Document 11" },
  { "name": "Document 12" },
  { "name": "Document 13" },
  { "name": "Document 14" },
  { "name": "Document 15" },
  { "name": "Document 16" },
  { "name": "Document 17" },
  { "name": "Document 18" },
  { "name": "Document 19" },
  { "name": "Document 20" }
]);
```

Query example

```
db.collection.find({}, { "name": 1 })
             .skip(10)
             .limit(10);
```

Output

```
[
  { "_id" : ObjectId("..."), "name" : "Document 11" },
  { "_id" : ObjectId("..."), "name" : "Document 12" },
  { "_id" : ObjectId("..."), "name" : "Document 13" },
  { "_id" : ObjectId("..."), "name" : "Document 14" },
  { "_id" : ObjectId("..."), "name" : "Document 15" },
  { "_id" : ObjectId("..."), "name" : "Document 16" },
  { "_id" : ObjectId("..."), "name" : "Document 17" },
  { "_id" : ObjectId("..."), "name" : "Document 18" },
  { "_id" : ObjectId("..."), "name" : "Document 19" },
  { "_id" : ObjectId("..."), "name" : "Document 20" }
]
```

Code examples

To view a code example for using the `$skip` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const results = await collection.find({}, { projection: { name: 1 } })
                                  .skip(10)
                                  .limit(10)
                                  .toArray();

  console.log(results);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.mydatabase
    collection = db.collection

    results = list(collection.find({}, {'name': 1})
                   .skip(10)
                   .limit(10))
```

```
    print(results)

    client.close()

if __name__ == '__main__':
    main()
```

$slice

The `$slice` aggregation operator enables you to return a subset of an array by either traversing the array from the beginning or the end of the array. This is used to display a limited number of items from an array field, such as the top or bottom N items.

Parameters

- `array`: The array field to be sliced.
- n: An integer that specifies the number of elements to return. A positive value starts from the beginning of the array, while a negative value starts from the end of the array.

Example (MongoDB Shell)

The following example demonstrates how to use `$slice` to return the first two favorite sweets for each chef.

Create sample documents

```
db.sweets.insertMany([
  { "_id" : 1, "name" : "Alvin", "favorites": [ "chocolate", "cake", "toffee",
 "beignets" ] },
  { "_id" : 2, "name" : "Tom", "favorites": [ "donuts", "pudding", "pie" ] },
  { "_id" : 3, "name" : "Jessica", "favorites": [ "fudge", "smores", "pudding",
 "cupcakes" ] },
  { "_id" : 4, "name" : "Rachel", "favorites": [ "ice cream" ] }
]);
```

Query example

```
db.sweets.aggregate([
  { $project: { _id: 0, name: 1, topTwoFavorites: { $slice: [ "$favorites", 2 ] } } }
]);
```

Output

```
[
  { name: 'Alvin', topTwoFavorites: [ 'chocolate', 'cake' ] },
  { name: 'Tom', topTwoFavorites: [ 'donuts', 'pudding' ] },
  { name: 'Jessica', topTwoFavorites: [ 'fudge', 'smores' ] },
  { name: 'Rachel', topTwoFavorites: [ 'ice cream' ] }
]
```

In this example, the `$slice` operator is used to extract the first two elements from the `favorites` array for each document.

Code examples

To view a code example for using the `$slice` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('sweets');

  const result = await collection.aggregate([
    { $project: { name: 1, topTwoFavorites: { $slice: ['$favorites', 2] } } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['sweets']

    result = list(collection.aggregate([
        { '$project': { 'name': 1, 'topTwoFavorites': { '$slice': ['$favorites',
 2] } } }
    ]))

    print(result)
    client.close()

example()
```

$size

The `$size` operator is used to return the count of items within an array field. This can be used to determine the number of elements in an array stored in a document.

Parameters

- `field`: The field path whose array size you want to return.

Example (MongoDB Shell)

This example shows how to use the `$size` operator to return the count of teams each user is following.

Create sample documents

```
db.profiles.insertMany([
  { _id: 1, name: "John Doe", teams: ["Acme", "Widgets", "Gadgets"] },
  { _id: 2, name: "Jane Smith", teams: ["Acme", "Gadgets"] },
  { _id: 3, name: "Bob Johnson", teams: ["Acme", "Widgets", "Gadgets"] }
]);
```

Query example

```
db.profiles.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      "numberOfTeams": { $size: "$teams" }
    }
  }
])
```

Output

```
[
  { name: 'John Doe', numberOfTeams: 3 },
  { name: 'Jane Smith', numberOfTeams: 2 },
  { name: 'Bob Johnson', numberOfTeams: 3 }
]
```

Code examples

To view a code example for using the `$size` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const profiles = db.collection('profiles');

  const result = await profiles.aggregate([
    {
      $project: {
        item: 1,
        "numberOfTeams": { $size: "$teams" }
      }
    }
  ]).toArray();
```

```
  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    profiles = db.profiles

    result = list(profiles.aggregate([
        {
            '$project': {
                'item': 1,
                'numberOfTeams': { '$size': '$teams' }
            }
        }
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$sort

The `$sort` aggregation stage orders documents in the pipeline based on specified field values. Documents are arranged in ascending or descending order according to the sort criteria provided.

Parameters

- `field`: The field name to sort by.
- `order`: Use `1` for ascending order or `-1` for descending order.

Example (MongoDB Shell)

The following example demonstrates using the $sort stage to order products by price in descending order.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Laptop", category: "Electronics", price: 1200 },
  { _id: 2, name: "Mouse", category: "Electronics", price: 25 },
  { _id: 3, name: "Desk", category: "Furniture", price: 350 },
  { _id: 4, name: "Chair", category: "Furniture", price: 150 },
  { _id: 5, name: "Monitor", category: "Electronics", price: 400 }
]);
```

Query example

```
db.products.aggregate([
  { $sort: { price: -1 } }
]);
```

Output

```
[
  { _id: 1, name: 'Laptop', category: 'Electronics', price: 1200 },
  { _id: 5, name: 'Monitor', category: 'Electronics', price: 400 },
  { _id: 3, name: 'Desk', category: 'Furniture', price: 350 },
  { _id: 4, name: 'Chair', category: 'Furniture', price: 150 },
  { _id: 2, name: 'Mouse', category: 'Electronics', price: 25 }
]
```

Code examples

To view a code example for using the $sort aggregation stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const result = await collection.aggregate([
    { $sort: { price: -1 } }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    result = list(collection.aggregate([
        { '$sort': { 'price': -1 } }
    ]))

    print(result)
    client.close()

example()
```

Incremental sort

Incremental sort is an optimization in which Amazon DocumentDB uses an index that already orders the *leading* fields of a multi-field sort. Amazon DocumentDB then sorts only the remaining fields within each group of documents that share the same leading values. Instead of sorting the entire result set in memory as one block (a full sort), Amazon DocumentDB reads documents

in index order. It then performs many small, independent sorts, one per group. The results are identical to a full sort. Only the query plan and the amount of memory and work it uses change.

Benefits

Incremental sort provides the following benefits:

- **Lower memory use and fewer disk-based sort spills.** Amazon DocumentDB orders one small group at a time instead of the whole result set at once.
- **Faster results with `limit`.** Amazon DocumentDB can emit sorted documents before it reads the entire input. A sort followed by a `limit` (for example, a top-N or paginated query) can stop as soon as it has produced enough documents.

When Amazon DocumentDB chooses an incremental sort

For example, given a sort on `{ customerId: 1, orderDate: -1 }` and an index on only `{ customerId: 1 }`, the index already returns documents ordered by `customerId`. Amazon DocumentDB only needs to order each group of documents that share the same `customerId` by `orderDate`. The already-ordered leading field (`customerId`) is the **presorted key**. Depending on how much of the sort an index already provides, Amazon DocumentDB chooses one of three behaviors:

- **The index orders every sort key.** No sort is needed. Amazon DocumentDB reads documents directly in the requested order.
- **The index orders a leading prefix of the sort keys.** Amazon DocumentDB uses an incremental sort, ordering only the trailing keys within each presorted group.
- **No index orders any leading sort key.** Amazon DocumentDB performs a full sort over the entire input.

Requirements and limitations

Incremental sort requires Amazon DocumentDB 8.0.2 or later, running planner version 3.0 (the default query planner as of Amazon DocumentDB 8.0.2). For more information about planner versions, see Query planner v3.

Amazon DocumentDB cannot use a partial index or a multikey (array) index to supply the presorted prefix for an incremental sort. In those cases, Amazon DocumentDB falls back to a full sort. This never affects the correctness of results.

Verifying an incremental sort with `explain()`

Use `explain()` to confirm that a query uses an incremental sort. In the winning plan, an incremental sort appears as a SORT stage that includes a `presortedKeys` array listing the leading sort fields the index already ordered. A full sort appears as a SORT stage without a `presortedKeys` array. The `presortedKeys` field is a Amazon DocumentDB-specific addition to the `explain()` output and does not appear in MongoDB explain plans.

```
db.orders.createIndex({ customerId: 1 })

db.orders.find({}).sort({ customerId: 1, orderDate: -1 }).explain()
```

```
{
    "stage": "SORT",
    "sortPattern": {
        "customerId": 1,
        "orderDate": -1
    },
    "presortedKeys": [
        "customerId"
    ],
    "inputStage": {
        "stage": "IXSCAN",
        "indexName": "customerId_1",
        "direction": "forward"
    }
}
```

$sortArray

New from version 8.0

Not supported by Elastic cluster.

The `$sortArray` operator in Amazon DocumentDB sorts the elements of an array. For arrays of scalar values, the array is sorted by element value. For arrays of documents, the array is sorted by the specified field or fields. The operator returns the sorted array in ascending (1) or descending (-1) order.

> **Note**
>
> `$sortArray` is supported in the `$project` stage of the aggregation pipeline with query planner version 3.

Syntax

```
{ $sortArray: { input: <array>, sortBy: <sort specification> } }
```

Parameters

- `input`: The array to sort.
- `sortBy`: The sort order. For an array of scalar values, specify 1 for ascending or -1 for descending. For an array of documents, specify a document of the form `{ field: 1 }` or `{ field: -1 }` for each field to sort by.

Example (MongoDB Shell)

The following example demonstrates how to use the `$sortArray` operator to sort the elements of an array in ascending order.

Create sample documents

```
db.miles.insertMany([
  { "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "flight_miles" : [ 1205, 2560, 880 ]},
  { "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 2560, 890, 2780]},
  { "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 1205, 880]}
]);
```

Query example

```
db.miles.aggregate([
  {
    $project: {
```

```
      _id: 1,
      member_since: 1,
      credit_card: 1,
      sorted_flight_miles: { $sortArray: { input: "$flight_miles", sortBy: 1 } }
    }
  }
]);
```

Output

```
{ "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "sorted_flight_miles" : [ 880, 1205, 2560 ] }
{ "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "sorted_flight_miles" : [ 890, 1205, 2560, 2780 ] }
{ "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "sorted_flight_miles" : [ 880, 1205 ] }
```

In this example, the `$sortArray` operator sorts the `flight_miles` array in ascending order. The resulting `sorted_flight_miles` field in the output shows the elements of the array in sorted order.

Code examples

To view a code example for using the `$sortArray` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$sortArray` operator in a Node.js application:

```
const { MongoClient } = require('mongodb');

async function sortArray() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('miles');

  const result = await collection.aggregate([
    {
      $project: {
```

```
        _id: 1,
        member_since: 1,
        credit_card: 1,
        sorted_flight_miles: { $sortArray: { input: '$flight_miles', sortBy: 1 } }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

sortArray();
```

Python

Here's an example of using the `$sortArray` operator in a Python application:

```
from pymongo import MongoClient

def sort_array():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.miles

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'member_since': 1,
                'credit_card': 1,
                'sorted_flight_miles': { '$sortArray': { 'input': '$flight_miles',
 'sortBy': 1 } }
            }
        }
    ]))

    print(result)
    client.close()

sort_array()
```

$sortByCount

New from version 8.0.1.

The `$sortByCount` aggregation stage in Amazon DocumentDB groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group, and sorts the results by count in descending order. It is equivalent to a `$group` stage with a `$sum` accumulator followed by a `$sort` stage on the count field.

Parameters

- `expression`: The expression to group by. This can be a field path (prefixed with $) or any valid aggregation expression.

Example (MongoDB Shell)

The following example shows how to use the `$sortByCount` stage to count and sort documents by a category field.

Create sample documents

```
db.products.insertMany([
  { name: "Widget", category: "electronics" },
  { name: "Gadget", category: "electronics" },
  { name: "Doohickey", category: "electronics" },
  { name: "Apple", category: "food" },
  { name: "Banana", category: "food" },
  { name: "Hammer", category: "tools" }
]);
```

Query example

```
db.products.aggregate([
  { $sortByCount: "$category" }
]);
```

Output

```
[
  { "_id": "electronics", "count": 3 },
```

```
  { "_id": "food", "count": 2 },
  { "_id": "tools", "count": 1 }
]
```

The results are automatically sorted by `count` in descending order.

Code examples

To view a code example for using the `$sortByCount` stage, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('products');

    const result = await collection.aggregate([
      { $sortByCount: "$category" }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        collection = db['products']

        result = list(collection.aggregate([
            { '$sortByCount': '$category' }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$split

The `$split` aggregation operator in Amazon DocumentDB is used to split a string into an array of substrings, based on a specified delimiter. This can be useful for parsing complex string fields and extracting individual components for further processing.

Parameters

- `string`: The string to be split.
- `delimiter`: The character or string used to split the input string.

Example (MongoDB Shell)

In this example, we use `$split` to separate the components of a "Desk" field into an array, making it easier to process the data.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-BVV-021" },
  { "_id": 2, "Desk": "Munich-HGG-32a" },
  { "_id": 3, "Desk": "Cologne-ayu-892.50" },
```

```
  { "_id": 4, "Desk": "Dortmund-Hop-78" }
]);
```

Query example

```
db.people.aggregate([
  { $project: { parts: { $split: ["$Desk", "-"] } } }
]);
```

Output

```
{ "_id" : 1, "parts" : [ "Düsseldorf", "BVV", "021" ] }
{ "_id" : 2, "parts" : [ "Munich", "HGG", "32a" ] }
{ "_id" : 3, "parts" : [ "Cologne", "ayu", "892.50" ] }
{ "_id" : 4, "parts" : [ "Dortmund", "Hop", "78" ] }
```

The output of `$split` creates an array that can be used in the application to display the information for the employees.

Code examples

To view a code example for using the `$split` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const result = await db.collection('people').aggregate([
    { $project: { parts: { $split: ['$Desk', '-'] } } }
  ]).toArray();
  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    result = list(db.people.aggregate([
        { '$project': { 'parts': { '$split': ['$Desk', '-'] } } }
    ]))
    print(result)
    client.close()

example()
```

$sqrt

New from version 4.0.

The `$sqrt` operator in Amazon DocumentDB is used to calculate the square root of a number.

Parameters

- `expression`: The argument can be any valid expression as long as it resolves to a non-negative number.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$sqrt` operator to calculate the square root of a number.

Create sample documents

```
db.numbers.insertMany([
  { "_id": 1, "number": 16 },
  { "_id": 2, "number": 36 },
  { "_id": 3, "number": 64 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    "_id": 1,
    "square_root": { $sqrt: "$number" }
  }}
]);
```

Output

```
[
  { _id: 1, square_root: 4 },
  { _id: 2, square_root: 6 },
  { _id: 3, square_root: 8 }
]
```

Code examples

To view a code example for using the `$sqrt` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
      await client.connect();

      const db = client.db('test');
      const collection = db.collection('numbers');

      const pipeline = [
        {
          $project: {
            _id: 1,
            square_root: { $sqrt: '$number' }
```

```
          }
        }
      ];

      const results = await collection.aggregate(pipeline).toArray();

      console.dir(results, { depth: null });

    } finally {
      await client.close();
    }
}

example().catch(console.error);
```

Python

```
from pymongo import MongoClient

def example():

  client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

  try:
      db = client.test
      collection = db.numbers

      pipeline = [
          {
              "$project": {
                  "_id": 1,
                  "square_root": {
                    "$sqrt": "$number"
                  }
              }
          }
      ]

      results = collection.aggregate(pipeline)

      for doc in results:
```

```
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

example()
```

$stdDevPop

New from version 8.0.1.

The `$stdDevPop` operator in Amazon DocumentDB calculates the population standard deviation of numeric values. As an accumulator, it computes the population standard deviation across documents within a group in the `$group` stage of an aggregation pipeline. As an expression, it calculates the population standard deviation of an array of numbers. The population standard deviation uses N as the divisor (not N-1). Non-numeric values are ignored. If there are no numeric values, it returns `null`. If there is only one numeric value, it returns `0`.

Parameters

- `expression`: An expression that resolves to a numeric value or an array of numeric values.

Example (MongoDB Shell)

The following example shows how to use the `$stdDevPop` operator to calculate the population standard deviation of scores per subject.

Create sample documents

```
db.scores.insertMany([
  { subject: "math", score: 60 },
  { subject: "math", score: 75 },
  { subject: "math", score: 85 },
  { subject: "math", score: 92 },
  { subject: "math", score: 78 },
  { subject: "science", score: 55 },
  { subject: "science", score: 70 },
  { subject: "science", score: 82 },
```

```
  { subject: "science", score: 91 },
  { subject: "science", score: 67 }
]);
```

Query example

```
db.scores.aggregate([
  { $group: {
      _id: "$subject",
      stdDev: { $stdDevPop: "$score" }
    }}
]);
```

Output

```
[
  { "_id": "math", "stdDev": 10.75174404457249 },
  { "_id": "science", "stdDev": 12.441864811996632 }
]
```

Expression usage example (MongoDB Shell)

The `$stdDevPop` operator can also be used as an expression within a `$project` stage to compute the population standard deviation of an array field.

Create sample documents

```
db.experiments.insertMany([
  { _id: 1, measurements: [10, 12, 14, 16, 18] },
  { _id: 2, measurements: [5, 5, 5, 5, 5] },
  { _id: 3, measurements: [2, 4, 6, 8, 10] }
]);
```

Query example

```
db.experiments.aggregate([
  { $project: {
      stdDev: { $stdDevPop: "$measurements" }
    }}
]);
```

Output

```
[
  { "_id": 1, "stdDev": 2.8284271247461903 },
  { "_id": 2, "stdDev": 0 },
  { "_id": 3, "stdDev": 2.8284271247461903 }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$stdDevPop` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the population standard deviation of the specified expression for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

Create sample documents

```
db.sensorData.insertMany([
  { _id: 1, sensor: "A", time: 1, reading: 4 },
  { _id: 2, sensor: "A", time: 2, reading: 10 },
  { _id: 3, sensor: "A", time: 3, reading: 10 },
  { _id: 4, sensor: "B", time: 1, reading: 10 },
  { _id: 5, sensor: "B", time: 2, reading: 20 }
]);
```

Query example

The following example partitions the documents by `sensor`, sorts each partition by `time`, and returns the running population standard deviation of `reading` from the start of the partition through the current document.

```
db.sensorData.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$sensor",
      sortBy: { time: 1 },
      output: {
        runningStdDev: {
          $stdDevPop: "$reading",
          window: { documents: ["unbounded", "current"] }
        }
      }
```

```
    }
  }
]);
```

Output

```
[
  { "_id": 1, "sensor": "A", "time": 1, "reading": 4, "runningStdDev": 0 },
  { "_id": 2, "sensor": "A", "time": 2, "reading": 10, "runningStdDev": 3 },
  { "_id": 3, "sensor": "A", "time": 3, "reading": 10, "runningStdDev":
 2.8284271247461903 },
  { "_id": 4, "sensor": "B", "time": 1, "reading": 10, "runningStdDev": 0 },
  { "_id": 5, "sensor": "B", "time": 2, "reading": 20, "runningStdDev": 5 }
]
```

Each document is augmented with `runningStdDev`, the population standard deviation of `reading` within its partition up to and including the current document. When the window contains a single value, `$stdDevPop` returns `0`.

Code examples

To view a code example for using the `$stdDevPop` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: stdDevPop across grouped documents
    const scores = db.collection('scores');
    await scores.insertMany([
```

```
  { subject: "math", score: 60 },
  { subject: "math", score: 75 },
  { subject: "math", score: 85 },
  { subject: "math", score: 92 },
  { subject: "math", score: 78 },
  { subject: "science", score: 55 },
  { subject: "science", score: 70 },
  { subject: "science", score: 82 },
  { subject: "science", score: 91 },
  { subject: "science", score: 67 }
]);
const accumulatorResult = await scores.aggregate([
  { $group: {
      _id: "$subject",
      stdDev: { $stdDevPop: "$score" }
    }}
]).toArray();
console.log('Accumulator result:', accumulatorResult);

// Expression usage: stdDevPop of an array field
const experiments = db.collection('experiments');
await experiments.insertMany([
  { _id: 1, measurements: [10, 12, 14, 16, 18] },
  { _id: 2, measurements: [5, 5, 5, 5, 5] },
  { _id: 3, measurements: [2, 4, 6, 8, 10] }
]);
const expressionResult = await experiments.aggregate([
  { $project: {
      stdDev: { $stdDevPop: "$measurements" }
    }}
]).toArray();
console.log('Expression result:', expressionResult);

// Window operator usage: running stdDevPop within each partition
const sensorData = db.collection('sensorData');
await sensorData.insertMany([
  { _id: 1, sensor: "A", time: 1, reading: 4 },
  { _id: 2, sensor: "A", time: 2, reading: 10 },
  { _id: 3, sensor: "A", time: 3, reading: 10 },
  { _id: 4, sensor: "B", time: 1, reading: 10 },
  { _id: 5, sensor: "B", time: 2, reading: 20 }
]);
const windowResult = await sensorData.aggregate([
  {
```

```
        $setWindowFields: {
          partitionBy: "$sensor",
          sortBy: { time: 1 },
          output: {
            runningStdDev: {
              $stdDevPop: "$reading",
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: stdDevPop across grouped documents
        scores = db['scores']
        scores.insert_many([
            { 'subject': 'math', 'score': 60 },
            { 'subject': 'math', 'score': 75 },
            { 'subject': 'math', 'score': 85 },
            { 'subject': 'math', 'score': 92 },
            { 'subject': 'math', 'score': 78 },
            { 'subject': 'science', 'score': 55 },
            { 'subject': 'science', 'score': 70 },
```

```
            { 'subject': 'science', 'score': 82 },
            { 'subject': 'science', 'score': 91 },
            { 'subject': 'science', 'score': 67 }
        ])
        accumulator_result = list(scores.aggregate([
            { '$group': {
                '_id': '$subject',
                'stdDev': { '$stdDevPop': '$score' }
            }}
        ]))
        print('Accumulator result:', accumulator_result)

        # Expression usage: stdDevPop of an array field
        experiments = db['experiments']
        experiments.insert_many([
            { '_id': 1, 'measurements': [10, 12, 14, 16, 18] },
            { '_id': 2, 'measurements': [5, 5, 5, 5, 5] },
            { '_id': 3, 'measurements': [2, 4, 6, 8, 10] }
        ])
        expression_result = list(experiments.aggregate([
            { '$project': {
                'stdDev': { '$stdDevPop': '$measurements' }
            }}
        ]))
        print('Expression result:', expression_result)

        # Window operator usage: running stdDevPop within each partition
        sensor_data = db['sensorData']
        sensor_data.insert_many([
            { '_id': 1, 'sensor': 'A', 'time': 1, 'reading': 4 },
            { '_id': 2, 'sensor': 'A', 'time': 2, 'reading': 10 },
            { '_id': 3, 'sensor': 'A', 'time': 3, 'reading': 10 },
            { '_id': 4, 'sensor': 'B', 'time': 1, 'reading': 10 },
            { '_id': 5, 'sensor': 'B', 'time': 2, 'reading': 20 }
        ])
        window_result = list(sensor_data.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$sensor',
                    'sortBy': { 'time': 1 },
                    'output': {
                        'runningStdDev': {
                            '$stdDevPop': '$reading',
                            'window': { 'documents': ['unbounded', 'current'] }
```

```
                        }
                    }
                }
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$stdDevSamp

New from version 8.0.1.

The `$stdDevSamp` operator in Amazon DocumentDB calculates the sample standard deviation of numeric values. As an accumulator, it computes the sample standard deviation across documents within a group in the `$group` stage of an aggregation pipeline. As an expression, it calculates the sample standard deviation of an array of numbers. The sample standard deviation uses N-1 as the divisor (Bessel's correction). Non-numeric values are ignored. If there are fewer than two numeric values, it returns `null`.

Parameters

- `expression`: An expression that resolves to a numeric value or an array of numeric values.

Example (MongoDB Shell)

The following example shows how to use the `$stdDevSamp` operator to calculate the sample standard deviation of scores per subject.

Create sample documents

```
db.scores.insertMany([
  { subject: "math", score: 80 },
  { subject: "math", score: 90 },
  { subject: "math", score: 85 },
  { subject: "math", score: 95 },
  { subject: "science", score: 70 },
  { subject: "science", score: 75 },
```

```
  { subject: "science", score: 80 },
  { subject: "science", score: 85 }
]);
```

Query example

```
db.scores.aggregate([
  { $group: {
      _id: "$subject",
      stdDev: { $stdDevSamp: "$score" }
    }}
]);
```

Output

```
[
  { "_id": "math", "stdDev": 6.454972243679028 },
  { "_id": "science", "stdDev": 6.454972243679028 }
]
```

Expression usage example (MongoDB Shell)

The `$stdDevSamp` operator can also be used as an expression within a `$project` stage to compute the sample standard deviation of an array field.

Create sample documents

```
db.experiments.insertMany([
  { _id: 1, measurements: [10, 12, 14, 16, 18] },
  { _id: 2, measurements: [5, 5, 5, 5, 5] },
  { _id: 3, measurements: [2, 4, 6, 8, 10] }
]);
```

Query example

```
db.experiments.aggregate([
  { $project: {
      stdDev: { $stdDevSamp: "$measurements" }
    }}
]);
```

Output

```
[
  { "_id": 1, "stdDev": 3.1622776601683795 },
  { "_id": 2, "stdDev": 0 },
  { "_id": 3, "stdDev": 3.1622776601683795 }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$stdDevSamp` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the sample standard deviation of the specified expression for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

Create sample documents

```
db.sensorReadings.insertMany([
  { _id: 1, sensor: "A", time: 1, value: 2 },
  { _id: 2, sensor: "A", time: 2, value: 4 },
  { _id: 3, sensor: "A", time: 3, value: 6 },
  { _id: 4, sensor: "B", time: 1, value: 10 },
  { _id: 5, sensor: "B", time: 2, value: 12 }
]);
```

Query example

The following example partitions the documents by `sensor`, sorts each partition by `time`, and returns the running sample standard deviation of `value` from the start of the partition through the current document.

```
db.sensorReadings.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$sensor",
      sortBy: { time: 1 },
      output: {
        runningStdDev: {
          $stdDevSamp: "$value",
          window: { documents: ["unbounded", "current"] }
        }
      }
```

```
    }
  }
]);
```

Output

```
[
  { "_id": 1, "sensor": "A", "time": 1, "value": 2, "runningStdDev": null },
  { "_id": 2, "sensor": "A", "time": 2, "value": 4, "runningStdDev":
 1.4142135623730951 },
  { "_id": 3, "sensor": "A", "time": 3, "value": 6, "runningStdDev": 2 },
  { "_id": 4, "sensor": "B", "time": 1, "value": 10, "runningStdDev": null },
  { "_id": 5, "sensor": "B", "time": 2, "value": 12, "runningStdDev":
 1.4142135623730951 }
]
```

Each document is augmented with `runningStdDev`, the sample standard deviation of `value` within its partition up to and including the current document. When the window contains fewer than two values (the first document in each partition), `$stdDevSamp` returns `null`.

Code examples

To view a code example for using the `$stdDevSamp` operator, choose the tab for the language that you want to use. The following examples show accumulator usage (in `$group`), expression usage (in `$project`), and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');

    // Accumulator usage: stdDevSamp across grouped documents
    const scores = db.collection('scores');
```

```
await scores.insertMany([
  { subject: "math", score: 80 },
  { subject: "math", score: 90 },
  { subject: "math", score: 85 },
  { subject: "math", score: 95 },
  { subject: "science", score: 70 },
  { subject: "science", score: 75 },
  { subject: "science", score: 80 },
  { subject: "science", score: 85 }
]);
const accumulatorResult = await scores.aggregate([
  { $group: {
      _id: "$subject",
      stdDev: { $stdDevSamp: "$score" }
    }}
]).toArray();
console.log('Accumulator result:', accumulatorResult);

// Expression usage: stdDevSamp of an array field
const experiments = db.collection('experiments');
await experiments.insertMany([
  { _id: 1, measurements: [10, 12, 14, 16, 18] },
  { _id: 2, measurements: [5, 5, 5, 5, 5] },
  { _id: 3, measurements: [2, 4, 6, 8, 10] }
]);
const expressionResult = await experiments.aggregate([
  { $project: {
      stdDev: { $stdDevSamp: "$measurements" }
    }}
]).toArray();
console.log('Expression result:', expressionResult);

// Window operator usage: running stdDevSamp within each partition
const sensorReadings = db.collection('sensorReadings');
await sensorReadings.insertMany([
  { _id: 1, sensor: "A", time: 1, value: 2 },
  { _id: 2, sensor: "A", time: 2, value: 4 },
  { _id: 3, sensor: "A", time: 3, value: 6 },
  { _id: 4, sensor: "B", time: 1, value: 10 },
  { _id: 5, sensor: "B", time: 2, value: 12 }
]);
const windowResult = await sensorReadings.aggregate([
  {
    $setWindowFields: {
```

```
          partitionBy: "$sensor",
          sortBy: { time: 1 },
          output: {
            runningStdDev: {
              $stdDevSamp: "$value",
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']

        # Accumulator usage: stdDevSamp across grouped documents
        scores = db['scores']
        scores.insert_many([
            { 'subject': 'math', 'score': 80 },
            { 'subject': 'math', 'score': 90 },
            { 'subject': 'math', 'score': 85 },
            { 'subject': 'math', 'score': 95 },
            { 'subject': 'science', 'score': 70 },
            { 'subject': 'science', 'score': 75 },
            { 'subject': 'science', 'score': 80 },
            { 'subject': 'science', 'score': 85 }
```

```
])
accumulator_result = list(scores.aggregate([
    { '$group': {
        '_id': '$subject',
        'stdDev': { '$stdDevSamp': '$score' }
    }}
]))
print('Accumulator result:', accumulator_result)

# Expression usage: stdDevSamp of an array field
experiments = db['experiments']
experiments.insert_many([
    { '_id': 1, 'measurements': [10, 12, 14, 16, 18] },
    { '_id': 2, 'measurements': [5, 5, 5, 5, 5] },
    { '_id': 3, 'measurements': [2, 4, 6, 8, 10] }
])
expression_result = list(experiments.aggregate([
    { '$project': {
        'stdDev': { '$stdDevSamp': '$measurements' }
    }}
]))
print('Expression result:', expression_result)

# Window operator usage: running stdDevSamp within each partition
sensor_readings = db['sensorReadings']
sensor_readings.insert_many([
    { '_id': 1, 'sensor': 'A', 'time': 1, 'value': 2 },
    { '_id': 2, 'sensor': 'A', 'time': 2, 'value': 4 },
    { '_id': 3, 'sensor': 'A', 'time': 3, 'value': 6 },
    { '_id': 4, 'sensor': 'B', 'time': 1, 'value': 10 },
    { '_id': 5, 'sensor': 'B', 'time': 2, 'value': 12 }
])
window_result = list(sensor_readings.aggregate([
    {
        '$setWindowFields': {
            'partitionBy': '$sensor',
            'sortBy': { 'time': 1 },
            'output': {
                'runningStdDev': {
                    '$stdDevSamp': '$value',
                    'window': { 'documents': ['unbounded', 'current'] }
                }
            }
        }
```

```
            }
        ]))
        print('Window result:', window_result)

    finally:
        client.close()

example()
```

$strLenBytes

The `$strLenBytes` operator in Amazon DocumentDB is used to determine the length of a string in bytes. This is useful when you need to understand the storage size of a string field, especially when dealing with Unicode characters that may use more than one byte per character.

Parameters

- `expression`: The string expression to calculate the length of.

Example (MongoDB Shell)

This example demonstrates how to use the `$strLenBytes` operator to calculate the length of string fields in bytes.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-BVV-021" },
  { "_id": 2, "Desk": "Munich-HGG-32a" },
  { "_id": 3, "Desk": "Cologne-ayu-892.50" },
  { "_id": 4, "Desk": "Dortmund-Hop-78" }
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      "Desk": 1,
      "length": { $strLenBytes: "$Desk" }
    }
```

```
  }
])
```

Output

```
{ "_id" : 1, "Desk" : "Düsseldorf-BVV-021", "length" : 19 }
{ "_id" : 2, "Desk" : "Munich-HGG-32a", "length" : 14 }
{ "_id" : 3, "Desk" : "Cologne-ayu-892.50", "length" : 18 }
{ "_id" : 4, "Desk" : "Dortmund-Hop-78", "length" : 15 }
```

Note that the length of the "Düsseldorf-BVV-021" string is 19 bytes, which is different from the number of code points (18) due to the Unicode character "Ü" occupying 2 bytes.

Code examples

To view a code example for using the `$strLenBytes` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    {
      $project: {
        "Desk": 1,
        "length": { $strLenBytes: "$Desk" }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.people

    result = list(collection.aggregate([
        {
            '$project': {
                "Desk": 1,
                "length": { "$strLenBytes": "$Desk" }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$strLenCP

The `$strLenCP` operator in Amazon DocumentDB is used to determine the length of a string expression in code points (Unicode characters). This is useful when you need to know the number of characters in a string, rather than the number of bytes.

Parameters

- `expression`: The string expression for which to return the length in code points.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$strLenCP` operator to determine the length of strings with Unicode characters.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-BVV-021" },
  { "_id": 2, "Desk": "Munich-HGG-32a" },
  { "_id": 3, "Desk": "Cologne-ayu-892.50" },
  { "_id": 4, "Desk": "Dortmund-Hop-78" }
])
```

Query example

```
db.people.aggregate([
  {
    $project: {
      "Desk": 1,
      "length": { $strLenCP: "$Desk" }
    }
  }
])
```

Output

```
{ "_id" : 1, "Desk" : "Düsseldorf-BVV-021", "length" : 18 }
{ "_id" : 2, "Desk" : "Munich-HGG-32a", "length" : 14 }
{ "_id" : 3, "Desk" : "Cologne-ayu-892.50", "length" : 18 }
{ "_id" : 4, "Desk" : "Dortmund-Hop-78", "length" : 15 }
```

Notice the difference in the length measurement for the "Düsseldorf-BVV-021" string, which contains a Unicode character (Ü). The `$strLenCP` operator correctly counts the number of Unicode characters, while the `$strLenBytes` operator counts the number of bytes.

Code examples

To view a code example for using the `$strLenCP` command, choose the tab for the language that you want to use:

Node.js

Here's an example of using the `$strLenCP` operator in a Node.js application with the MongoDB driver:

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    {
      $project: {
        "Desk": 1,
        "length": { $strLenCP: "$Desk" }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

Here's an example of using the `$strLenCP` operator in a Python application with the PyMongo driver:

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.people

    result = list(collection.aggregate([
        {
            '$project': {
                "Desk": 1,
                "length": { "$strLenCP": "$Desk" }
```

```
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$strcasecmp

The `$strcasecmp` operator in Amazon DocumentDB performs a case-insensitive comparison between two strings. It returns an integer value indicating the lexicographic comparison of the two input strings, ignoring case differences.

Parameters

- `string1`: The first string to compare.
- `string2`: The second string to compare.

Example (MongoDB Shell)

This example demonstrates how to use the `$strcasecmp` operator to compare desk location strings in a `people` collection, ignoring case differences.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "mke233-wi" },
  { "_id": 2, "Desk": "MKE233-WI" },
  { "_id": 3, "Desk": "mke233-wi" }
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      item: 1,
```

```
      compare: { $strcasecmp: ["$Desk", "mke233-wi"] }
    }
  }
]);
```

Output

```
{ "_id" : 1, "compare" : 0 }
{ "_id" : 2, "compare" : 0 }
{ "_id" : 3, "compare" : 0 }
```

The output shows that the comparison between the "Desk" field and the string "mke233-wi" returns 0 for all three documents, indicating that the strings are equal when case is ignored.

Code examples

To view a code example for using the $strcasecmp command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    {
      $project: {
        item: 1,
        compare: { $strcasecmp: ["$Desk", "mke233-wi"] }
      }
    }
  ]).toArray();

  console.log(result);
```

```
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.people

    result = list(collection.aggregate([
        {
            '$project': {
                'item': 1,
                'compare': { '$strcasecmp': ['$Desk', 'mke233-wi'] }
            }
        }
    ]))

    print(result)

    client.close()

example()
```

$substr

The `$substr` operator in Amazon DocumentDB is used to extract a substring from a given string. It is particularly useful when you need to define substrings based on a range of characters, rather than a range of bytes. This is especially important when dealing with Unicode strings, where the number of bytes used to represent a character can vary.

Parameters

- `string`: The input string from which to extract the substring.

- `start`: The starting position (zero-based) of the substring to be extracted. Can be a non-negative integer expression.
- `length`: The number of characters in the extracted substring. Can be a non-negative integer expression.

Example (MongoDB Shell)

In this example, we'll demonstrate the use of `$substr` to extract the state abbreviation from an employee's desk location.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-NRW-021" },
  { "_id": 2, "Desk": "Bremerhaven-HBB-32a" },
  { "_id": 3, "Desk": "Norderstedt-SHH-892.50" },
  { "_id": 4, "Desk": "Brandenburg-BBB-78" }
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      "state": { $substr: ["$Desk", 12, 3] }
    }
  }
])
```

Output

```
{ "_id": 1, "state": "NRW" },
{ "_id": 2, "state": "HBB" },
{ "_id": 3, "state": "SHH" },
{ "_id": 4, "state": "BBB" }
```

Code examples

To view a code example for using the `$substr` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require("mongodb");

async function example() {
  const client = await MongoClient.connect("mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false");
  const db = client.db("test");
  const collection = db.collection("people");

  const result = await collection.aggregate([
    {
      $project: {
        "state": { $substrCP: ["$Desk", 12, 3] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient("mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false")
    db = client["test"]
    collection = db["people"]

    result = list(collection.aggregate([
        {
            "$project": {
                "state": { "$substrCP": ["$Desk", 12, 3] }
            }
        }
    ]))
```

```
    print(result)
    client.close()

example()
```

$substrBytes

The $substrBytes operator in Amazon DocumentDB is used to extract a substring from a string based on a specified byte range. This operator is useful when you need to extract a substring from a string and the number of bytes required to represent each character in the string is important.

Unlike $substrCP, which operates on the number of Unicode code points, $substrBytes operates on the number of bytes required to represent the characters in the string. This can be particularly useful when working with strings that contain non-ASCII characters, as these characters may require more than one byte to represent.

Note: $substr has been deprecated since version 3.4. $substr is now an alias for $substrBytes.

Parameters

- string: The input string from which the substring will be extracted.
- startByte: The zero-based starting byte position of the substring to be extracted. A negative value can be used to specify a position from the end of the string.
- length: The number of bytes in the substring to be extracted.

Example (MongoDB Shell)

In this example, we'll use $substrBytes to extract a substring from a string that contains non-ASCII characters.

Create sample documents

```
db.people.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-NRW-021" },
  { "_id": 2, "Desk": "Bremerhaven-HBB-32a" },
  { "_id": 3, "Desk": "Norderstedt-SHH-892.50" },
  { "_id": 4, "Desk": "Brandenburg-BBB-78" }
```

```
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      "state": { $substrBytes: [ "$Desk", 12, 3] }
    }
  }
])
```

Output

```
{ "_id": 1, "state": "NRW" },
{ "_id": 2, "state": "HBB" },
{ "_id": 3, "state": "SHH" },
{ "_id": 4, "state": "BBB" }
```

In this example, we use `$substrBytes` to extract a 3-byte substring starting from the 12th byte of the `Desk` field. This allows us to extract the 2-character state abbreviation, even though the string may contain non-ASCII characters.

Code examples

To view a code example for using the `$substrBytes` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const people = db.collection('people');

  const result = await people.aggregate([
    {
      $project: {
```

```
        "state": { $substrBytes: ["$Desk", 12, 3] }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    people = db.people

    result = list(people.aggregate([
        {
            '$project': {
                "state": { '$substrBytes': ["$Desk", 12, 3] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$substrCP

The `$substrCP` operator in Amazon DocumentDB is used to extract a substring from a string, where the substring is specified as a range of UTF-8 code points (CP). This operator is particularly useful when working with Unicode strings, as it allows you to extract substrings without having to worry about the underlying byte representation of the characters.

Unlike the `$substrBytes` operator, which operates on byte positions, the `$substrCP` operator works with code point positions. This makes it easier to work with strings that contain non-ASCII characters, as the number of code points may not match the number of bytes or characters.

Parameters

- `string`: The input string from which to extract the substring.
- `start`: The starting code point position (zero-based) from which to extract the substring.
- `length`: The number of code points to extract.

Example (MongoDB Shell)

In this example, we'll use the `$substrCP` operator to extract the state abbreviation from a string containing the employee's desk location.

Create sample documents

```
db.people.insert([
  { "_id": 1, "first_name": "Jane", "last_name": "Doe", "Desk": "12 Main St,
 Minneapolis, MN 55401" },
  { "_id": 2, "first_name": "John", "last_name": "Doe", "Desk": "456 Oak Rd, New
 Orleans, LA 70032" },
  { "_id": 3, "first_name": "Steve", "last_name": "Smith", "Desk": "789 Elm Ln,
 Bakersfield, CA 93263" }
]);
```

Query example

```
db.people.aggregate([
  {
    $project: {
      "state": { $substrCP: ["$Desk", 25, 2] }
    }
  }
]);
```

Output

```
{ "_id" : 1, "state" : "MN" }
{ "_id" : 2, "state" : "LA" }
```

```
{ "_id" : 3, "state" : "CA" }
```

In this example, we know that the state abbreviation starts at the 25th code point in the `Desk` field and is 2 code points long. By using the `$substrCP` operator, we can extract the state abbreviation without having to worry about the underlying byte representation of the string.

Code examples

To view a code example for using the `$substrCP` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findStates() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const result = await db.collection('people').aggregate([
    {
      $project: {
        "state": { $substrCP: ["$Desk", 25, 2] }
      }
    }
  ]).toArray();
  console.log(result);
  client.close();
}

findStates();
```

Python

```
from pymongo import MongoClient

def find_states():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
```

```
    result = list(db.people.aggregate([
        {
            '$project': {
                'state': { '$substrCP': ['$Desk', 25, 2] }
            }
        }
    ]))
    print(result)
    client.close()

find_states()
```

In both the Node.js and Python examples, we use the `$substrCP` operator to extract the state abbreviation from the `Desk` field, similar to the MongoDB Shell example.

$subtract

The `$subtract` operator in Amazon DocumentDB is used to subtract values. It can be used to subtract dates, numbers, or a combination of both. This operator is useful for calculating the difference between two dates or subtracting a value from a number.

Parameters

- `expression1`: The first value to be subtracted.
- `expression2`: The second value to be subtracted from `<expression1>`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$subtract` operator to calculate the difference between two dates.

Create sample document

```
db.dates.insert([
  {
  "_id": 1,
  "startDate": ISODate("2023-01-01T00:00:00Z"),
  "endDate": ISODate("2023-01-05T12:00:00Z")
  }
]);
```

Query example

```
db.dates.aggregate([
  {
    $project: {
      _id: 1,
      durationDays: {
        $divide: [
          { $subtract: ["$endDate", "$startDate"] },
          1000 * 60 * 60 * 24  // milliseconds in a day
        ]
      }
    }
  }
]);
```

Output

```
[ { _id: 1, durationDays: 4.5 } ]
```

In this example, the `$subtract` operator is used to calculate the difference between the `$endDate` and `$startDate` in days.

Code examples

To view a code example for using the `$subtract` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('dates');
```

```
    const pipeline = [
      {
        $project: {
          _id: 1,
          durationDays: {
            $divide: [
              { $subtract: ["$endDate", "$startDate"] },
              1000 * 60 * 60 * 24  // Convert milliseconds to days
            ]
          }
        }
      }
    ];

    const results = await collection.aggregate(pipeline).toArray();

    console.dir(results, { depth: null });

  } finally {
    await client.close();
  }
}

example().catch(console.error);
```

Python

```
from datetime import datetime, timedelta
from pymongo import MongoClient

def example():

    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client.test
        collection = db.dates

        pipeline = [
            {
                "$project": {
```

```
                    "_id": 1,
                    "durationDays": {
                        "$divide": [
                            { "$subtract": ["$endDate", "$startDate"] },
                            1000 * 60 * 60 * 24  # Convert milliseconds to days
                        ]
                    }
                }
            }
        ]

        results = collection.aggregate(pipeline)

        for doc in results:
            print(doc)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        client.close()

example()
```

$sum

The $sum operator in Amazon DocumentDB returns the sum of the specified expression for each document in a group. It is a group accumulator operator that is typically used in the $group stage of an aggregation pipeline to perform summation calculations.

Parameters

- expression: The numeric expression to sum. This can be a field path, an expression, or a constant.

Example (MongoDB Shell)

The following example demonstrates the use of the $sum operator to calculate the total sales for each product.

Create sample documents

```
db.sales.insertMany([
  { product: "abc", price: 10, quantity: 2 },
  { product: "abc", price: 10, quantity: 3 },
  { product: "xyz", price: 20, quantity: 1 },
  { product: "xyz", price: 20, quantity: 5 }
]);
```

Query example

```
db.sales.aggregate([
  { $group: {
      _id: "$product",
      totalSales: { $sum: { $multiply: [ "$price", "$quantity" ] } }
    }}
]);
```

Output

```
[
  { "_id": "abc", "totalSales": 50 },
  { "_id": "xyz", "totalSales": 120 }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The `$sum` operator can also be used as a window operator in the `$setWindowFields` stage. In this context, it returns the sum of the specified expression for the documents in each window. You specify the operator under the `output` field, and optionally define the window boundaries with a `window` document.

Create sample documents

```
db.dailySales.insertMany([
  { _id: 1, product: "abc", day: 1, amount: 10 },
  { _id: 2, product: "abc", day: 2, amount: 20 },
  { _id: 3, product: "abc", day: 3, amount: 15 },
  { _id: 4, product: "xyz", day: 1, amount: 30 },
  { _id: 5, product: "xyz", day: 2, amount: 25 }
]);
```

Query example

The following example partitions the documents by `product`, sorts each partition by `day`, and returns a running total of `amount` from the start of the partition through the current document.

```
db.dailySales.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$product",
      sortBy: { day: 1 },
      output: {
        runningTotal: {
          $sum: "$amount",
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "product": "abc", "day": 1, "amount": 10, "runningTotal": 10 },
  { "_id": 2, "product": "abc", "day": 2, "amount": 20, "runningTotal": 30 },
  { "_id": 3, "product": "abc", "day": 3, "amount": 15, "runningTotal": 45 },
  { "_id": 4, "product": "xyz", "day": 1, "amount": 30, "runningTotal": 30 },
  { "_id": 5, "product": "xyz", "day": 2, "amount": 25, "runningTotal": 55 }
]
```

Each document is augmented with `runningTotal`, the cumulative sum of `amount` within its partition up to and including the current document.

Code examples

To view a code example for using the `$sum` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: sum per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { product: "abc", price: 10, quantity: 2 },
      { product: "abc", price: 10, quantity: 3 },
      { product: "xyz", price: 20, quantity: 1 },
      { product: "xyz", price: 20, quantity: 5 }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: {
          _id: "$product",
          totalSales: { $sum: { $multiply: [ "$price", "$quantity" ] } }
        }}
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: running total within each partition
    const dailySales = db.collection('dailySales');
    await dailySales.insertMany([
      { _id: 1, product: "abc", day: 1, amount: 10 },
      { _id: 2, product: "abc", day: 2, amount: 20 },
      { _id: 3, product: "abc", day: 3, amount: 15 },
      { _id: 4, product: "xyz", day: 1, amount: 30 },
      { _id: 5, product: "xyz", day: 2, amount: 25 }
    ]);
    const windowResult = await dailySales.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$product",
          sortBy: { day: 1 },
          output: {
            runningTotal: {
```

```
              $sum: "$amount",
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client.test

        # Accumulator usage: sum per group
        sales = db.sales
        sales.insert_many([
            { 'product': 'abc', 'price': 10, 'quantity': 2 },
            { 'product': 'abc', 'price': 10, 'quantity': 3 },
            { 'product': 'xyz', 'price': 20, 'quantity': 1 },
            { 'product': 'xyz', 'price': 20, 'quantity': 5 }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': {
                '_id': '$product',
```

```
                'totalSales': { '$sum': { '$multiply': [ '$price', '$quantity' ] } }
            }}
        ]))
        pprint(accumulator_result)

        # Window operator usage: running total within each partition
        daily_sales = db.dailySales
        daily_sales.insert_many([
            { '_id': 1, 'product': 'abc', 'day': 1, 'amount': 10 },
            { '_id': 2, 'product': 'abc', 'day': 2, 'amount': 20 },
            { '_id': 3, 'product': 'abc', 'day': 3, 'amount': 15 },
            { '_id': 4, 'product': 'xyz', 'day': 1, 'amount': 30 },
            { '_id': 5, 'product': 'xyz', 'day': 2, 'amount': 25 }
        ])
        window_result = list(daily_sales.aggregate([
            {
                '$setWindowFields': {
                    'partitionBy': '$product',
                    'sortBy': { 'day': 1 },
                    'output': {
                        'runningTotal': {
                            '$sum': '$amount',
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$switch

New from version 4.0.

Not supported by Elastic cluster.

The `$switch` operator is a conditional expression operator in Amazon DocumentDB that allows you to evaluate a list of case expressions and return the value of the first case that evaluates to true, or a default value if no case expression is true.

Parameters

- `branches`: An array of documents, each of which has a case field that contains the boolean expression to evaluate, and a then field that contains the value to return if the case expression is true.
- `default`: (optional) The value to return if none of the case expressions are true.

Example (MongoDB Shell)

The following example demonstrates the use of the `$switch` operator to determine the shipping cost for an order based on the order total.

Create sample documents

```
db.orders.insertMany([
  { _id: 1, total: 50 },
  { _id: 2, total: 150 },
  { _id: 3, total: 250 }
]);
```

Query example

```
db.orders.aggregate([
  {
    $project: {
      _id: 1,
      total: 1,
      shippingCost: {
        $switch: {
          branches: [
            { case: { $lte: ["$total", 100] }, then: 5 },
            { case: { $lte: ["$total", 200] }, then: 10 },
            { case: { $gt: ["$total", 200] }, then: 15 }
          ],
          default: 0
```

```
        }
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "total": 50,
    "shippingCost": 5
  },
  {
    "_id": 2,
    "total": 150,
    "shippingCost": 10
  },
  {
    "_id": 3,
    "total": 250,
    "shippingCost": 15
  }
]
```

Code examples

To view a code example for using the `$switch` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('orders');

  const result = await collection.aggregate([
```

```
    {
      $project: {
        _id: 1,
        total: 1,
        shippingCost: {
          $switch: {
            branches: [
              { case: { $lte: ['$total', 100] }, then: 5 },
              { case: { $lte: ['$total', 200] }, then: 10 },
              { case: { $gt: ['$total', 200] }, then: 15 }
            ],
            default: 0
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.orders

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'total': 1,
                'shippingCost': {
                    '$switch': {
                        'branches': [
```

```
                            { 'case': { '$lte': ['$total', 100] }, 'then': 5 },
                            { 'case': { '$lte': ['$total', 200] }, 'then': 10 },
                            { 'case': { '$gt': ['$total', 200] }, 'then': 15 }
                        ],
                        'default': 0
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

if __name__ == '__main__':
    main()
```

$tan

New from version 8.0.1.

The `$tan` operator in Amazon DocumentDB returns the tangent of a value that is measured in radians.

The input expression must resolve to a numeric value. If your value is in degrees, apply `$degreesToRadians` before passing it to `$tan`.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Parameters

- `expression`: An expression that resolves to a number in radians.

Behavior

null, NaN, and +/- Infinity

Example	Results
`{ $tan: NaN }`	`NaN`

Example	Results
`{ $tan: null }`	`null`
`{ $tan: Infinity }` or `{ $tan: -Infinity }`	Throws an error.

When the input is `null` or the referenced field is missing, `$tan` returns `null`. An input of `NaN` produces `NaN`. Positive or negative infinity causes an error because tangent is undefined at those values.

Example (MongoDB Shell)

The following example shows how to use the `$tan` operator to calculate the tangent of an angle in radians.

Create sample documents

```
db.angles.insertMany([
  { "_id": 1, "angle": 0 },
  { "_id": 2, "angle": 0.7853981633974483 },
  { "_id": 3, "angle": -0.7853981633974483 }
]);
```

Query example

```
db.angles.aggregate([
  { $project: {
    "tangent": { $tan: "$angle" }
  }}
]);
```

Output

```
[
  { "_id": 1, "tangent": 0 },
  { "_id": 2, "tangent": 0.9999999999999999 },
  { "_id": 3, "tangent": -0.9999999999999999 }
]
```

Code examples

To view a code example for using the $tan operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('angles');

  const result = await collection.aggregate([
    { $project: {
      "tangent": { $tan: "$angle" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['angles']

    result = list(collection.aggregate([
        { '$project': {
            'tangent': { '$tan': '$angle' }
        }}
```

```
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$tanh

New from version 8.0.1.

The `$tanh` operator in Amazon DocumentDB returns the hyperbolic tangent of a value.

The input expression must resolve to a numeric value.

The return type is `double` by default. If the input is a 128-bit decimal, the output is also a 128-bit decimal.

Parameters

- `expression`: An expression that resolves to a number.

Behavior

null, NaN, and +/- Infinity

Example	Results
`{ $tanh: NaN }`	`NaN`
`{ $tanh: null }`	`null`
`{ $tanh: -Infinity }`	`-1`
`{ $tanh: Infinity }`	`1`

When the input is `null` or the referenced field is missing, `$tanh` returns `null`. An input of `NaN` produces `NaN`. For positive infinity the result is `1`; for negative infinity the result is `-1`.

Example (MongoDB Shell)

The following example shows how to use the $tanh operator to calculate the hyperbolic tangent of a value.

Create sample documents

```
db.values.insertMany([
  { "_id": 1, "value": 0 },
  { "_id": 2, "value": 1 },
  { "_id": 3, "value": -1 }
]);
```

Query example

```
db.values.aggregate([
  { $project: {
    "result": { $tanh: "$value" }
  }}
]);
```

Output

```
[
  { "_id": 1, "result": 0 },
  { "_id": 2, "result": 0.7615941559557649 },
  { "_id": 3, "result": -0.7615941559557649 }
]
```

Code examples

To view a code example for using the $tanh operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('values');

  const result = await collection.aggregate([
    { $project: {
      "result": { $tanh: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['values']

    result = list(collection.aggregate([
        { '$project': {
            'result': { '$tanh': '$value' }
        }}
    ]))

    print(result)
    client.close()

if __name__ == "__main__":
    main()
```

$top

New from version 8.0.1.

Use the `$top` accumulator in the `$group` stage to return the highest-ranked document per group according to a specified sort order.

Parameters

- `sortBy`: A document specifying the sort order. Use `1` for ascending or `-1` for descending.
- `output`: An expression that specifies the fields to return from the top document.

Example (MongoDB Shell)

The following example shows how to use the `$top` accumulator to find the top sale (highest quantity) per item in a sales collection.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, price: 5 },
  { item: "abc", quantity: 5, price: 8 },
  { item: "xyz", quantity: 15, price: 3 },
  { item: "xyz", quantity: 7, price: 6 }
])
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", topSale: { $top: { sortBy: { quantity: -1 }, output:
 { quantity: "$quantity", price: "$price" } } } } }
])
```

Output

```
[
  { "_id": "xyz", "topSale": { "quantity": 15, "price": 3 } },
  { "_id": "abc", "topSale": { "quantity": 10, "price": 5 } }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The $top operator can also be used as a window operator in the $setWindowFields stage. In this context, it returns the output from the top-ranked document (according to the operator's own sortBy) among the documents in each window. You specify the operator under the output field, and optionally define the window boundaries with a window document.

> **Note**
>
> When used as a window operator in $setWindowFields, $top is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.matchScores.insertMany([
  { _id: 1, player: "Alice", round: 1, score: 20 },
  { _id: 2, player: "Alice", round: 2, score: 35 },
  { _id: 3, player: "Alice", round: 3, score: 28 },
  { _id: 4, player: "Bob", round: 1, score: 15 },
  { _id: 5, player: "Bob", round: 2, score: 40 }
]);
```

Query example

The following example partitions the documents by player, sorts each partition by round, and returns the highest-scoring round seen from the start of the partition through the current document.

```
db.matchScores.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$player",
      sortBy: { round: 1 },
      output: {
        bestRoundSoFar: {
          $top: { sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
```

```
]);
```

Output

```
[
  { "_id": 1, "player": "Alice", "round": 1, "score": 20, "bestRoundSoFar": { "round":
 1, "score": 20 } },
  { "_id": 2, "player": "Alice", "round": 2, "score": 35, "bestRoundSoFar": { "round":
 2, "score": 35 } },
  { "_id": 3, "player": "Alice", "round": 3, "score": 28, "bestRoundSoFar": { "round":
 2, "score": 35 } },
  { "_id": 4, "player": "Bob", "round": 1, "score": 15, "bestRoundSoFar": { "round": 1,
 "score": 15 } },
  { "_id": 5, "player": "Bob", "round": 2, "score": 40, "bestRoundSoFar": { "round": 2,
 "score": 40 } }
]
```

Each document is augmented with `bestRoundSoFar`, the top-scoring round within its partition up to and including the current document.

Code examples

To view a code example for using the `$top` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: top document per group
    const sales = db.collection('sales');
```

```
    await sales.insertMany([
      { item: "abc", quantity: 10, price: 5 },
      { item: "abc", quantity: 5, price: 8 },
      { item: "xyz", quantity: 15, price: 3 },
      { item: "xyz", quantity: 7, price: 6 }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", topSale: { $top: { sortBy: { quantity: -1 }, output:
 { quantity: "$quantity", price: "$price" } } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: best round so far within each partition
    const matchScores = db.collection('matchScores');
    await matchScores.insertMany([
      { _id: 1, player: "Alice", round: 1, score: 20 },
      { _id: 2, player: "Alice", round: 2, score: 35 },
      { _id: 3, player: "Alice", round: 3, score: 28 },
      { _id: 4, player: "Bob", round: 1, score: 15 },
      { _id: 5, player: "Bob", round: 2, score: 40 }
    ]);
    const windowResult = await matchScores.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$player",
          sortBy: { round: 1 },
          output: {
            bestRoundSoFar: {
              $top: { sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}
```

```
example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: top document per group
        sales = db['sales']
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'price': 5 },
            { 'item': 'abc', 'quantity': 5, 'price': 8 },
            { 'item': 'xyz', 'quantity': 15, 'price': 3 },
            { 'item': 'xyz', 'quantity': 7, 'price': 6 }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'topSale': { '$top': { 'sortBy':
 { 'quantity': -1 }, 'output': { 'quantity': '$quantity', 'price':
 '$price' } } } } }
        ]))
        pprint(accumulator_result)

        # Window operator usage: best round so far within each partition
        match_scores = db['matchScores']
        match_scores.insert_many([
            { '_id': 1, 'player': 'Alice', 'round': 1, 'score': 20 },
            { '_id': 2, 'player': 'Alice', 'round': 2, 'score': 35 },
            { '_id': 3, 'player': 'Alice', 'round': 3, 'score': 28 },
            { '_id': 4, 'player': 'Bob', 'round': 1, 'score': 15 },
            { '_id': 5, 'player': 'Bob', 'round': 2, 'score': 40 }
        ])
        window_result = list(match_scores.aggregate([
            {
```

```
                '$setWindowFields': {
                    'partitionBy': '$player',
                    'sortBy': { 'round': 1 },
                    'output': {
                        'bestRoundSoFar': {
                            '$top': { 'sortBy': { 'score': -1 }, 'output':
 { 'round': '$round', 'score': '$score' } },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$topN

New from version 8.0.1.

Use the `$topN` accumulator in the `$group` stage to return the top N elements within a group according to a specified sort order. If a group contains fewer than N elements, `$topN` returns all elements in the group.

Parameters

- n: A positive integer, or an expression that resolves to one, specifying how many top results to return per group.
- `sortBy`: A document specifying the sort order. Use `1` for ascending or `-1` for descending.
- `output`: An expression that specifies the fields to return from each of the top N documents.

Example (MongoDB Shell)

The following example shows how to use the $topN accumulator to find the top 2 sales (highest quantity) per item in a sales collection.

Create sample documents

```
db.sales.insertMany([
  { item: "abc", quantity: 10, price: 5 },
  { item: "abc", quantity: 7, price: 8 },
  { item: "abc", quantity: 5, price: 10 },
  { item: "xyz", quantity: 15, price: 3 },
  { item: "xyz", quantity: 9, price: 6 },
  { item: "xyz", quantity: 3, price: 12 }
])
```

Query example

```
db.sales.aggregate([
  { $group: { _id: "$item", topTwoSales: { $topN: { n: 2, sortBy: { quantity: -1 },
 output: { quantity: "$quantity", price: "$price" } } } } }
])
```

Output

```
[
  { "_id": "xyz", "topTwoSales": [{ "quantity": 15, "price": 3 }, { "quantity": 9,
 "price": 6 }] },
  { "_id": "abc", "topTwoSales": [{ "quantity": 10, "price": 5 }, { "quantity": 7,
 "price": 8 }] }
]
```

Window operator usage example (MongoDB Shell)

New from version 8.0.2.

The $topN operator can also be used as a window operator in the $setWindowFields stage. In this context, it returns an array of the output from the top N documents (according to the operator's own sortBy) among the documents in each window. You specify the operator under the output field, and optionally define the window boundaries with a window document.

> **Note**
>
> When used as a window operator in `$setWindowFields`, `$topN` is limited to 100 MB of intermediate data. An operation that exceeds this limit returns an error.

Create sample documents

```
db.matchScores.insertMany([
  { _id: 1, player: "Alice", round: 1, score: 20 },
  { _id: 2, player: "Alice", round: 2, score: 35 },
  { _id: 3, player: "Alice", round: 3, score: 28 },
  { _id: 4, player: "Bob", round: 1, score: 15 },
  { _id: 5, player: "Bob", round: 2, score: 40 }
]);
```

Query example

The following example partitions the documents by `player`, sorts each partition by `round`, and returns the two highest-scoring rounds seen from the start of the partition through the current document.

```
db.matchScores.aggregate([
  {
    $setWindowFields: {
      partitionBy: "$player",
      sortBy: { round: 1 },
      output: {
        topTwoRoundsSoFar: {
          $topN: { n: 2, sortBy: { score: -1 }, output: { round: "$round", score:
 "$score" } },
          window: { documents: ["unbounded", "current"] }
        }
      }
    }
  }
]);
```

Output

```
[
```

```
  { "_id": 1, "player": "Alice", "round": 1, "score": 20, "topTwoRoundsSoFar":
 [{ "round": 1, "score": 20 }] },
  { "_id": 2, "player": "Alice", "round": 2, "score": 35, "topTwoRoundsSoFar":
 [{ "round": 2, "score": 35 }, { "round": 1, "score": 20 }] },
  { "_id": 3, "player": "Alice", "round": 3, "score": 28, "topTwoRoundsSoFar":
 [{ "round": 2, "score": 35 }, { "round": 3, "score": 28 }] },
  { "_id": 4, "player": "Bob", "round": 1, "score": 15, "topTwoRoundsSoFar":
 [{ "round": 1, "score": 15 }] },
  { "_id": 5, "player": "Bob", "round": 2, "score": 40, "topTwoRoundsSoFar":
 [{ "round": 2, "score": 40 }, { "round": 1, "score": 15 }] }
]
```

Each document is augmented with `topTwoRoundsSoFar`, the two highest-scoring rounds within its partition up to and including the current document. If the window contains fewer than two documents, all of them are returned.

Code examples

To view a code example for using the `$topN` operator, choose the tab for the language that you want to use. The following examples show both accumulator usage (in `$group`) and window operator usage (in `$setWindowFields`):

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');

    // Accumulator usage: top N documents per group
    const sales = db.collection('sales');
    await sales.insertMany([
      { item: "abc", quantity: 10, price: 5 },
      { item: "abc", quantity: 7, price: 8 },
      { item: "abc", quantity: 5, price: 10 },
```

```
      { item: "xyz", quantity: 15, price: 3 },
      { item: "xyz", quantity: 9, price: 6 },
      { item: "xyz", quantity: 3, price: 12 }
    ]);
    const accumulatorResult = await sales.aggregate([
      { $group: { _id: "$item", topTwoSales: { $topN: { n: 2, sortBy: { quantity:
 -1 }, output: { quantity: "$quantity", price: "$price" } } } } }
    ]).toArray();
    console.log('Accumulator result:', accumulatorResult);

    // Window operator usage: top two rounds so far within each partition
    const matchScores = db.collection('matchScores');
    await matchScores.insertMany([
      { _id: 1, player: "Alice", round: 1, score: 20 },
      { _id: 2, player: "Alice", round: 2, score: 35 },
      { _id: 3, player: "Alice", round: 3, score: 28 },
      { _id: 4, player: "Bob", round: 1, score: 15 },
      { _id: 5, player: "Bob", round: 2, score: 40 }
    ]);
    const windowResult = await matchScores.aggregate([
      {
        $setWindowFields: {
          partitionBy: "$player",
          sortBy: { round: 1 },
          output: {
            topTwoRoundsSoFar: {
              $topN: { n: 2, sortBy: { score: -1 }, output: { round: "$round",
 score: "$score" } },
              window: { documents: ["unbounded", "current"] }
            }
          }
        }
      }
    ]).toArray();
    console.log('Window result:', windowResult);

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}
```

```
example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

        db = client['test']

        # Accumulator usage: top N documents per group
        sales = db['sales']
        sales.insert_many([
            { 'item': 'abc', 'quantity': 10, 'price': 5 },
            { 'item': 'abc', 'quantity': 7, 'price': 8 },
            { 'item': 'abc', 'quantity': 5, 'price': 10 },
            { 'item': 'xyz', 'quantity': 15, 'price': 3 },
            { 'item': 'xyz', 'quantity': 9, 'price': 6 },
            { 'item': 'xyz', 'quantity': 3, 'price': 12 }
        ])
        accumulator_result = list(sales.aggregate([
            { '$group': { '_id': '$item', 'topTwoSales': { '$topN': { 'n': 2,
 'sortBy': { 'quantity': -1 }, 'output': { 'quantity': '$quantity', 'price':
 '$price' } } } } }
        ]))
        pprint(accumulator_result)

        # Window operator usage: top two rounds so far within each partition
        match_scores = db['matchScores']
        match_scores.insert_many([
            { '_id': 1, 'player': 'Alice', 'round': 1, 'score': 20 },
            { '_id': 2, 'player': 'Alice', 'round': 2, 'score': 35 },
            { '_id': 3, 'player': 'Alice', 'round': 3, 'score': 28 },
            { '_id': 4, 'player': 'Bob', 'round': 1, 'score': 15 },
            { '_id': 5, 'player': 'Bob', 'round': 2, 'score': 40 }
        ])
        window_result = list(match_scores.aggregate([
```

```
            {
                '$setWindowFields': {
                    'partitionBy': '$player',
                    'sortBy': { 'round': 1 },
                    'output': {
                        'topTwoRoundsSoFar': {
                            '$topN': { 'n': 2, 'sortBy': { 'score': -1 }, 'output':
 { 'round': '$round', 'score': '$score' } },
                            'window': { 'documents': ['unbounded', 'current'] }
                        }
                    }
                }
            }
        ]))
        pprint(window_result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$trunc

New from version 8.0.1.

The `$trunc` operator in Amazon DocumentDB truncates a number to a specified decimal place, removing digits without rounding.

Parameters

- `number`: An expression that resolves to a number.
- `place`: Optional. An integer expression between -20 and 100 that specifies the number of decimal places to truncate to. A negative value truncates to the left of the decimal. Defaults to 0.

Example (MongoDB Shell)

The following example shows how to use the `$trunc` operator to truncate numeric values to one decimal place.

Truncating with a positive place value

```
db.measurements.insertMany([
  {_id: 1, value: 3.456},
  {_id: 2, value: 7.891},
  {_id: 3, value: 12.345}
]);
```

Query example

```
db.measurements.aggregate([
  { $project: { truncated: { $trunc: ["$value", 1] } } }
]);
```

Output

```
[
  {_id: 1, truncated: 3.4},
  {_id: 2, truncated: 7.8},
  {_id: 3, truncated: 12.3}
]
```

Truncating with a negative place value

When `place` is negative, `$trunc` replaces digits to the left of the decimal with zeros. The following example truncates values using the first digit to the left of the decimal.

```
db.samples.insertMany([
  {_id: 1, value: 19.25},
  {_id: 2, value: 28.73},
  {_id: 3, value: 34.32}
]);
```

```
db.samples.aggregate([
  { $project: { truncated: { $trunc: ["$value", -1] } } }
]);
```

Output

```
[
  {_id: 1, truncated: 10},
  {_id: 2, truncated: 20},
  {_id: 3, truncated: 30}
]
```

Code examples

To view a code example for using the `$trunc` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('measurements');
    const result = await collection.aggregate([
      { $project: { truncated: { $trunc: ["$value", 1] } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    try:
        db = client['test']
        collection = db['measurements']
        result = list(collection.aggregate([
            {'$project': {'truncated': {'$trunc': ['$value', 1]}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$tsIncrement

New from version 8.0.1.

The `$tsIncrement` operator in Amazon DocumentDB returns the incrementing ordinal from a Timestamp value as a long integer.

Parameters

- `expression`: An expression that resolves to a Timestamp.

Example (MongoDB Shell)

The following example shows how to use the `$tsIncrement` operator to extract the ordinal increment from Timestamp values.

Create sample documents

```
db.events.insertMany([
  {_id: 1, ts: Timestamp(1678900000, 1)},
  {_id: 2, ts: Timestamp(1678900000, 2)},
  {_id: 3, ts: Timestamp(1678900001, 1)}
]);
```

Query example

```
db.events.aggregate([
  { $project: { increment: { $tsIncrement: "$ts" } } }
]);
```

Output

```
[
  {_id: 1, increment: Long("1")},
  {_id: 2, increment: Long("2")},
  {_id: 3, increment: Long("1")}
]
```

Code examples

To view a code example for using the `$tsIncrement` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient, Timestamp } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('events');
    const result = await collection.aggregate([
      { $project: { increment: { $tsIncrement: "$ts" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    try:
        db = client['test']
        collection = db['events']
        result = list(collection.aggregate([
            {'$project': {'increment': {'$tsIncrement': '$ts'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$tsSecond

New from version 8.0.1.

The `$tsSecond` operator in Amazon DocumentDB returns the seconds portion of a Timestamp value as a long integer (the Unix epoch seconds).

Parameters

- `expression`: An expression that resolves to a Timestamp.

Example (MongoDB Shell)

The following example shows how to use the `$tsSecond` operator to extract the seconds portion from Timestamp values.

Create sample documents

```
db.events.insertMany([
  {_id: 1, ts: Timestamp(1678900000, 1)},
  {_id: 2, ts: Timestamp(1678900000, 2)},
  {_id: 3, ts: Timestamp(1678900001, 1)}
]);
```

Query example

```
db.events.aggregate([
  { $project: { seconds: { $tsSecond: "$ts" } } }
]);
```

Output

```
[
  {_id: 1, seconds: Long("1678900000")},
  {_id: 2, seconds: Long("1678900000")},
  {_id: 3, seconds: Long("1678900001")}
]
```

Code examples

To view a code example for using the $tsSecond operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient, Timestamp } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('events');
    const result = await collection.aggregate([
      { $project: { seconds: { $tsSecond: "$ts" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    try:
        db = client['test']
        collection = db['events']
        result = list(collection.aggregate([
            {'$project': {'seconds': {'$tsSecond': '$ts'}}}
        ]))
        print(result)
    finally:
        client.close()

example()
```

$toBool

The `$toBool` operator in Amazon DocumentDB converts an expression to a boolean value.

Parameters

- `expression`: The expression to be converted to a boolean value.

Note: Any string converts to `true`.

Example (MongoDB Shell)

The following example demonstrates using the `$toBool` operator to normalize device state values from different data types.

Create sample documents

```
db.deviceStates.insertMany([
  { _id: 1, deviceId: "sensor-001", status: true },
  { _id: 2, deviceId: "camera-002", status: 1 },
  { _id: 3, deviceId: "thermostat-003", status: "active" },
  { _id: 4, deviceId: "doorlock-004", status: 0 }
]);
```

Query example

```
db.deviceStates.aggregate([
  {
```

```
    $project: {
      _id: 1,
      deviceId: 1,
      isActive: { $toBool: "$status" }
    }
  }
]);
```

Output

```
[
  { "_id": 1, "deviceId": "sensor-001", "isActive": true },
  { "_id": 2, "deviceId": "camera-002", "isActive": true },
  { "_id": 3, "deviceId": "thermostat-003", "isActive": true },
  { "_id": 4, "deviceId": "doorlock-004", "isActive": false }
]
```

Code examples

To view a code example for using the `$toBool` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('deviceStates');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        deviceId: 1,
        isActive: { $toBool: '$status' }
      }
    }
  ]).toArray();
```

```
  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['deviceStates']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
                'deviceId': 1,
                'isActive': { '$toBool': '$status' }
            }
        }
    ]))

    print(result)

    client.close()

if __name__ == '__main__':
    main()
```

$toDate

New from version 4.0

The `$toDate` aggregation operator in Amazon DocumentDB is used to convert a date or date and time string to a BSON Date type. This is the inverse operation of the `$dateToString` operator.

Parameters

- `dateString`: A string representation of a date or date and time to be converted to a BSON Date type.
- `format`: (optional) A string that specifies the format of the `dateString`. If not provided, the operator will attempt to parse the `dateString` in various standard date and time formats.
- `timezone`: (optional) A string representing the time zone to use for the conversion. If not provided, the local time zone is used.

Example (MongoDB Shell)

The following example demonstrates how to use the `$toDate` operator to convert a date string to a BSON Date type.

Create sample documents

```
db.events.insertMany([
  { _id: 1, eventName: "Mission Start", eventTime: "2023-04-15T10:30:00Z" },
  { _id: 2, eventName: "Checkpoint Reached", eventTime: "2023-04-15T11:15:00Z" },
  { _id: 3, eventName: "Mission End", eventTime: "2023-04-15T12:00:00Z" }
]);
```

Query example

```
db.events.aggregate([
  {
    $project: {
      eventName: 1,
      eventTimeDate: { $toDate: "$eventTime" }
    }
  }
]);
```

Output

```
[
  {
    "_id": 1,
    "eventName": "Mission Start",
    "eventTimeDate": ISODate("2023-04-15T10:30:00Z")
```

```
  },
  {
    "_id": 2,
    "eventName": "Checkpoint Reached",
    "eventTimeDate": ISODate("2023-04-15T11:15:00Z")
  },
  {
    "_id": 3,
    "eventName": "Mission End",
    "eventTimeDate": ISODate("2023-04-15T12:00:00Z")
  }
]
```

Code examples

To view a code example for using the `$toDate` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('events');

  const result = await collection.aggregate([
    {
      $project: {
        eventName: 1,
        eventTimeDate: { $toDate: '$eventTime' }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['events']

    result = list(collection.aggregate([
        {
            '$project': {
                'eventName': 1,
                'eventTimeDate': { '$toDate': '$eventTime' }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$toDecimal

New from version 4.0

The `$toDecimal` operator in Amazon DocumentDB is used to convert a value to the Decimal128 data type. This is useful when you need to perform precise decimal arithmetic or handle large decimal values that cannot be accurately represented using the Double data type.

Parameters

- `expression`: The expression to convert to the Decimal128 data type.

Example (MongoDB Shell)

This example demonstrates how to use the `$toDecimal` operator to convert a string value to a Decimal128 data type.

Create sample documents

```
db.numbers.insertOne({ _id: 1, value: "3.14" });
db.numbers.insertOne({ _id: 2, value: "2.71" });
```

Query example

```
db.numbers.aggregate([
  { $project: {
    _id: 1,
    decimalValue: { $toDecimal: "$value" }
  }}
])
```

Output

```
[
  { "_id" : 1, "decimalValue" : Decimal128("3.14") },
  { "_id" : 2, "decimalValue" : Decimal128("2.71") }
]
```

Code examples

To view a code example for using the `$toDecimal` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      _id: 1,
      decimalValue: { $toDecimal: "$value" }
```

```
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.numbers

    result = list(collection.aggregate([
        {'$project': {
            '_id': 1,
            'decimalValue': {'$toDecimal': '$value'}
        }}
    ]))

    print(result)
    client.close()

example()
```

$toDouble

New from version 4.0

The `$toDouble` operator in Amazon DocumentDB is used to convert a value to a double-precision 64-bit floating-point number. This can be useful when you need to perform arithmetic operations on values that are not originally in a numeric format.

Parameters

<expression>: The expression to convert to a double value. This can be any valid expression that resolves to a numeric, string, or boolean value.

Example (MongoDB Shell)

This example demonstrates how to use the $toDouble operator to convert a string value to a numeric value for the purpose of performing a mathematical calculation.

Create sample documents

```
db.numbers.insertMany([
  { _id: 1, value: "10.5" },
  { _id: 2, value: "20.25" },
  { _id: 3, value: "7" }
])
```

Query example

```
db.numbers.aggregate([
  {
    $project: {
      _id: 1,
      value: 1,
      double_value: { $toDouble: "$value" },
      double_plus_five: { $add: [{ $toDouble: "$value" }, 5] }
    }
  }
])
```

Output

```
[
  { "_id" : 1, "value" : "10.5", "double_value" : 10.5, "double_plus_five" : 15.5 },
  { "_id" : 2, "value" : "20.25", "double_value" : 20.25, "double_plus_five" : 25.25 },
  { "_id" : 3, "value" : "7", "double_value" : 7.0, "double_plus_five" : 12.0 }
]
```

Code examples

To view a code example for using the $toDouble command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    {
      $project: {
        _id: 1,
        value: 1,
        double_value: { $toDouble: "$value" },
        double_plus_five: { $add: [{ $toDouble: "$value" }, 5] }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['numbers']

    result = list(collection.aggregate([
        {
            '$project': {
                '_id': 1,
```

```
                'value': 1,
                'double_value': { '$toDouble': '$value' },
                'double_plus_five': { '$add': [{ '$toDouble': '$value' }, 5] }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$toInt

New from version 4.0

The `$toInt` operator in Amazon DocumentDB is used to convert an input value to an integer data type. This operator is useful when you need to ensure that a field or expression is represented as an integer, which can be important for certain operations or data processing tasks.

Parameters

- `expression`: The expression to be converted to an integer.

Example (MongoDB Shell)

The following example demonstrates how to use the `$toInt` operator to convert a string value to an integer.

Create sample documents

```
db.numbers.insertMany([
  { "name": "one", "value": "1" },
  { "name": "hundred", "value": "100" }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
```

```
    "_id": 0,
    "name": 1,
    "intValue": { $toInt: "$value" }
  }}
]);
```

Output

```
{ "name": "one", "intValue": 1 }
{ "name": "hundred", "intValue": 100 }
```

Code examples

To view a code example for using the `$toInt` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      "_id": 0,
      "name": 1,
      "intValue": { $toInt: "$value" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['numbers']

    result = list(collection.aggregate([
        { "$project": {
            "_id": 0,
            "name": 1,
            "intValue": { "$toInt": "$value" }
        }}
    ]))

    print(result)
    client.close()

example()
```

$toLong

New from version 4.0

The `$toLong` operator in Amazon DocumentDB is used to convert a value to a 64-bit integer (long) data type. This can be useful when you need to perform arithmetic operations or comparisons on numeric values that may be stored as strings or other data types.

Parameters

- `expression`: The expression to convert to a 64-bit integer.

Example (MongoDB Shell)

This example demonstrates how to use the `$toLong` operator to convert a string value to a 64-bit integer.

Create sample documents

```
db.numbers.insertMany([
  { _id: 1, value: "42" },
  { _id: 3, value: "92233720368547758O7" }
]);
```

Query example

```
db.numbers.aggregate([
  {
    $project: {
      _id: 1,
      longValue: { $toLong: "$value" }
    }
  }
])
```

Output

```
[
  { "_id" : 1, "longValue" : 42 },
  { "_id" : 3, "longValue" : 9223372036854775807 }
]
```

Code examples

To view a code example for using the $toLong command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const numbers = db.collection('numbers');

  const result = await numbers.aggregate([
```

```
    {
      $project: {
        _id: 1,
        longValue: { $toLong: "$value" }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    numbers = db.numbers

    result = list(numbers.aggregate([
        {
            '$project': {
                '_id': 1,
                'longValue': { '$toLong': '$value' }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$toLower

The `$toLower` operator in Amazon DocumentDB is used to convert a string to lowercase.

Parameters

- `expression`: The string expression to convert to lowercase.

Example (MongoDB Shell)

The following example demonstrates how to use the `$toLower` operator to convert the `Desk` field to lowercase.

Create sample documents

```
db.locations.insertMany([
  { "_id": 1, "Desk": "Düsseldorf-BVV-021" },
  { "_id": 2, "Desk": "Munich-HGG-32a" }
]);
```

Query example

```
db.locations.aggregate([
  { $project: { item: { $toLower: "$Desk" } } }
]);
```

Output

```
{ "_id" : 1, "item" : "düsseldorf-bvv-021" }
{ "_id" : 2, "item" : "munich-hgg-32a" }
```

Code examples

To view a code example for using the `$toLower` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require("mongodb");

async function main() {
  const client = await MongoClient.connect("mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false");
```

```
  const db = client.db("test");
  const collection = db.collection("locations");

  const result = await collection.aggregate([
    { $project: { item: { $toLower: "$Desk" } } }
  ]).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient("mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false")
    db = client["test"]
    collection = db["locations"]

    result = list(collection.aggregate([
        { "$project": { "item": { "$toLower": "$Desk" } } }
    ]))

    print(result)

    client.close()

if __name__ == "__main__":
    main()
```

$toObjectId

New from version 4.0

The `$toObjectId` operator in Amazon DocumentDB is used to convert a string representation of an ObjectId to an actual ObjectId data type. This can be useful when working with data that

has been stored as string representations of ObjectIds, as it allows you to perform operations that require the ObjectId data type.

Parameters

- `expression`: A string expression representing a valid ObjectId.

Example (MongoDB Shell)

The following example demonstrates how to use the `$toObjectId` operator to convert a string representation of an ObjectId to the ObjectId data type.

Create sample documents

```
db.employees.insertMany([
  { _id: 1, empId:"64e5f8886218c620cf0e8f8a", name: "Carol Smith", employeeId:
 "c720a" },
  { _id: 2, empId:"64e5f94e6218c620cf0e8f8c", name: "Bill Taylor", employeeId:
 "c721a" }
]);
```

Query example

```
db.employees.aggregate([
  { $project: {
    "empIdAsObjectId": {$toObjectId: "$empId"}}
  }
]);
```

Output

```
[
  { _id: 1, empIdAsObjectId: ObjectId('64e5f8886218c620cf0e8f8a') },
  { _id: 2, empIdAsObjectId: ObjectId('64e5f94e6218c620cf0e8f8c') }
]
```

Code examples

To view a code example for using the `$toObjectId` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('employees');

  const result = await collection.aggregate([
    { $project: {
      "empIdAsObjectId": {$toObjectId: "$empId"}}
    }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['employees']

    result = list(collection.aggregate([
      { "$project": {
        "empIdAsObjectId": {"$toObjectId": "$empId"}}
      }
    ]))

    print(result)
    client.close()
```

```
example()
```

$toString

New from version 4.0

The `$toString` operator in Amazon DocumentDB is used to convert a value of any type (except for null) to a string representation. This can be useful when you need to perform string operations on values that are not originally in string format.

Parameters

- `expression`: The expression to convert to a string.

Example (MongoDB Shell)

The following example demonstrates how to use the `$toString` operator to convert numeric values to strings.

Create sample documents

```
db.numbers.insertMany([
  { "_id": 1, "value": 42 },
  { "_id": 2, "value": 3.14 }
]);
```

Query example

```
db.numbers.aggregate([
  { $project: {
    _id: 1,
    valueAsString: { $toString: "$value" }
  }}
]);
```

Output

```
{ "_id": 1, "valueAsString": "42" }
{ "_id": 2, "valueAsString": "3.14" }
```

Code examples

To view a code example for using the `$toString` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  const result = await collection.aggregate([
    { $project: {
      _id: 1,
      valueAsString: { $toString: '$value' }
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['numbers']

    result = list(collection.aggregate([
        { '$project': {
            '_id': 1,
```

```
            'valueAsString': { '$toString': '$value' }
        }}
    ]))

    print(result)
    client.close()

example()
```

$toUpper

The `$toUpper` operator in Amazon DocumentDB is used to convert a string to uppercase.

Parameters

- `expression`: The string expression to convert to uppercase.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$toUpper` operator to convert the `Desk` field to uppercase.

Create sample documents

```
db.locations.insertMany([
  { "_id": 1, "Desk": "düsseldorf-bvv-021" },
  { "_id": 2, "Desk": "munich-hgg-32a" }
]);
```

Query example

```
db.locations.aggregate([
  { $project: { item: { $toUpper: "$Desk" } } }
]);
```

Output

```
{ "_id" : 1, "item" : "DüSSELDORF-BVV-021" }
{ "_id" : 2, "item" : "MUNICH-HGG-32A" }
```

Code examples

To view a code example for using the `$toUpper` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('locations');

  const result = await collection.aggregate([
    { $project: { item: { $toUpper: '$Desk' } } }
  ]).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['locations']

    result = list(collection.aggregate([
        { '$project': { 'item': { '$toUpper': '$Desk' } } }
    ]))

    print(result)
```

```
    client.close()

example()
```

$toUUID

New from version 8.0.1.

The `$toUUID` operator in Amazon DocumentDB converts a string value to a UUID (Binary subtype 4).

Parameters

- `expression`: An expression that resolves to a string in UUID format (for example, "12345678-1234-1234-1234-123456789abc").

Example (MongoDB Shell)

The following example shows how to use the `$toUUID` operator to convert string values to UUID format.

Create sample documents

```
db.records.insertMany([
  {_id: 1, uuidStr: "550e8400-e29b-41d4-a716-446655440000"},
  {_id: 2, uuidStr: "6ba7b810-9dad-11d1-80b4-00c04fd430c8"}
]);
```

Query example

```
db.records.aggregate([
  { $project: { uuid: { $toUUID: "$uuidStr" } } }
]);
```

Output

```
[
  {_id: 1, uuid: UUID("550e8400-e29b-41d4-a716-446655440000")},
  {_id: 2, uuid: UUID("6ba7b810-9dad-11d1-80b4-00c04fd430c8")}
```

```
]
```

Code examples

To view a code example for using the `$toUUID` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = new MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('records');
    const result = await collection.aggregate([
      { $project: { uuid: { $toUUID: "$uuidStr" } } }
    ]).toArray();
    console.log(result);
  } finally {
    await client.close();
  }
}
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    try:
        db = client['test']
        collection = db['records']
        result = list(collection.aggregate([
            {'$project': {'uuid': {'$toUUID': '$uuidStr'}}}
        ]))
```

```
        print(result)
    finally:
        client.close()

example()
```

$trim

New from version 4.0

The `$trim` operator in Amazon DocumentDB is used to remove leading and/or trailing whitespace characters from a string.

Parameters

- `input`: The string expression to trim.
- `chars`: (optional) Specifies the characters to trim from the beginning and end of the input, the default is whitespace.

Example (MongoDB Shell)

The following example demonstrates how to use the `$trim` operator to remove leading and trailing whitespace from a string.

Create sample documents

```
db.people.insertMany([
  { "name": "   John Doe   " },
  { "name": "   Bob Johnson   " }
])
```

Query example

```
db.people.aggregate([
  { $project: {
    "name": { $trim: {input: "$name"}}
  }}
])
```

Output

```
[
  { "name": "John Doe" },
  { "name": "Bob Johnson" }
]
```

Code examples

To view a code example for using the `$trim` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    { $project: {
      "name": { $trim: {input: "$name" }}
    }}
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
```

```
    collection = db['people']

    result = list(collection.aggregate([
        {"$project": {
            "name": {"$trim": {"input": "$name"}}
        }}
    ]))

    print(result)
    client.close()

example()
```

$type

The $type aggregation operator returns the BSON data type of a specified field. This is useful for identifying the data type of field values during aggregation operations.

Parameters

- expression: The field or expression whose type to return.

Example (MongoDB Shell)

The following example demonstrates using the $type operator to identify the data type of the price field for each product.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "Notebook", price: 15.99 },
  { _id: 2, item: "Pen", price: "2.50" },
  { _id: 3, item: "Eraser", price: 1 },
  { _id: 4, item: "Ruler", price: null }
]);
```

Query example

```
db.inventory.aggregate([
  {
```

```
    $project: {
      item: 1,
      price: 1,
      priceType: { $type: "$price" }
    }
  }
]);
```

Output

```
[
  { _id: 1, item: 'Notebook', price: 15.99, priceType: 'double' },
  { _id: 2, item: 'Pen', price: '2.50', priceType: 'string' },
  { _id: 3, item: 'Eraser', price: 1, priceType: 'int' },
  { _id: 4, item: 'Ruler', price: null, priceType: 'null' }
]
```

Code examples

To view a code example for using the `$type` aggregation operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  const result = await collection.aggregate([
    {
      $project: {
        item: 1,
        price: 1,
        priceType: { $type: "$price" }
      }
    }
  ]).toArray();
```

```
  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    result = list(collection.aggregate([
        {
            '$project': {
                'item': 1,
                'price': 1,
                'priceType': { '$type': '$price' }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

$unionWith

New from version 8.0.3.

The `$unionWith` aggregation stage in Amazon DocumentDB combines the results of two aggregations into a single result set. The stage passes the documents from the previous stage of the pipeline together with the documents from a second collection to the next stage. The combined result set includes duplicates, and `$unionWith` does not guarantee the order of the documents.

Use $unionWith to combine data from multiple collections in a single query, such as merging historical and current data for reporting. This avoids writing separate queries for each collection and combining the results yourself. You can analyze data across collections without maintaining application-side merge logic.

Syntax

To combine the documents from a collection and optionally run an aggregation pipeline on them before they are added to the result set, use the following form:

```
{ $unionWith: { coll: "<collection>", pipeline: [ <stage1>, <stage2>, ... ] } }
```

To include all documents from the specified collection without any processing, use the simplified form:

```
{ $unionWith: "<collection>" }
```

Parameters

The $unionWith stage accepts the following parameters:

- coll: The name of the collection or view whose documents you want to include in the result set.
- pipeline: An optional aggregation pipeline that processes the documents in the coll collection before adding them to the result set. The pipeline cannot include the $out or $merge stages. If you omit pipeline, $unionWith includes all documents from the coll collection in the result set without any processing.

Restrictions

The $unionWith stage has the following restrictions:

- The $unionWith pipeline cannot include the $out stage.
- The $unionWith pipeline cannot include the $merge stage.
- The $unionWith stage cannot be used inside transactions.
- If the source collection that the aggregation runs on does not exist, $unionWith returns no results, even when coll specifies a collection that contains documents.

Example – Combine documents from two collections (MongoDB Shell)

The following example demonstrates how to use the `$unionWith` stage to combine documents from two collections into a single result set.

Create sample documents

```
db.sales_2020.insertMany([
  { _id: 1, item: "abc", quantity: 2 },
  { _id: 2, item: "def", quantity: 1 }
]);

db.sales_2021.insertMany([
  { _id: 1, item: "ghi", quantity: 3 },
  { _id: 2, item: "jkl", quantity: 5 }
]);
```

Query example

The following MongoDB Shell command combines the documents from the `sales_2020` collection with the documents from the `sales_2021` collection:

```
db.sales_2020.aggregate([
  { $unionWith: { coll: "sales_2021" } }
]);
```

Output

```
[
  { _id: 1, item: 'abc', quantity: 2 },
  { _id: 2, item: 'def', quantity: 1 },
  { _id: 1, item: 'ghi', quantity: 3 },
  { _id: 2, item: 'jkl', quantity: 5 }
]
```

Code examples

The following Node.js and Python code examples show how to use the `$unionWith` command. Choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();

    const db = client.db('test');
    const sales2020 = db.collection('sales_2020');

    const result = await sales2020.aggregate([
      { $unionWith: { coll: 'sales_2021' } }
    ]).toArray();

    console.log(JSON.stringify(result, null, 2));

  } catch (error) {
    console.error('Error:', error);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient
from pprint import pprint

def example():
    client = None
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
        db = client['test']
        sales_2020 = db['sales_2020']

        pipeline = [
            { '$unionWith': { 'coll': 'sales_2021' } }
        ]

        result = list(sales_2020.aggregate(pipeline))
        pprint(result)

    except Exception as e:
        print(f"An error occurred: {e}")

    finally:
        if client:
            client.close()

example()
```

$unset

New from version 8.0

Not supported by Elastic cluster.

The `$unset` aggregation stage in Amazon DocumentDB allows you to remove fields from documents.

Parameters

- `expression`: Field name or list of multiple field names.

Example (MongoDB Shell)

The following example demonstrates the use of the `$unset` aggregation stage to remove the `price` field.

Create sample documents

```
db.inventory.insertMany([
  { item: "pencil", quantity: 100, price: 0.24},
```

```
  { item: "pen", quantity: 204, price: 1.78 }
]);
```

Aggregation example

```
db.inventory.aggregate([
  {
    $unset: "price"
  }
])
```

Output

```
[
  {
    _id: ObjectId('69248951d66dcae121d2950d'),
    item: 'pencil',
    quantity: 100
  },
  {
    _id: ObjectId('69248951d66dcae121d2950e'),
    item: 'pen',
    quantity: 204
  }
]
```

Code examples

To view a code example for using the $unset command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const inventory = db.collection('inventory');
```

```
  const result = await inventory.aggregate([
      {
        $unset: "price"
      }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    inventory = db['inventory']

    result = list(inventory.aggregate([
      {
        "$unset": "price"
      }
    ]))

    print(result)
    client.close()

example()
```

$unwind

The $unwind operator is used to deconstruct an array field from the input documents to output a document for each element. This can be useful when you want to perform operations on the individual elements of an array, such as filtering, sorting, or transforming the data.

Parameters

- `path`: The path to the array field to be unwound.
- `includeArrayIndex`: (optional) Specifies the name of the new field to hold the index of the array element.
- `preserveNullAndEmptyArrays`: (optional) Determines whether the operation keeps the original document when the array field is null or an empty array.

Example (MongoDB Shell)

The following example demonstrates how to use the $unwind operator to deconstruct an array field and perform further operations on the individual elements.

Create sample documents

```
db.people.insertMany([
  { _id: 1, name: "jon", hobbies: ["painting", "dancing", "singing"] },
  { _id: 2, name: "jane", hobbies: ["reading", "swimming"] },
  { _id: 3, name: "jack", hobbies: [] }
])
```

Query example

```
db.people.aggregate([
  { $unwind: "$hobbies" }
])
```

Output

```
[
  { _id: 1, name: 'jon', hobbies: 'painting' },
  { _id: 1, name: 'jon', hobbies: 'dancing' },
  { _id: 1, name: 'jon', hobbies: 'singing' },
  { _id: 2, name: 'jane', hobbies: 'reading' },
  { _id: 2, name: 'jane', hobbies: 'swimming' }
]
```

Code examples

To view a code example for using the $unwind command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('people');

  const result = await collection.aggregate([
    { $unwind: '$hobbies' }
  ]).toArray();

  console.log(result);
  client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['people']

    result = list(collection.aggregate([
        { '$unwind': '$hobbies' }
    ]))

    print(result)
    client.close()

example()
```

$vectorSearch

New from version 8.0

Not supported by Elastic cluster.

The $vectorSearch operator in Amazon DocumentDB allows you to perform vector search, a method used in machine learning to find similar data points by comparing their vector representations using distance or similarity metrics. This capability combines the flexibility and rich querying of a JSON-based document database with the power of vector search, enabling you to build machine learning and generative AI use cases such as semantic search, product recommendation, and more.

Parameters

- <exact> (optional): Flag that specifies whether to run Exact Nearest Neighbor (ENN) or Approximate Nearest Neighbor (ANN) search. Value can be one of the following:
- false - to run ANN search
- true - to run ENN search

If omitted or set to false, numCandidates is required.

```
- `<index>` : Name of the Vector Search index to use.
- `<limit>` : Number of documents to return in the results.
- `<numCandidates>` (optional): This field is required if 'exact' is false or omitted.
 Number of nearest neighbors to use during the search. Value must be less than or equal
 to (<=) 1000. You can't specify a number less than the number of documents to return
 ('limit').
- `<path>` : Indexed vector type field to search.
- `<queryVector>` : Array of numbers that represent the query vector.
```

Example (MongoDB Shell)

The following example demonstrates how to use the $vectorSearch operator to find similar product descriptions based on their vector representations.

Create sample documents

```
db.products.insertMany([
  {
```

```
    _id: 1,
    name: "Product A",
    description: "A high-quality, eco-friendly product for your home.",
    description_vector: [ 0.2, 0.5, 0.8 ]
  },
  {
    _id: 2,
    name: "Product B",
    description: "An innovative and modern kitchen appliance.",
    description_vector: [0.7, 0.3, 0.9]
  },
  {
    _id: 3,
    name: "Product C",
    description: "A comfortable and stylish piece of furniture.",
    description_vector: [0.1, 0.2, 0.4]
  }
]);
```

Create vector search index

```
db.runCommand(
    {
        createIndexes: "products",
        indexes: [{
            key: {
                "description_vector": "vector"
            },
            vectorOptions: {
                type: "hnsw",
                dimensions: 3,
                similarity: "cosine",
                m: 16,
                efConstruction: 64
            },
            name: "description_index"
        }]
    }
);
```

Query example

```
db.products.aggregate([
```

```
  { $vectorSearch: {
      index: "description_index",
      limit: 2,
      numCandidates: 10,
      path: "description_vector",
      queryVector: [0.1, 0.2, 0.3]
    }
  }
]);
```

Output

```
[
  {
    "_id": 1,
    "name": "Product A",
    "description": "A high-quality, eco-friendly product for your home.",
    "description_vector": [ 0.2, 0.5, 0.8 ]
  },
  {
    "_id": 3,
    "name": "Product C",
    "description": "A comfortable and stylish piece of furniture.",
    "description_vector": [ 0.1, 0.2, 0.4 ]
  }
]
```

Code examples

To view a code example for using the `$vectorSearch` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findSimilarProducts(queryVector) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');
```

```
  const result = await collection.aggregate([
    { $vectorSearch: {
        index: "description_index",
        limit: 2,
        numCandidates: 10,
        path: "description_vector",
        queryVector: queryVector
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}

findSimilarProducts([0.1, 0.2, 0.3]);
```

Python

```
from pymongo import MongoClient


def find_similar_products(query_vector):
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.products

    result = list(collection.aggregate([
        {
            '$vectorSearch': {
                'index': "description_index",
                'limit': 2,
                'numCandidates': 10,
                'path': "description_vector",
                'queryVector': query_vector
            }
        }
    ]))

    print(result)
    client.close()
```

```
find_similar_products([0.1, 0.2, 0.3])
```

$week

The $week operator in Amazon DocumentDB returns the week number of a date (0-53) based on the ISO 8601 standard. The week number is calculated based on the year and the day of the week, with Monday as the first day of the week.

Parameters

None

Example (MongoDB Shell)

The following example demonstrates how to use the $week operator to retrieve the week number of a given date.

Create sample documents

```
db.events.insertMany([
  { _id: 1, date: new Date("2023-01-01") },
  { _id: 2, date: new Date("2023-01-08") },
  { _id: 3, date: new Date("2023-12-31") }
]);
```

Query example

```
db.events.aggregate([
  { $project: {
    _id: 1,
    week: { $week: "$date" }
  }}
]);
```

Output

```
[
  { "_id": 1, "week": 1 },
  { "_id": 2, "week": 2 },
  { "_id": 3, "week": 53 }
```

```
]
```

Code examples

To view a code example for using the $week command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const events = db.collection('events');

  const result = await events.aggregate([
    { $project: {
      _id: 1,
      week: { $week: "$date" }
    }}
  ]).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
  client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
  db = client['test']
  events = db['events']

  result = list(events.aggregate([
```

```
        {'$project': {
            '_id': 1,
            'week': {'$week': '$date'}
        }}
    ]))

    print(result)

    client.close()

example()
```

$year

The $year operator in Amazon DocumentDB extracts the year component from a date or timestamp.

Parameters

- expression: The date or timestamp expression from which to extract the year component.

Example (MongoDB Shell)

The following example demonstrates how to use the $year operator to extract the year component from a date field.

Create sample documents

```
db.events.insertMany([
  { "_id": 1, "date": ISODate("2023-04-15T00:00:00Z") },
  { "_id": 3, "date": ISODate("2021-12-31T00:00:00Z") }
]);
```

Query example

```
db.events.aggregate([
  { $project: { year: { $year: "$date" } } }
]);
```

Output

```
[
  { "_id": 1, "year": 2023 },
  { "_id": 3, "year": 2021 }
]
```

Code examples

To view a code example for using the `$year` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('events');

  const result = await collection.aggregate([
    { $project: { year: { $year: "$date" } } }
  ]).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['events']
```

```
    result = list(collection.aggregate([
        {'$project': {'year': {'$year': '$date'}}}
    ]))

    print(result)

    client.close()

example()
```

$zip

The `$zip` operator in Amazon DocumentDB allows you to combine multiple arrays into a single array of tuples (key-value pairs). This is useful when you need to create new documents or objects by combining data from different sources or arrays within a document.

Parameters

- `inputs`: An array of expressions that resolve to arrays. These arrays will be combined into a single array of tuples.
- `useLongestLength`: (optional) If `true`, the output array will have the length of the longest input array, padding shorter arrays with `null` values. If `false`, the output array will have the length of the shortest input array.
- `defaults`: (optional) An array of default values to use for the tuples if the corresponding input array is shorter than the longest input array and `useLongestLength` is `true`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$zip` operator to combine two arrays into a single array of tuples.

Create sample documents

```
db.grades.insert([
  {
    "_id": 1,
    "name": "John",
    "scores": [90, 85, 92],
```

```
    "classes": ["Math", "English", "Science"]
  },
  {
    "_id": 2,
    "name": "Jane",
    "scores": [88, 91, 90, 85],
    "classes": ["Math", "English", "Science", "History"]
  }
])
```

Query example

```
db.grades.aggregate([
  {
    $project: {
      "name": 1,
      "scoredClasses": {
        $zip: {
          inputs: ["$scores", "$classes"],
          useLongestLength: true,
          defaults: [null, null]
        }
      }
    }
  }
])
```

Output

```
[
  {
    "_id": 1,
    "name": "John",
    "scoredClasses": [
      [90, "Math"],
      [85, "English"],
      [92, "Science"],
      [null, null]
    ]
  },
  {
    "_id": 2,
    "name": "Jane",
```

```
    "scoredClasses": [
      [88, "Math"],
      [91, "English"],
      [90, "Science"],
      [85, "History"]
    ]
  }
]
```

Code examples

To view a code example for using the $zip command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('grades');

  const result = await collection.aggregate([
    {
      $project: {
        "name": 1,
        "scoredClasses": {
          $zip: {
            inputs: ["$scores", "$classes"],
            useLongestLength: true,
            defaults: [null, null]
          }
        }
      }
    }
  ]).toArray();

  console.log(result);
  client.close();
}
```

```
example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['grades']

    result = list(collection.aggregate([
        {
            '$project': {
                'name': 1,
                'scoredClasses': {
                    '$zip': {
                        'inputs': ['$scores', '$classes'],
                        'useLongestLength': True,
                        'defaults': [None, None]
                    }
                }
            }
        }
    ]))

    print(result)
    client.close()

example()
```

Geospatial

This section provides detailed information about geospatial operators supported by Amazon DocumentDB.

Topics

- $geometry

- $geoIntersects
- $geoWithin
- $maxDistance
- $minDistance
- $near
- $nearSphere

$geometry

The `$geometry` operator in Amazon DocumentDB is used to specify a GeoJSON geometry object as part of a geospatial query. This operator is used in conjunction with other geospatial query operators like `$geoWithin` and `$geoIntersects` to perform spatial queries on your data.

In Amazon DocumentDB, the `$geometry` operator supports the following GeoJSON geometry types:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

Parameters

- `type`: The type of the GeoJSON geometry object, e.g., `Point`, `Polygon`, etc.
- `coordinates`: An array of coordinates representing the geometry. The structure of the coordinates array depends on the geometry type.

Example (MongoDB Shell)

The following example demonstrates how to use the `$geometry` operator to perform a `$geoIntersects` query in Amazon DocumentDB.

Create sample documents

```
db.locations.insertMany([
  {
    "_id": 1,
    "name": "Location 1",
    "location": {
      "type": "Point",
      "coordinates": [-73.983253, 40.753941]
    }
  },
  {
    "_id": 2,
    "name": "Location 2",
    "location": {
      "type": "Polygon",
      "coordinates": [[
        [-73.998427, 40.730309],
        [-73.954348, 40.730309],
        [-73.954348, 40.780816],
        [-73.998427, 40.780816],
        [-73.998427, 40.730309]
      ]]
    }
  }
]);
```

Query example

```
db.locations.find({
  "location": {
    "$geoIntersects": {
      "$geometry": {
        "type": "Polygon",
        "coordinates": [[
          [-73.998, 40.730],
          [-73.954, 40.730],
          [-73.954, 40.781],
          [-73.998, 40.781],
          [-73.998, 40.730]
        ]]
      }
    }
```

```
  }
})
```

Output

```
[
  {
    "_id": 2,
    "name": "Location 2",
    "location": {
      "type": "Polygon",
      "coordinates": [
        [
          [-73.998427, 40.730309],
          [-73.954348, 40.730309],
          [-73.954348, 40.780816],
          [-73.998427, 40.780816],
          [-73.998427, 40.730309]
        ]
      ]
    }
  }
]
```

Code examples

To view a code example for using the `$geometry` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('locations');

  const query = {
    "location": {
      "$geoIntersects": {
```

```
        "$geometry": {
          "type": "Polygon",
          "coordinates": [[
            [-73.998, 40.730],
            [-73.954, 40.730],
            [-73.954, 40.781],
            [-73.998, 40.781],
            [-73.998, 40.730]
          ]]
        }
      }
    }
  };

  const result = await collection.find(query).toArray();
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['locations']

    query = {
        "location": {
            "$geoIntersects": {
                "$geometry": {
                    "type": "Polygon",
                    "coordinates": [[
                        [-73.998, 40.730],
                        [-73.954, 40.730],
                        [-73.954, 40.781],
                        [-73.998, 40.781],
```

```
                        [-73.998, 40.730]
                    ]]
                }
            }
        }
    }

    result = list(collection.find(query))
    print(result)

    client.close()

example()
```

$geoIntersects

The `$geoIntersects` operator in Amazon DocumentDB is used to find documents whose geospatial data intersects with a specified GeoJSON object. This operator is useful for applications that require identifying documents based on their spatial relationship with a given geographic shape, such as a polygon or multipolygon.

Parameters

- `$geometry`: A GeoJSON object that represents the shape to check for intersection. The supported GeoJSON object types are `Point`, `LineString`, `Polygon`, and `MultiPolygon`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$geoIntersects` operator to find the state name for a given set of coordinates in Amazon DocumentDB.

Create sample documents

```
db.states.insertMany([
  {
    "name": "New York",
    "loc": {
      "type": "Polygon",
      "coordinates": [[
        [-74.25909423828125, 40.47556838210948],
```

```
        [-73.70819091796875, 40.47556838210948],
        [-73.70819091796875, 41.31342607582222],
        [-74.25909423828125, 41.31342607582222],
        [-74.25909423828125, 40.47556838210948]
      ]]
    }
  },
  {
    "name": "California",
    "loc": {
      "type": "Polygon",
      "coordinates": [[
        [-124.4091796875, 32.56456771381587],
        [-114.5458984375, 32.56456771381587],
        [-114.5458984375, 42.00964153424558],
        [-124.4091796875, 42.00964153424558],
        [-124.4091796875, 32.56456771381587]
      ]]
    }
  }
]);
```

Query example

```
var location = [-73.965355, 40.782865];

db.states.find({
  "loc": {
    "$geoIntersects": {
      "$geometry": {
        "type": "Point",
        "coordinates": location
      }
    }
  }
}, {
  "name": 1
});
```

Output

```
{ "_id" : ObjectId("536b0a143004b15885c91a2c"), "name" : "New York" }
```

Code examples

To view a code example for using the `$geoIntersects` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findStateByGeoIntersects(longitude, latitude) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('states');

  const query = {
    loc: {
      $geoIntersects: {
        $geometry: {
          type: 'Point',
          coordinates: [longitude, latitude]
        }
      }
    }
  };

  const projection = {
    _id: 0,
    name: 1
  };

  const document = await collection.findOne(query, { projection });

  await client.close();

  if (document) {
    return document.name;
  } else {
    throw new Error('The geo location you entered was not found in the United
 States!');
  }
}
```

Python

```
from pymongo import MongoClient

def find_state_by_geointersects(longitude, latitude):
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
        db = client.test
        collection_states = db.states

        query_geointersects = {
            "loc": {
                "$geoIntersects": {
                    "$geometry": {
                        "type": "Point",
                        "coordinates": [longitude, latitude]
                    }
                }
            }
        }

        document = collection_states.find_one(query_geointersects,
                                              projection={
                                                  "_id": 0,
                                                  "name": 1
                                              })
        if document is not None:
            state_name = document['name']
            return state_name
        else:
            raise Exception("The geo location you entered was not found in the
 United States!")
    except Exception as e:
        print('Exception in geoIntersects: {}'.format(e))
        raise
    finally:
        if client is not None:
            client.close()
```

$geoWithin

The `$geoWithin` operator in Amazon DocumentDB is used to find documents whose location data (represented as GeoJSON objects) are completely contained within a specified shape, such as a polygon or multipolygon. This is useful for querying for objects that are located within a specific geographic region.

Parameters

- `$geometry`: A GeoJSON object that represents the shape to query against.

Example (MongoDB Shell)

The following example demonstrates how to use the `$geoWithin` operator to find all airports located within the state of New York.

Create sample documents

```
// Insert state document
db.states.insert({
    "name": "New York",
    "loc": {
        "type": "Polygon",
        "coordinates": [[
            [-79.76278, 45.0],
            [-73.94, 45.0],
            [-73.94, 40.5],
            [-79.76278, 40.5],
            [-79.76278, 45.0]
        ]]
    }
});

// Insert airport documents
db.airports.insert([
    {
        "name": "John F. Kennedy International Airport",
        "type": "airport",
        "code": "JFK",
        "loc": {
            "type": "Point",
            "coordinates": [-73.7781, 40.6413]
```

```
        }
    },
    {
        "name": "LaGuardia Airport",
        "type": "airport",
        "code": "LGA",
        "loc": {
            "type": "Point",
            "coordinates": [-73.8772, 40.7769]
        }
    },
    {
        "name": "Buffalo Niagara International Airport",
        "type": "airport",
        "code": "BUF",
        "loc": {
            "type": "Point",
            "coordinates": [-78.7322, 42.9403]
        }
    }
]);
```

Query example

```
var state = db.states.findOne({"name": "New York"});

db.airports.find({
    "loc": {
        "$geoWithin": {
            "$geometry": state.loc
        }
    }
}, {
    "name": 1,
    "type": 1,
    "code": 1,
    "_id": 0
});
```

Output

```
[
  {
```

```
    "name": "John F. Kennedy International Airport",
    "type": "airport",
    "code": "JFK"
  },
  {
    "name": "LaGuardia Airport",
    "type": "airport",
    "code": "LGA"
  }
]
```

Code examples

To view a code example for using the `$geoWithin` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findAirportsWithinState(stateName) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');

  const stateDoc = await db.collection('states').findOne({ name: stateName },
 { projection: { _id: 0, loc: 1 } });
  const airportDocs = await db.collection('airports').find({
    loc: {
      $geoWithin: {
        $geometry: stateDoc.loc
      }
    }
  }, { projection: { name: 1, type: 1, code: 1, _id: 0 } }).toArray();

  console.log(airportDocs);

  await client.close();
}

findAirportsWithinState('New York');
```

Python

```
from pymongo import MongoClient

def find_airports_within_state(state_name):
    try:
        client = MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
        db = client['test']
        state_doc = db.states.find_one({'name': state_name}, {'_id': 0, 'loc': 1})
        airport_docs = db.airports.find({
            'loc': {
                '$geoWithin': {
                    '$geometry': state_doc['loc']
                }
            }
        }, {'name': 1, 'type': 1, 'code': 1, '_id': 0})

        return list(airport_docs)
    except Exception as e:
        print(f'Error: {e}')
    finally:
        client.close()

airports = find_airports_within_state('New York')
print(airports)
```

$maxDistance

The `$maxDistance` operator in Amazon DocumentDB is used to specify the maximum distance (in meters) from a GeoJSON point that documents must be within to be included in the query results. This operator is used in conjunction with the `$nearSphere` operator to perform geospatial queries.

Parameters

- `$maxDistance`: The maximum distance (in meters) from the reference point that documents must be within to be included in the query results.

Example (MongoDB Shell)

The following example demonstrates how to use the `$maxDistance` operator in Amazon DocumentDB to find all state capitals within 100 kilometers of Boston.

Create sample documents

```
db.capitals.insert([
  { state: "Massachusetts", city: "Boston", location: { type: "Point", coordinates:
 [-71.0589, 42.3601] } },
  { state: "Rhode Island", city: "Providence", location: { type: "Point", coordinates:
 [-71.4128, 41.8239] } },
  { state: "New Hampshire", city: "Concord", location: { type: "Point", coordinates:
 [-71.5383, 43.2067] } },
  { state: "Vermont", city: "Montpelier", location: { type: "Point", coordinates:
 [-72.5751, 44.2604] } }
]);
```

Query example

```
db.capitals.find(
  {
    location: {
      $nearSphere: {
        $geometry: { type: "Point", coordinates: [-71.0589, 42.3601] },
        $maxDistance: 100000
      }
    }
  },
  { state: 1, city: 1, _id: 0 }
);
```

Output

```
[
  { "state": "Rhode Island", "city": "Providence" },
  { "state": "New Hampshire", "city": "Concord" }
]
```

Code examples

To view a code example for using the `$maxDistance` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findCapitalsNearBoston() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const capitals = db.collection('capitals');

  const result = await capitals.find({
    location: {
      $nearSphere: {
        $geometry: { type: "Point", coordinates: [-71.0589, 42.3601] },
        $maxDistance: 100000
      }
    }
  }, {
    projection: { state: 1, city: 1, _id: 0 }
  }).toArray();

  console.log(result);
  await client.close();
}

findCapitalsNearBoston();
```

Python

```
from pymongo import MongoClient

def find_capitals_near_boston():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    capitals = db.capitals
```

```
    result = list(capitals.find(
        {
            "location": {
                "$nearSphere": {
                    "$geometry": { "type": "Point", "coordinates": [-71.0589,
 42.3601] },
                    "$maxDistance": 100000
                }
            }
        },
        {"state": 1, "city": 1, "_id": 0}
    ))

    print(result)
    client.close()

find_capitals_near_boston()
```

$minDistance

`$minDistance` is a find operator used in conjunction with `$nearSphere` or `$geoNear` to filter documents that are at least at the specified minimum distance from the center point. This operator is supported in Amazon DocumentDB and functions similarly to its counterpart in MongoDB.

Parameters

- `$minDistance`: The minimum distance (in meters) from the center point to include documents in the results.

Example (MongoDB Shell)

In this example, we'll find all restaurants within a 2-kilometer radius of a specific location in Seattle, Washington.

Create sample documents

```
db.usarestaurants.insertMany([
  {
    "state": "Washington",
    "city": "Seattle",
```

```
    "name": "Noodle House",
    "rating": 4.8,
    "location": {
      "type": "Point",
      "coordinates": [-122.3517, 47.6159]
    }
  },
  {
    "state": "Washington",
    "city": "Seattle",
    "name": "Pike Place Grill",
    "rating": 4.5,
    "location": {
      "type": "Point",
      "coordinates": [-122.3412, 47.6102]
    }
  },
  {
    "state": "Washington",
    "city": "Bellevue",
    "name": "The Burger Joint",
    "rating": 4.2,
    "location": {
      "type": "Point",
      "coordinates": [-122.2007, 47.6105]
    }
  }
]);
```

Query example

```
db.usarestaurants.find({
  "location": {
    "$nearSphere": {
      "$geometry": {
        "type": "Point",
        "coordinates": [-122.3516, 47.6156]
      },
      "$minDistance": 1,
      "$maxDistance": 2000
    }
  }
}, {
```

```
  "name": 1
});
```

Output

```
{ "_id" : ObjectId("611f3da985009a81ad38e74b"), "name" : "Noodle House" }
{ "_id" : ObjectId("611f3da985009a81ad38e74c"), "name" : "Pike Place Grill" }
```

Code examples

To view a code example for using the `$minDistance` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findRestaurantsNearby() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('usarestaurants');

  const result = await collection.find({
    "location": {
      "$nearSphere": {
        "$geometry": {
          "type": "Point",
          "coordinates": [-122.3516, 47.6156]
        },
        "$minDistance": 1,
        "$maxDistance": 2000
      }
    }
  }, {
    "projection": { "name": 1 }
  }).toArray();

  console.log(result);
  client.close();
}
```

```
findRestaurantsNearby();
```

Python

```
from pymongo import MongoClient

def find_restaurants_nearby():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.usarestaurants

    result = list(collection.find({
        "location": {
            "$nearSphere": {
                "$geometry": {
                    "type": "Point",
                    "coordinates": [-122.3516, 47.6156]
                },
                "$minDistance": 1,
                "$maxDistance": 2000
            }
        }
    }, {
        "projection": {"name": 1}
    }))

    print(result)
    client.close()

find_restaurants_nearby()
```

$near

The `$near` operator in Amazon DocumentDB is used to find documents that are geographically near a specified point. It returns documents ordered by distance, with the closest documents first. This operator requires a 2dsphere geospatial index and is useful for proximity queries on location data.

Parameters

- `$geometry`: A GeoJSON Point object that defines the center point for the near query.
- `$maxDistance`: (optional) The maximum distance in meters from the specified point that a document can be to match the query.
- `$minDistance`: (optional) The minimum distance in meters from the specified point that a document can be to match the query.

Index Requirements

- `2dsphere index`: Required for geospatial queries on GeoJSON Point data.

Example (MongoDB Shell)

The following example demonstrates how to use the `$near` operator to find the nearest restaurants to a specific location in Seattle, Washington.

Create sample documents

```
db.usarestaurants.insert([
  {
    "name": "Noodle House",
    "city": "Seattle",
    "state": "Washington",
    "rating": 4.8,
    "location": { "type": "Point", "coordinates": [-122.3517, 47.6159] }
  },
  {
    "name": "Pike Place Grill",
    "city": "Seattle",
    "state": "Washington",
    "rating": 4.2,
    "location": { "type": "Point", "coordinates": [-122.3403, 47.6062] }
  },
  {
    "name": "Lola",
    "city": "Seattle",
    "state": "Washington",
    "rating": 4.5,
    "location": { "type": "Point", "coordinates": [-122.3407, 47.6107] }
  }
]);
```

Create 2dsphere index

```
db.usarestaurants.createIndex({ "location": "2dsphere" });
```

Query example with GeoJSON Point

```
db.usarestaurants.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [-122.3516, 47.6156]
      },
      $maxDistance: 100,
      $minDistance: 10
    }
  }
});
```

Output

```
{
  "_id" : ObjectId("69031ec9ea1c2922a1ce5f4a"),
  "name" : "Noodle House",
  "city" : "Seattle",
  "state" : "Washington",
  "rating" : 4.8,
  "location" : {
    "type" : "Point",
    "coordinates" : [ -122.3517, 47.6159 ]
  }
}
```

Code examples

To view a code example for using the `$near` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function findNearbyRestaurants() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const restaurants = db.collection('usarestaurants');

  // Create 2dsphere index
  await restaurants.createIndex({ "location": "2dsphere" });

  const result = await restaurants.find({
    location: {
      $near: {
        $geometry: {
          type: "Point",
          coordinates: [-122.3516, 47.6156]
        },
        $maxDistance: 100,
        $minDistance: 10
      }
    }
  }).toArray();

  console.log(result);

  client.close();
}

findNearbyRestaurants();
```

Python

```
from pymongo import MongoClient

def find_nearby_restaurants():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    restaurants = db['usarestaurants']

    # Create 2dsphere index
```

```
    restaurants.create_index([("location", "2dsphere")])

    result = list(restaurants.find({
        'location': {
            '$near': {
                '$geometry': {
                    'type': 'Point',
                    'coordinates': [-122.3516, 47.6156]
                },
                '$maxDistance': 100,
                '$minDistance': 10
            }
        }
    }))

    print(result)

    client.close()

find_nearby_restaurants()
```

$nearSphere

The `$nearSphere` operator in Amazon DocumentDB is used to find documents that are within a specified distance of a geospatial point. This operator is particularly useful for geo-spatial queries, such as finding all restaurants within a certain radius of a given location.

Parameters

- `$geometry`: A GeoJSON object that represents the reference point. Must be a `Point` object with `type` and `coordinates` fields.
- `$minDistance`: (optional) The minimum distance (in meters) from the reference point that documents must be.
- `$maxDistance`: (optional) The maximum distance (in meters) from the reference point that documents must be.

Example (MongoDB Shell)

In this example, we will find all restaurants within 2 kilometers (2000 meters) of a specific location in Seattle, WA.

Create sample documents

```
db.usarestaurants.insert([
  {
    name: "Noodle House",
    location: { type: "Point", coordinates: [-122.3516, 47.6156] }
  },
  {
    name: "Pike Place Grill",
    location: { type: "Point", coordinates: [-122.3403, 47.6101] }
  },
  {
    name: "Seattle Coffee Co.",
    location: { type: "Point", coordinates: [-122.3339, 47.6062] }
  }
]);
```

Query example

```
db.usarestaurants.find({
  location: {
    $nearSphere: {
      $geometry: {
        type: "Point",
        coordinates: [-122.3516, 47.6156]
      },
      $minDistance: 1,
      $maxDistance: 2000
    }
  }
}, {
  name: 1
});
```

Output

```
{ "_id" : ObjectId("611f3da985009a81ad38e74b"), "name" : "Noodle House" }
{ "_id" : ObjectId("611f3da985009a81ad38e74c"), "name" : "Pike Place Grill" }
```

Code examples

To view a code example for using the `$nearSphere` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findNearbyRestaurants() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const restaurants = db.collection('usarestaurants');

  const result = await restaurants.find({
    location: {
      $nearSphere: {
        $geometry: {
          type: "Point",
          coordinates: [-122.3516, 47.6156]
        },
        $minDistance: 1,
        $maxDistance: 2000
      }
    }
  }, {
    projection: { name: 1 }
  }).toArray();

  console.log(result);
  client.close();
}

findNearbyRestaurants();
```

Python

```
from pymongo import MongoClient

def find_nearby_restaurants():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    restaurants = db.usarestaurants

    result = list(restaurants.find({
        'location': {
            '$nearSphere': {
                '$geometry': {
                    'type': 'Point',
                    'coordinates': [-122.3516, 47.6156]
                },
                '$minDistance': 1,
                '$maxDistance': 2000
            }
        }
    }, {
        'name': 1
    }))

    print(result)
    client.close()

find_nearby_restaurants()
```

Query and projection operators

This section provides detailed information about query and projection operators supported by Amazon DocumentDB.

Topics

- $
- $all
- $and
- $bitsAllClear
- $bitsAllSet
- $bitsAnyClear
- $bitsAnySet

- $comment
- $elemMatch
- $eq
- $exists
- $expr
- $gt
- $gte
- $in
- $jsonSchema
- $lt
- $mod
- $lte
- $meta
- $ne
- $nin
- $nor
- $not
- $or
- $regex
- $sampleRate
- $slice
- $size
- $text
- $type

$

The $ projection operator limits the contents of an array field to return only the first element that matches the query condition. It is used to project a single matching array element.

Parameters

- `field.$`: The array field with the positional operator to project the first matching element.

Example (MongoDB Shell)

The following example demonstrates using the $ projection operator to return only the matching array element.

Create sample documents

```
db.students.insertMany([
  { _id: 1, name: "Alice", grades: [85, 92, 78, 95] },
  { _id: 2, name: "Bob", grades: [70, 88, 92, 65] },
  { _id: 3, name: "Charlie", grades: [95, 89, 91, 88] }
]);
```

Query example

```
db.students.find(
  { grades: { $gte: 90 } },
  { name: 1, "grades.$": 1 }
);
```

Output

```
{ "_id" : 1, "name" : "Alice", "grades" : [ 92 ] }
{ "_id" : 2, "name" : "Bob", "grades" : [ 92 ] }
{ "_id" : 3, "name" : "Charlie", "grades" : [ 95 ] }
```

In this example, only the first grade that is greater than or equal to 90 is returned for each student.

Code examples

To view a code example for using the $ projection operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('students');

  const result = await collection.find(
    { grades: { $gte: 90 } },
    { projection: { name: 1, "grades.$": 1 } }
  ).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['students']

    result = list(collection.find(
        {'grades': {'$gte': 90}},
        {'name': 1, 'grades.$': 1}
    ))

    print(result)
    client.close()

example()
```

$all

The `$all` operator in Amazon DocumentDB is used to match documents where the value of a field is an array and contains all the specified elements, regardless of the order of the elements in the array.

Parameters

- `field`: The name of the field to check.
- `[value1, value2, ...]`: The list of values to match in the array.

Using `$elemMatch` within an `$all` expression

See Using `$elemMatch` within an `$all` expression for limitations regarding the use of the `$elemMatch` operator within an `$all` expression.

Dollar ($) in field names

See Dollar($) and dot(.) in field names for limitations regarding querying $ prefixed fields in `$all` in nested objects.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$all` operator to retrieve documents where the "Colors" field is an array that contains both "Red" and "Blue".

Create sample documents

```
db.example.insertMany([
  { "Item": "Pen", "Colors": ["Red", "Blue", "Green"] },
  { "Item": "Notebook", "Colors": ["Blue", "White"] },
  { "Item": "Poster Paint", "Colors": ["Red", "Yellow", "White"] }
])
```

Query example

```
db.example.find({ "Colors": { $all: ["Red", "Blue"] } }).pretty()
```

Output

```
{
  "_id" : ObjectId("6137d6c5b3a1d35e0b6ee6ad"),
  "Item" : "Pen",
  "Colors" : [
          "Red",
```

```
          "Blue",
          "Green"
  ]
}
```

Code examples

To view a code example for using the `$all` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('example');

  const result = await collection.find({ "Colors": { $all: ["Red",
 "Blue"] } }).toArray();
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['example']

    result = list(collection.find({ "Colors": { "$all": ["Red", "Blue"] } }))
    print(result)
```

```
    client.close()

example()
```

$and

The $and operator in Amazon DocumentDB is used to combine multiple expressions and evaluate them as a single condition. It returns `true` if all the provided expressions evaluate to `true`, and `false` otherwise. This operator is useful for applying multiple criteria to a query.

Parameters

- `expression1`: A required expression that evaluates to a boolean value.
- `expression2`: A required expression that evaluates to a boolean value.
- `...`: Additional required expressions that evaluate to boolean values.

Example (MongoDB Shell)

The following example demonstrates the use of the $and operator to find all documents in the "users" collection where the "age" field is greater than 18 and the "status" field is "active".

Create sample documents

```
db.users.insertMany([
  { name: "John", age: 25, status: "active" },
  { name: "Jane", age: 17, status: "active" },
  { name: "Bob", age: 30, status: "inactive" },
  { name: "Alice", age: 22, status: "active" }
]);
```

Query example

```
db.users.find({
  $and: [
    { age: { $gt: 18 } },
    { status: "active" }
  ]
});
```

Output

```
[
  { "_id" : ObjectId("614e3c4b63f5892e7c4e2345"), "name" : "John", "age" : 25,
 "status" : "active" },
  { "_id" : ObjectId("614e3c4b63f5892e7c4e2347"), "name" : "Alice", "age" : 22,
 "status" : "active" }
]
```

Code examples

To view a code example for using the $and command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findActiveUsersOlderThan18() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const activeUsersOlderThan18 = await users.find({
    $and: [
      { age: { $gt: 18 } },
      { status: 'active' }
    ]
  }).toArray();

  console.log(activeUsersOlderThan18);
  await client.close();
}

findActiveUsersOlderThan18();
```

Python

```
from pymongo import MongoClient

def find_active_users_older_than_18():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    users = db['users']

    active_users_older_than_18 = list(users.find({
        '$and': [
            {'age': {'$gt': 18}},
            {'status': 'active'}
        ]
    }))

    print(active_users_older_than_18)
    client.close()

find_active_users_older_than_18()
```

$bitsAllClear

The `$bitsAllClear` operator in Amazon DocumentDB is used to match the documents where all the specified bits in a field are cleared (set to 0). This operator can be useful for performing bitwise operations on stored data.

Parameters

- `field`: The field to check for the specified bits being cleared.
- `value`: The numeric bitmask that specifies which bits should be checked, or a list of bits positions to be checked. A numeric bitmask can be a binary (0b...), decimal, hexadecimal (0x...), octal (0o...), or binary (BinData) form. In a list of bits positions, the position of the least significant bit is 0.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$bitsAllClear` operator in Amazon DocumentDB.

Create sample documents

```
db.collection.insertMany([
```

```
  { _id: 1, bits: 0b1010 },
  { _id: 2, bits: 0b1100 },
  { _id: 3, bits: 0b0101 }
]);
```

Query example

```
db.collection.find({
  bits: { $bitsAllClear: 0b0011 }
})
```

Output

```
{ "_id" : 2, "bits" : 12 }
```

The query checks if all the bits specified by the bitmask `0b0011` (the two least significant bits) are cleared in the `bits` field. The document with `_id` 2 satisfies this condition, as its `bits` field has those bits cleared.

Code examples

To view a code example for using the `$bitsAllClear` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.find({
    bits: { $bitsAllClear: 0b0011 }
  }).toArray();

  console.log(result);
```

```
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['collection']

    result = list(collection.find({
        'bits': { '$bitsAllClear': 0b0011 }
    }))

    print(result)

    client.close()

example()
```

$bitsAllSet

The `$bitsAllSet` operator in Amazon DocumentDB is used to query for documents where a specific set of bits are all set to 1 in a field. This operator allows you to perform bitwise operations on field values and can be useful when you need to check the state of individual bits within a numeric value.

Parameters

- `field`: The name of the numeric field to perform the bitwise operation on.
- `value`: The numeric bitmask that specifies which bits should be checked, or a list of bits positions to be checked. A numeric bitmask can be a binary (0b...), decimal, hexadecimal (0x...), octal (0o...), or binary (BinData) form. In a list of bits positions, the position of the least significant bit is 0.

Example (MongoDB Shell)

The following example demonstrates how to use the `$bitsAllSet` operator to find documents where the `flags` field has all the bits set that are specified by the bitmask.

Create sample documents

```
db.collection.insert([
  { _id: 1, flags: 0b1010 },
  { _id: 2, flags: 0b1100 },
  { _id: 3, flags: 0b1110 }
])
```

Query example

```
db.collection.find({ flags: { $bitsAllSet: 0b1100 } })
```

Output

```
{ "_id": 2, "flags": 12 },
{ "_id": 3, "flags": 14 }
```

In this example, the query checks for documents where the `flags` field has all the bits set that are specified by the bitmask `0b1100` (which represents the decimal value 12). The documents with `_id` 2 and 3 match this criteria, as their `flags` field values have all the required bits set (the 3rd and 4th least significant bits).

Code examples

To view a code example for using the `$bitsAllSet` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findWithBitsAllSet() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('collection');
```

```
  const result = await collection.find({ flags: { $bitsAllSet:
 0b1100 } }).toArray();
  console.log(result);

  await client.close();
}

findWithBitsAllSet();
```

Python

```
from pymongo import MongoClient

def find_with_bits_all_set():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.collection

    result = list(collection.find({ 'flags': { '$bitsAllSet': 0b1100 } }))
    print(result)

    client.close()

find_with_bits_all_set()
```

$bitsAnyClear

The `$bitsAnyClear` operator in Amazon DocumentDB is used to match the documents where any of the specified bit in a field are cleared (set to 0). This can be useful for performing bitwise operations on field values stored in documents.

Parameters

- `field`: The field to check.
- `value`: The numeric bitmask that specifies which bits should be checked, or a list of bits positions to be checked. A numeric bitmask can be a binary (0b...), decimal, hexadecimal (0x...), octal (0o...), or binary (BinData) form. In a list of bits positions, the position of the least significant bit is 0.

Example (MongoDB Shell)

The following example demonstrates how to use the `$bitsAnyClear` operator to check if any bit is clear in the `status` field of the `items` collection.

Create sample documents

```
db.items.insertMany([
  { "_id": 1, "status": 7 },
  { "_id": 2, "status": 15 },
  { "_id": 3, "status": 31 }
]);
```

Query example

```
db.items.find({ "status": { $bitsAnyClear: 8 } })
```

Output

```
{ "_id" : 1, "status" : 7 }
```

In this example, the query checks for documents where the `status` field has any bits clear (0) in the bitmask 8 (binary 1000). The document with `status` values of 7 (binary 111) matches the query, as it has at least one bit clear in the provided bitmask. The matching clear bit is the 4th least significant bit.

Code examples

To view a code example for using the `$bitsAnyClear` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('items');
```

```
  const result = await collection.find({ "status": { $bitsAnyClear:
 8 } }).toArray();
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.items

    result = list(collection.find({ "status": { "$bitsAnyClear": 8 } }))
    print(result)

    client.close()

example()
```

$bitsAnySet

The `$bitsAnySet` operator in Amazon DocumentDB is used to query documents where at least one bit is set to 1 in the specified bits in a field. This operator allows you to perform bitwise operations on the values stored in fields, enabling efficient querying and analysis of data with bitwise characteristics.

Parameters

- `field`: The field name to apply the bitwise operation to.
- `value`: The numeric bitmask that specifies which bits should be checked, or a list of bits positions to be checked. A numeric bitmask can be a binary (0b...), decimal, hexadecimal (0x...), octal (0o...), or binary (BinData) form. In a list of bits positions, the position of the least significant bit is 0.

Example (MongoDB Shell)

The following example demonstrates how to use the `$bitsAnySet` operator to find documents where at least one bit is set in the `flags` field.

Create sample documents

```
db.collection.insertMany([
  { _id: 1, flags: 0b1010 },
  { _id: 2, flags: 0b1100 },
  { _id: 3, flags: 0b0011 },
  { _id: 4, flags: 0b0100 }
]);
```

Query example

```
db.collection.find({
  flags: { $bitsAnySet: 0b1010 }
})
```

Output

```
{ "_id" : 1, "flags" : 10 }
{ "_id" : 2, "flags" : 12 }
{ "_id" : 3, "flags" : 3 }
```

The query returns the documents where at least one of the bits specified in the bitmask `0b1010` is set in the `flags` field.

Code examples

To view a code example for using the `$bitsAnySet` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
```

```
  const db = client.db('test');
  const collection = db.collection('collection');

  const result = await collection.find({
    flags: { $bitsAnySet: 0b1010 }
  }).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['collection']

result = list(collection.find({
    'flags': { '$bitsAnySet': 0b1010 }
}))

print(result)

client.close()
```

$comment

The `$comment` operator in Amazon DocumentDB is used to attach a comment to a query. This can be useful for providing additional context or information about the query, which can be helpful for debugging or documenting purposes. The attached comment will appear as part of the output of operations like db.currentOp().

Parameters

- `string`: The comment attached to the query.

Example (MongoDB Shell)

The following example demonstrates how to use the $comment operator in Amazon DocumentDB.

Create sample documents

```
db.users.insertMany([
  { name: "John Doe", age: 30, email: "john.doe@example.com" },
  { name: "Jane Smith", age: 25, email: "jane.smith@example.com" },
  { name: "Bob Johnson", age: 35, email: "bob.johnson@example.com" }
]);
```

Query example

```
db.users.find({ age: { $gt: 25 } }, { _id: 0, name: 1, age: 1 }).comment("Retrieve
 users older than 25");
```

Output

```
{ "name" : "John Doe", "age" : 30 }
{ "name" : "Bob Johnson", "age" : 35 }
```

Code examples

To view a code example for using the $comment command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const result = await users.find({ age: { $gt: 25 } }, { projection: { _id: 0,
 name: 1, age: 1 } })
                            .comment('Retrieve users older than 25')
                            .toArray();
```

```
  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    users = db.users

    result = list(users.find({ 'age': { '$gt': 25 }}, { '_id': 0, 'name': 1, 'age':
 1 })
                .comment('Retrieve users older than 25'))

    print(result)

    client.close()

if __name__ == '__main__':
    main()
```

$elemMatch

The `$elemMatch` operator in Amazon DocumentDB is used to query an array field and return documents where at least one element in the array matches the specified criteria. This operator is particularly useful when you have complex data structures with nested arrays or embedded documents.

Planner version 2.0 added index support for `$elemMatch`.

Parameters

- `field`: The array field to query.

- `query`: The criteria to match against the array elements.

Using `$elemMatch` within an `$all` expression

See Using `$elemMatch` within an `$all` expression for limitations regarding the use of the `$elemMatch` operator within an `$all` expression.

Example (MongoDB Shell)

The following example demonstrates how to use the `$elemMatch` operator to find documents where the `parts` array has at least one element that matches the specified criteria.

Create sample documents

```
db.col.insertMany([
  { _id: 1, parts: [{ part: "xyz", qty: 10 }, { part: "abc", qty: 20 }] },
  { _id: 2, parts: [{ part: "xyz", qty: 5 }, { part: "abc", qty: 10 }] },
  { _id: 3, parts: [{ part: "xyz", qty: 15 }, { part: "abc", qty: 100 }] },
  { _id: 4, parts: [{ part: "abc", qty: 150 }] }
]);
```

Query example

```
db.col.find({
  parts: { "$elemMatch": { part: "xyz", qty: { $lt: 11 } } }
})
```

Output

```
{ "_id" : 1, "parts" : [ { "part" : "xyz", "qty" : 10 }, { "part" : "abc", "qty" :
 20 } ] }
{ "_id" : 2, "parts" : [ { "part" : "xyz", "qty" : 5 }, { "part" : "abc", "qty" :
 10 } ] }
```

Code examples

To view a code example for using the `$elemMatch` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const col = db.collection('col');

  const result = await col.find({
    parts: {
      "$elemMatch": { part: "xyz", qty: { $lt: 11 } }
    }
  }).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    col = db['col']

    result = list(col.find({
      'parts': {
        '$elemMatch': {'part': 'xyz', 'qty': {'$lt': 11}}
      }
    }))

    print(result)
    client.close()
```

```
example()
```

$eq

The $eq operator in Amazon DocumentDB is used to match documents where the value of a field is equal to the specified value. This operator is commonly used in the `find()` method to retrieve documents that meet the specified criteria.

Parameters

- `field` : The field to check for the equality condition.
- `value` : The value to compare against the field.

Example (MongoDB Shell)

The following example demonstrates the usage of the $eq operator to find all documents where the name field is equal to `"Thai Curry Palace"`.

Create sample documents

```
db.restaurants.insertMany([
  { name: "Thai Curry Palace", cuisine: "Thai", features: ["Private Dining"] },
  { name: "Italian Bistro", cuisine: "Italian", features: ["Outdoor Seating"] },
  { name: "Mexican Grill", cuisine: "Mexican", features: ["Takeout"] }
]);
```

Query example

```
db.restaurants.find({ name: { $eq: "Thai Curry Palace" } });
```

Output

```
{ "_id" : ObjectId("68ee586f916df9d39f3d9414"), "name" : "Thai Curry Palace",
 "cuisine" : "Thai", "features" : [ "Private Dining" ] }
```

Code examples

To view a code example for using the $eq command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findByName(name) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('restaurants');

  const results = await collection.find({ name: { $eq: name } }).toArray();
  console.log(results);

  await client.close();
}

findByName("Thai Curry Palace");
```

Python

```
from pymongo import MongoClient

def find_by_name(name):
    client = MongoClient("mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false")
    db = client["test"]
    collection = db["restaurants"]

    results = list(collection.find({ "name": { "$eq": name } }))
    print(results)

    client.close()

find_by_name("Thai Curry Palace")
```

$exists

The `$exists` operator is used to check whether a field exists in a document or not. `$exists` particularly useful when working with sparse indexes in Amazon DocumentDB, where the indexed field may not be present in all documents.

To use a sparse index that you have created in a query, you must use the `$exists` clause on the fields that cover the index. If you omit `$exists`, Amazon DocumentDB will not use the sparse index for the query.

Parameters

- `field`: The field name to check for existence.
- `value`: A boolean value (`true` or `false`) that specifies whether the field should exist (`true`) or not exist (`false`) in the matching documents.

Example (MongoDB Shell)

The following example demonstrates the use of the `$exists` operator with a sparse index on the `special_diets` field in the `food` collection.

Create sample documents

```
db.food.insertMany([
  { _id: 1, name: "Apple", special_diets: ["vegetarian", "gluten-free"] },
  { _id: 2, name: "Broccoli" },
  { _id: 3, name: "Chicken", special_diets: ["dairy-free"] }
]);
```

Create a sparse index on the `special_diets` field

```
db.food.createIndex({ "special_diets": 1 }, { sparse: true, name:
 "special_diets_sparse_asc" });
```

Query example

```
db.food.find({ "special_diets": { $exists: true } });
```

Output

```
[
  { "_id" : 1, "name" : "Apple", "special_diets" : [ "vegetarian", "gluten-free" ] },
  { "_id" : 3, "name" : "Chicken", "special_diets" : [ "dairy-free" ] }
]
```

Code examples

To view a code example for using the `$exists` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('food');

  const result = await collection.find({ "special_diets": { $exists:
 true } }).toArray();
  console.log(result);

  await client.close();
}

main();
```

Python

```
import pymongo

client = pymongo.MongoClient('mongodb://<username>:<password>@<cluster-
endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['food']

result = list(collection.find({"special_diets": {"$exists": True}}))
print(result)
```

```
client.close()
```

$expr

New from version 4.0.

Not supported by Elastic cluster.

The `$expr` operator in Amazon DocumentDB allows you to use aggregation expressions within the query language. It enables you to perform complex comparisons and computations on fields within a document, similar to the way you would use aggregation pipeline stages.

The `$eq`, `$lt`, `$lte`, `$gt`, and `$gte` comparison operators placed in an `$expr` operator can use an index, including an index on the `from` collection referenced in a `$lookup` stage.

Limitations

- Indexes can only be used for comparisons between fields and constants, so the operand must resolve to a constant. For example, a comparison between `$a` and a constant value can use an index, but a comparison between `$a` and `$b` cannot.
- Multikey, partial, or sparse indexes are not used. In those cases the query falls back to a collection scan.
- When the `let` operand resolves to an empty or missing value, the comparison cannot use a targeted index seek. In this case, Amazon DocumentDB performs a full index scan.

Parameters

- `expression`: An expression that returns a boolean value, allowing you to perform comparisons and computations on document fields.

Example (MongoDB Shell)

The following example demonstrates how to use the `$expr` operator to find all documents where the `manufacturingCost` field is greater than the `price` field.

Create sample documents

```
db.inventory.insertMany([
  { item: "abc", manufacturingCost: 500, price: 100 },
  { item: "def", manufacturingCost: 300, price: 450 },
  { item: "ghi", manufacturingCost: 400, price: 120 }
]);
```

Query example

```
db.inventory.find({
  $expr: {
    $gt: ["$manufacturingCost", "$price"]
  }
})
```

Output

```
{ "_id" : ObjectId("60b9d4d68d2cac581bc5a89a"), "item" : "abc", "manufacturingCost" :
 500, "price" : 100 },
{ "_id" : ObjectId("60b9d4d68d2cac581bc5a89c"), "item" : "ghi", "manufacturingCost" :
 400, "price" : 120 }
```

Code examples

To view a code example for using the `$expr` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  const result = await collection.find({
    $expr: {
      $gt: ['$manufacturingCost', '$price']
    }
```

```
  }).toArray();

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    result = list(collection.find({
        '$expr': {
            '$gt': ['$manufacturingCost', '$price']
        }
    }))

    print(result)
    client.close()

example()
```

$gt

The `$gt` operator in Amazon DocumentDB is used to select documents where the value of the specified field is greater than the specified value. This operator is useful for filtering and querying data based on numerical comparisons.

Parameters

- `field`: The field to compare.
- `value`: The value to compare against.

Example (MongoDB Shell)

The following example demonstrates how to use the $gt operator to find all documents where the age field is greater than 30.

Create sample documents

```
db.users.insertMany([
  { name: "John", age: 25 },
  { name: "Jane", age: 32 },
  { name: "Bob", age: 45 },
  { name: "Alice", age: 28 }
]);
```

Query example

```
db.users.find({ age: { $gt: 30 } });
```

Output

```
{ "_id" : ObjectId("6249e5c22a5d39884a0a0001"), "name" : "Jane", "age" : 32 },
{ "_id" : ObjectId("6249e5c22a5d39884a0a0002"), "name" : "Bob", "age" : 45 }
```

Code examples

To view a code example for using the $gt command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findUsersOlderThan30() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = await db.collection('users').find({ age: { $gt: 30 } }).toArray();
  console.log(users);
  await client.close();
}
```

```
findUsersOlderThan30();
```

Python

```
from pymongo import MongoClient

def find_users_older_than_30():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    users = list(db.users.find({ 'age': { '$gt': 30 } }))
    print(users)
    client.close()

find_users_older_than_30()
```

$gte

The `$gte` operator in Amazon DocumentDB is used to match values that are greater than or equal to a specified value. This operator is useful for filtering and querying data based on numerical comparisons.

Parameters

- `field`: The field to check against the provided value.
- `value`: The value to compare against the field.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$gte` operator in Amazon DocumentDB to find all documents where the "age" field is greater than or equal to 25.

Create sample documents

```
db.users.insertMany([
  { _id: 1, name: "John", age: 20 },
  { _id: 2, name: "Jane", age: 25 },
  { _id: 3, name: "Bob", age: 30 },
```

```
  { _id: 4, name: "Alice", age: 35 }
]);
```

Query example

```
db.users.find({ age: { $gte: 25 } }, { _id: 0, name: 1, age: 1 });
```

Output

```
{ "name" : "Jane", "age" : 25 }
{ "name" : "Bob", "age" : 30 }
{ "name" : "Alice", "age" : 35 }
```

Code examples

To view a code example for using the `$gte` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findUsersAboveAge(age) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  const result = await users.find({ age: { $gte: age } }, { projection: { _id: 0,
 name: 1, age: 1 } }).toArray();
  console.log(result);

  await client.close();
}

findUsersAboveAge(25);
```

Python

```
from pymongo import MongoClient
```

```
def find_users_above_age(age):
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    users = db.users

    result = list(users.find({ 'age': { '$gte': age } }, { '_id': 0, 'name': 1,
 'age': 1 }))
    print(result)

    client.close()

find_users_above_age(25)
```

$in

The `$in` operator in Amazon DocumentDB is a logical query operator that allows you to find documents where the value of a field equals any of the values specified in an array.

Parameters

- `field`: The field to check against the provided array.
- `[value1, value2, ...]`: An array of values to match against the specified field.

Dollar ($) in field names

See Dollar($) and dot(.) in field names for limitations regarding querying $ prefixed fields in `$in` in nested objects.

Example (MongoDB Shell)

The following example demonstrates how to use the `$in` operator to find documents where the `color` field is one of the values in the provided array.

Create sample documents

```
db.colors.insertMany([
```

```
  { "_id": 1, "color": "red" },
  { "_id": 2, "color": "green" },
  { "_id": 3, "color": "blue" },
  { "_id": 4, "color": "yellow" },
  { "_id": 5, "color": "purple" }
])
```

Query example

```
db.colors.find({ "color": { "$in": ["red", "blue", "purple"] } })
```

Output

```
{ "_id": 1, "color": "red" },
{ "_id": 3, "color": "blue" },
{ "_id": 5, "color": "purple" }
```

Code examples

To view a code example for using the `$in` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findByIn() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('colors');

  const result = await collection.find({ "color": { "$in": ["red", "blue",
 "purple"] } }).toArray();
  console.log(result);

  await client.close();
}

findByIn();
```

Python

```
from pymongo import MongoClient

def find_by_in():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.colors

    result = list(collection.find({ "color": { "$in": ["red", "blue",
 "purple"] } }))
    print(result)

    client.close()

find_by_in()
```

$jsonSchema

New from version 4.0.

Not supported by Elastic cluster.

The `$jsonSchema` operator in Amazon DocumentDB is used to filter documents based on a specified JSON schema. This operator allows you to query documents that match a particular JSON schema, ensuring that the retrieved documents adhere to specific structural and data type requirements.

Using the `$jsonSchema` evaluation query operator as part of a collection creation, you can validate the schema of the documents being inserted into the collection. See Using JSON schema validation for additional information.

Parameters

- `required` (array): Specifies the required fields in the document.
- `properties` (object): Defines the data type and other constraints for each field in the document.

Example (MongoDB Shell)

The following example demonstrates the use of the `$jsonSchema` operator to filter the `employees` collection to only retrieve documents that have the `name`, `employeeId` and `age` fields, and the `employeeId` field is of type `string`.

Create sample documents

```
db.employees.insertMany([
  { "name": { "firstName": "Carol", "lastName": "Smith" }, "employeeId": "1" },
  { "name": { "firstName": "Emily", "lastName": "Brown" }, "employeeId": "2", "age":
 25 },
  { "name": { "firstName": "William", "lastName": "Taylor" }, "employeeId": 3, "age":
 24 },
  { "name": { "firstName": "Jane", "lastName": "Doe" }, "employeeId": "4" }
]);
```

Query example

```
db.employees.aggregate([
  { $match: {
    $jsonSchema: {
      required: ["name", "employeeId", "age"],
      properties: { "employeeId": { "bsonType": "string" } }
    }
  }}
]);
```

Output

```
{ "_id" : ObjectId("6908e8b61f77fc26b2ecd26f"), "name" : { "firstName" : "Emily",
 "lastName" : "Brown" }, "employeeId" : "2", "age" : 25 }
```

Code examples

To view a code example for using the `$jsonSchema` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function filterByJsonSchema() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('employees');

  const result = await collection.aggregate([
    {
      $match: {
        $jsonSchema: {
          required: ['name', 'employeeId', 'age'],
          properties: { 'employeeId': { 'bsonType': 'string' } }
        }
      }
    }
  ]).toArray();

  console.log(result);
  await client.close();
}

filterByJsonSchema();
```

Python

```
from pymongo import MongoClient

def filter_by_json_schema():
  client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
  db = client['test']
  collection = db['employees']

  result = list(collection.aggregate([
    {
      '$match': {
        '$jsonSchema': {
          'required': ['name', 'employeeId', 'age'],
          'properties': {'employeeId': {'bsonType': 'string'}}
        }
```

```
      }
    }
  ]))

  print(result)
  client.close()

filter_by_json_schema()
```

$lt

The `$lt` operator in Amazon DocumentDB is used to match values that are less than the specified value. This operator is useful for filtering and querying data based on numerical comparisons.

Parameters

- `field`: The field name to check.
- `value`: The value to compare against.

Example (MongoDB Shell)

The following example demonstrates how to use the `$lt` operator to retrieve documents where the `Inventory.OnHand` value is less than 50.

Create sample documents

```
db.example.insertMany([
  { "_id": 1, "Inventory": { "OnHand": 25, "Sold": 60 }, "Colors": ["Red", "Blue"] },
  { "_id": 2, "Inventory": { "OnHand": 75, "Sold": 40 }, "Colors": ["Red", "White"] },
  { "_id": 3, "Inventory": { "OnHand": 10, "Sold": 85 }, "Colors": ["Red", "Green"] }
]);
```

Query example

```
db.example.find({ "Inventory.OnHand": { $lt: 50 } })
```

Output

```
{ "_id" : 1, "Inventory" : { "OnHand" : 25, "Sold" : 60 }, "Colors" : [ "Red",
 "Blue" ] }
```

```
{ "_id" : 3, "Inventory" : { "OnHand" : 10, "Sold" : 85 }, "Colors" : [ "Red",
 "Green" ] }
```

Code examples

To view a code example for using the `$lt` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('example');

  const result = await collection.find({ "Inventory.OnHand": { $lt:
 50 } }).toArray();
  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['example']

    result = list(collection.find({ "Inventory.OnHand": { "$lt": 50 } }))
    print(result)

    client.close()
```

```
example()
```

$mod

The $mod query operator selects documents where a field value divided by a divisor has a specified remainder. This is useful for filtering documents based on modulo arithmetic conditions.

Parameters

- `divisor`: The number to divide by.
- `remainder`: The expected remainder value.

Example (MongoDB Shell)

The following example demonstrates using the $mod operator to find all orders where the quantity is an odd number.

Create sample documents

```
db.orders.insertMany([
  { _id: 1, item: "Widget", quantity: 15 },
  { _id: 2, item: "Gadget", quantity: 20 },
  { _id: 3, item: "Tool", quantity: 7 },
  { _id: 4, item: "Device", quantity: 12 },
  { _id: 5, item: "Part", quantity: 9 }
]);
```

Query example

```
db.orders.find({ quantity: { $mod: [2, 1] } });
```

Output

```
{ "_id" : 1, "item" : "Widget", "quantity" : 15 }
{ "_id" : 3, "item" : "Tool", "quantity" : 7 }
{ "_id" : 5, "item" : "Part", "quantity" : 9 }
```

This query returns documents where the quantity divided by 2 has a remainder of 1, effectively selecting all odd quantities.

Code examples

To view a code example for using the $mod query operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('orders');

  const result = await collection.find({ quantity: { $mod: [2, 1] } }).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['orders']

    result = list(collection.find({'quantity': {'$mod': [2, 1]}}))

    print(result)
    client.close()

example()
```

$lte

The `$lte` operator in Amazon DocumentDB is used to match documents where the value of a specified field is less than or equal to the specified value. This operator is useful for filtering and querying data based on numerical comparisons.

Parameters

- `field`: The field to compare.
- `value`: The value to compare against.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$lte` operator to retrieve documents where the `quantity` field is less than or equal to 10.

Create sample documents

```
db.inventory.insertMany([
  { item: "canvas", qty: 100 },
  { item: "paint", qty: 50 },
  { item: "brush", qty: 10 },
  { item: "paper", qty: 5 }
]);
```

Query example

```
db.inventory.find({ qty: { $lte: 10 } });
```

Output

```
{ "_id" : ObjectId("..."), "item" : "brush", "qty" : 10 },
{ "_id" : ObjectId("..."), "item" : "paper", "qty" : 5 }
```

Code examples

To view a code example for using the `$lte` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require("mongodb");

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db("test");
  const collection = db.collection("inventory");

  const result = await collection.find({ qty: { $lte: 10 } }).toArray();
  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client["test"]
    collection = db["inventory"]

    result = list(collection.find({ "qty": { "$lte": 10 } }))
    print(result)

    client.close()

if __name__ == "__main__":
    main()
```

$meta

The $meta operator is used to access metadata associated with the current query execution. This operator is primarily used for text search operations, where the metadata can provide information about the relevance of the matched documents.

Parameters

- textScore: Retrieves the text search score for the document. This score indicates the relevance of the document to the text search query.

Example (MongoDB Shell)

The following example demonstrates how to use the $meta operator to retrieve the text search score for documents matching a text search query.

Create sample documents

```
db.documents.insertMany([
  { _id: 1, title: "Coffee Basics", content: "Coffee is a popular beverage made from
 roasted coffee beans." },
  { _id: 2, title: "Coffee Culture", content: "Coffee coffee coffee - the ultimate
 guide to coffee brewing and coffee preparation." },
  { _id: 3, title: "Tea vs Coffee", content: "Many people prefer tea over coffee for
 its health benefits." }
]);
```

Create text index

```
db.documents.createIndex({ content: "text" });
```

Query example

```
db.documents.find(
  { $text: { $search: "coffee" } },
  { _id: 0, title: 1, content: 1, score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } });
```

Output

```
[
  {
    title: 'Coffee Culture',
    content: 'Coffee coffee coffee - the ultimate guide to coffee brewing and coffee
 preparation.',
    score: 0.8897688388824463
  },
  {
    title: 'Coffee Basics',
    content: 'Coffee is a popular beverage made from roasted coffee beans.',
    score: 0.75990891456604
  },
  {
    title: 'Tea vs Coffee',
    content: 'Many people prefer tea over coffee for its health benefits.',
    score: 0.6079270839691162
  }
]
```

Code examples

To view a code example for using the `$meta` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findWithTextScore() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('documents');

  const result = await collection.find(
    { $text: { $search: "coffee" } },
    { projection: { _id: 0, title: 1, content: 1, score: { $meta: "textScore" } } }
  ).sort({ score: { $meta: "textScore" } }).toArray();

  console.log(result);
  client.close();
}
```

```
findWithTextScore();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['documents']

for doc in collection.find(
    {'$text': {'$search': 'coffee'}},
    {'_id': 0, 'title': 1, 'content': 1, 'score': {'$meta': 'textScore'}}
).sort([('score', {'$meta': 'textScore'})]):
    print(doc)

client.close()
```

$ne

The $ne operator is used to match documents where the value of a field is not equal to the specified value. It is a comparison operator that can be used in query predicates to filter documents.

Planner version 2.0 added index support for $ne.

Parameters

- `field`: The field to check.
- `value`: The value to check against.

Example (MongoDB Shell)

In this example, we'll find all documents in the `users` collection where the `status` field is not equal to `"active"`.

Create sample documents

```
db.users.insertMany([
  { name: "John", status: "active" },
  { name: "Jane", status: "inactive" },
  { name: "Bob", status: "suspended" },
  { name: "Alice", status: "active" }
]);
```

Query example

```
db.users.find({ status: { $ne: "active" } });
```

Output

```
[
  {
    _id: ObjectId('...'),
    name: 'Jane',
    status: 'inactive'
  },
  {
    _id: ObjectId('...'),
    name: 'Bob',
    status: 'suspended'
  }
]
```

Code examples

To view a code example for using the $ne command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');
```

```
  const result = await users.find({ status: { $ne: 'active' } }).toArray();
  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
users = db['users']

result = list(users.find({ 'status': { '$ne': 'active' } }))
print(result)

client.close()
```

$nin

The `$nin` operator is used to match values that are not in the specified array. It is the inverse of the `$in` operator, which matches values that are in the specified array.

Planner version 2.0 added index support for `$nin`.

Parameters

- `field`: The field to check.
- `array`: The array of values to check against.

Dollar ($) in field names

See Dollar($) and dot(.) in field names for limitations regarding querying $ prefixed fields in `$nin` in nested objects.

Example (MongoDB Shell)

The following example demonstrates how to use the `$nin` operator to find documents where the `category` field is not equal to "Fiction" or "Mystery".

Create sample documents

```
db.books.insertMany([
  { title: "The Great Gatsby", author: "F. Scott Fitzgerald", category: "Fiction" },
  { title: "To Kill a Mockingbird", author: "Harper Lee", category: "Fiction" },
  { title: "The Girl on the Train", author: "Paula Hawkins", category: "Mystery" },
  { title: "The Martian", author: "Andy Weir", category: "Science Fiction" },
  { title: "The Alchemist", author: "Paulo Coelho", category: "Philosophy" }
])
```

Query example

```
db.books.find({
  category: {
    $nin: ["Fiction", "Mystery"]
  }
})
```

Output

```
[
  {
    _id: ObjectId('...'),
    title: 'The Martian',
    author: 'Andy Weir',
    category: 'Science Fiction'
  },
  {
    _id: ObjectId('...'),
    title: 'The Alchemist',
    author: 'Paulo Coelho',
    category: 'Philosophy'
  }
]
```

Code examples

To view a code example for using the $nin command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findBooksNotInCategories(categories) {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const books = await db.collection('books').find({
    category: {
      $nin: categories
    }
  }).toArray();
  console.log(books);
  client.close();
}

findBooksNotInCategories(['Fiction', 'Mystery']);
```

Python

```
from pymongo import MongoClient

def find_books_not_in_categories(categories):
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    books = list(db.books.find({
        'category': {
            '$nin': categories
        }
    }))
    print(books)
    client.close()

find_books_not_in_categories(['Fiction', 'Mystery'])
```

$nor

The $nor operator is used to match documents where none of the specified query conditions are true. It is similar to the logical "NOR" operation, where the result is true if none of the operands are true.

Parameters

- expression1: The first expression to evaluate.
- expression2: The second expression to evaluate.
- expressionN: Additional expressions to evaluate.

Example (MongoDB Shell)

The following example demonstrates the usage of the $nor operator by retrieving documents where the qty field is not less than 20 and the size field is not equal to "XL".

Create sample documents

```
db.items.insertMany([
  { qty: 10, size: "M" },
  { qty: 15, size: "XL" },
  { qty: 25, size: "L" },
  { qty: 30, size: "XL" }
])
```

Query example

```
db.items.find({
  $nor: [
    { qty: { $lt: 20 } },
    { size: "XL" }
  ]
})
```

Output

```
[
  { "_id" : ObjectId("..."), "qty" : 25, "size" : "L" }
```

```
]
```

Code examples

To view a code example for using the $nor command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('items');

  const result = await collection.find({
    $nor: [
      { qty: { $lt: 20 } },
      { size: "XL" }
    ]
  }).toArray();

  console.log(result);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['items']
```

```
    result = list(collection.find({
        '$nor': [
            { 'qty': { '$lt': 20 } },
            { 'size': 'XL' }
        ]
    }))

    print(result)

    client.close()

example()
```

$not

The $not operator is used to negate the result of a given expression. It allows you to select documents where the specified condition does not match.

Planner version 2.0 added index support for $not {eq} and $not {in}.

Parameters

- expression: The expression to negate.

Example (MongoDB Shell)

The following example demonstrates how to use the $not operator to find documents where the status field is not equal to "active".

Create sample documents

```
db.users.insertMany([
  { name: "John", status: "active" },
  { name: "Jane", status: "inactive" },
  { name: "Bob", status: "pending" },
  { name: "Alice", status: "active" }
]);
```

Query example

```
db.users.find({ status: { $not: { $eq: "active" } } });
```

Output

```
[
  {
    _id: ObjectId('...'),
    name: 'Jane',
    status: 'inactive'
  },
  {
    _id: ObjectId('...'),
    name: 'Bob',
    status: 'pending'
  }
]
```

Code examples

To view a code example for using the $not command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient, Filters } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const result = await collection.find({
    status: { $not: { $eq: "active" } }
  }).toArray();

  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient
from bson.son import SON

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['users']

result = collection.find({
    "status": {"$not": {"$eq": "active"}}
})

for doc in result:
    print(doc)

client.close()
```

$or

The `$or` operator is used to perform a logical OR operation on an array of two or more expressions. It returns documents that match at least one of the expressions. This operator is useful when you need to query for documents that satisfy any one of multiple conditions.

Parameters

- `expression1`: The first expression to evaluate.
- `expression2`: The second expression to evaluate.
- `...`: Additional expressions to evaluate (optional).

Example (MongoDB Shell)

The following example demonstrates the usage of the `$or` operator to find documents where the `make` is either "TruckForYou" with the model "Heavy H1" or "SportForYou" with the model "Bolid 1".

Create sample documents

```
db.cars.insertMany([
```

```
  { make: "TruckForYou", model: "Heavy H1", year: 2020 },
  { make: "SportForYou", model: "Bolid 1", year: 2021 },
  { make: "TruckForYou", model: "Cargo 5", year: 2019 },
  { make: "SportForYou", model: "Racer 2", year: 2022 }
]);
```

Query example

```
db.cars.find({
  $or: [
    { make: "TruckForYou", model: "Heavy H1" },
    { make: "SportForYou", model: "Bolid 1" }
  ]
});
```

Output

```
[
  {
    _id: ObjectId('...'),
    make: 'TruckForYou',
    model: 'Heavy H1',
    year: 2020
  },
  {
    _id: ObjectId('...'),
    make: 'SportForYou',
    model: 'Bolid 1',
    year: 2021
  }
]
```

Code examples

To view a code example for using the `$or` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function findCarsByMakeModel() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const cars = db.collection('cars');

  const result = await cars.find({
    $or: [
      { make: "TruckForYou", model: "Heavy H1" },
      { make: "SportForYou", model: "Bolid 1" }
    ]
  }).toArray();

  console.log(result);
  client.close();
}

findCarsByMakeModel();
```

Python

```
from pymongo import MongoClient

def find_cars_by_make_model():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    cars = db.cars

    result = list(cars.find({
        '$or': [
            {'make': 'TruckForYou', 'model': 'Heavy H1'},
            {'make': 'SportForYou', 'model': 'Bolid 1'}
        ]
    }))

    print(result)
    client.close()

find_cars_by_make_model()
```

$regex

The `$regex` operator allows you to perform regular expression matching on string fields. It is a powerful tool for searching and filtering documents based on complex patterns.

Parameters

- `regular expression`: The regular expression pattern to match against the field.
- `$options`: (optional) Provides options to modify the search behavior, such as case-sensitivity, global matching, etc.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$regex` operator to search for documents where the "name" field matches a specific pattern.

Create sample documents

```
db.users.insertMany([
  { name: "John Doe" },
  { name: "Jane Smith" },
  { name: "Alice Johnson" },
  { name: "Bob Williams" },
  { name: "Charlie Davis" }
]);
```

Query example

```
db.users.find({ name: { $regex: /^A/ } })
```

Output

```
[
  { "_id" : ObjectId("..."), "name" : "Alice Johnson" }
]
```

This query will return all documents where the "name" field starts with the letter "A".

Code examples

To view a code example for using the `$regex` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const results = await collection.find({ name: { $regex: /^A/ } }).toArray();
  console.log(results);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['users']

    results = list(collection.find({ 'name': { '$regex': '^A' } }))
    print(results)

    client.close()

if __name__ == "__main__":
    main()
```

$sampleRate

New from version 8.0.1.

Not supported by Elastic cluster.

The `$sampleRate` operator in Amazon DocumentDB matches a random sample of the input documents based on the specified rate. You can use it in a `find()` query filter or in an aggregation `$match` stage. Because selection is probabilistic, the number of documents returned is approximate and can vary between runs.

Parameters

- `value`: A number in the range `[0, 1]` (inclusive), passed directly to the operator (for example, `{ $sampleRate: 0.3 }`), that sets the probability of including each document. A value of `0` returns no documents; a value of 1 returns all documents.

Example (MongoDB Shell)

The following example uses `$sampleRate` to return approximately 30 percent of the documents in a collection.

Create sample documents

```
db.events.insertMany([
  { _id: 1, type: "click" },
  { _id: 2, type: "view" },
  { _id: 3, type: "click" },
  { _id: 4, type: "view" },
  { _id: 5, type: "purchase" }
]);
```

Query example

```
db.events.aggregate([
  { $match: { $sampleRate: 0.3 } }
]);
```

Output

The operation returns a random subset of the documents. Because `$sampleRate` is probabilistic, the specific documents returned and their count vary each time the query runs.

Code examples

To view a code example for using the `$sampleRate` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const uri = 'mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const db = client.db('test');
    const collection = db.collection('events');

    // Return approximately 30% of the documents
    const result = await collection.aggregate([
      { $match: { $sampleRate: 0.3 } }
    ]).toArray();

    console.log(result);
  } finally {
    await client.close();
  }
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')

    try:
        db = client['test']
        collection = db['events']

        # Return approximately 30% of the documents
        result = list(collection.aggregate([
            { '$match': { '$sampleRate': 0.3 } }
        ]))

        print(result)
    finally:
        client.close()

example()
```

$slice

The `$slice` projection operator limits the number of array elements returned in a query result. It allows you to retrieve a specific number of elements from the start or end of an array field without loading the entire array.

Parameters

- `field`: The array field to project.
- `count`: Number of elements to return. Positive values return elements from the start, negative values from the end.

Example (MongoDB Shell)

The following example demonstrates how to use the `$slice` projection operator to return only the first two items from an array field.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "notebook", tags: ["office", "school", "supplies", "writing"] },
  { _id: 2, item: "pen", tags: ["office", "writing"] },
```

```
  { _id: 3, item: "folder", tags: ["office", "supplies", "storage", "organization"] }
]);
```

Query example

```
db.inventory.find(
  {},
  { item: 1, tags: { $slice: 2 } }
)
```

Output

```
{ "_id" : 1, "item" : "notebook", "tags" : [ "office", "school" ] }
{ "_id" : 2, "item" : "pen", "tags" : [ "office", "writing" ] }
{ "_id" : 3, "item" : "folder", "tags" : [ "office", "supplies" ] }
```

Code examples

To view a code example for using the `$slice` projection operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('inventory');

  const result = await collection.find(
    {},
    { projection: { item: 1, tags: { $slice: 2 } } }
  ).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['inventory']

    result = list(collection.find(
        {},
        {'item': 1, 'tags': {'$slice': 2}}
    ))

    print(result)
    client.close()

example()
```

$size

The `$size` query operator matches documents where an array field has exactly the specified number of elements. This is useful for filtering documents based on array length.

Parameters

- `field`: The array field to check.
- `count`: The exact number of elements the array must contain.

Example (MongoDB Shell)

The following example demonstrates using the `$size` operator to find all products that have exactly three tags.

Create sample documents

```
db.products.insertMany([
  { _id: 1, name: "Laptop", tags: ["electronics", "computers", "portable"] },
  { _id: 2, name: "Mouse", tags: ["electronics", "accessories"] },
```

```
  { _id: 3, name: "Desk", tags: ["furniture", "office", "workspace"] },
  { _id: 4, name: "Monitor", tags: ["electronics"] }
]);
```

Query example

```
db.products.find({ tags: { $size: 3 } });
```

Output

```
{ "_id" : 1, "name" : "Laptop", "tags" : [ "electronics", "computers", "portable" ] }
{ "_id" : 3, "name" : "Desk", "tags" : [ "furniture", "office", "workspace" ] }
```

Code examples

To view a code example for using the `$size` query operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  const result = await collection.find({ tags: { $size: 3 } }).toArray();

  console.log(JSON.stringify(result, null, 2));
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['products']

    result = list(collection.find({'tags': {'$size': 3}}))

    print(result)
    client.close()

example()
```

$text

The `$text` operator is used to perform full-text search on text-indexed fields within a document collection. This operator allows you to search for documents that contain specific words or phrases, and can be combined with other query operators to filter results based on additional criteria.

Parameters

- `$search`: The text string to search for.

Example (MongoDB Shell)

The following example demonstrates how to use the `$text` operator to search for documents containing the word "interest" and filter the results based on a "star_rating" field.

Create sample documents

```
db.test.insertMany([
  { "_id": 1, "star_rating": 4, "comments": "apple is red" },
  { "_id": 2, "star_rating": 5, "comments": "pie is delicious" },
  { "_id": 3, "star_rating": 3, "comments": "apples, oranges - healthy fruit" },
  { "_id": 4, "star_rating": 2, "comments": "bake the apple pie in the oven" },
  { "_id": 5, "star_rating": 5, "comments": "interesting couch" },
  { "_id": 6, "star_rating": 5, "comments": "interested in couch for sale, year 2022" }
]);
```

Create text index

```
db.test.createIndex({ comments: "text" });
```

Query example

```
db.test.find({$and: [{star_rating: 5}, {$text: {$search: "interest"}}]})
```

Output

```
{ "_id" : 5, "star_rating" : 5, "comments" : "interesting couch" }
{ "_id" : 6, "star_rating" : 5, "comments" : "interested in couch for sale, year
 2022" }
```

The command above returns documents with a text-indexed field containing any form of "interest" and a "star_rating" equal to 5.

Code examples

To view a code example for using the `$text` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function searchDocuments() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('test');

  const result = await collection.find({
    $and: [
      { star_rating: 5 },
      { $text: { $search: 'interest' } }
    ]
  }).toArray();

  console.log(result);
  client.close();
}
```

```
searchDocuments();
```

Python

```
from pymongo import MongoClient

def search_documents():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db.test

    result = list(collection.find({
        '$and': [
            {'star_rating': 5},
            {'$text': {'$search': 'interest'}}
        ]
    }))

    print(result)
    client.close()

search_documents()
```

$type

The `$type` operator is used to check the data type of a field in a document. It can used when type-specific operations or validations are needed. The `$type` operator returns the BSON type of the evaluated expression. The returned type is a string, which corresponds to the type of the field or expression.

Planner version 2.0 added index support for `$type`.

Parameters

- `expression` : The expression to evaluate.

Example (MongoDB Shell)

Create sample documents

```
db.documents.insertMany([
  { _id: 1, name: "John", age: 30, email: "john@example.com" },
  { _id: 2, name: "Jane", age: "25", email: 123456 },
  { _id: 3, name: 123, age: true, email: null }
]);
```

Query example

```
db.documents.find({
  $or: [
    { age: { $type: "number" } },
    { email: { $type: "string" } },
    { name: { $type: "string" } }
  ]
})
```

Output

```
[
  { "_id": 1, "name": "John", "age": 30, "email": "john@example.com" },
  { "_id": 2, "name": "Jane", "age": "25", "email": 123456 }
]
```

This query will return the documents where the age field is of type "number", the email field is of type "string", and the name field is of type "string".

Code examples

To view a code example for using the $type command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function findByType() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('documents');
```

```
  const results = await collection.find({
    $or: [
      { age: { $type: 'number' } },
      { email: { $type: 'string' } },
      { name: { $type: 'string' } }
    ]
  }).toArray();

  console.log(results);
  client.close();
}

findByType();
```

Python

```
from pymongo import MongoClient

def find_by_type():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['documents']

    results = list(collection.find({
        '$or': [
            {'age': {'$type': 'number'}},
            {'email': {'$type': 'string'}},
            {'name': {'$type': 'string'}}
        ]
    }))

    print(results)
    client.close()

find_by_type()
```

Update operators

This section provides detailed information about update operators supported by Amazon DocumentDB.

Topics

- $
- $[]
- $[<identifier>]
- $addToSet
- $bit
- $currentDate
- $each
- $inc
- $max
- $min
- $mul
- $pop
- $position
- $pull
- $pullAll
- $push
- $rename
- $set
- $setOnInsert
- $slice
- $sort
- $unset

$

The $ positional operator updates the first array element that matches the query condition. It acts as a placeholder for the matched array element's position.

Parameters

- `field.$`: The array field with the positional operator to update the first matching element.

Example (MongoDB Shell)

The following example demonstrates using the $ positional operator to update a specific array element.

Create sample documents

```
db.inventory.insertMany([
  { _id: 1, item: "Widget", quantities: [10, 20, 30] },
  { _id: 2, item: "Gadget", quantities: [5, 15, 25] }
]);
```

Query example

```
db.inventory.updateOne(
  { _id: 1, quantities: 20 },
  { $set: { "quantities.$": 22 } }
);
```

Output

```
{
  "_id" : 1,
  "item" : "Widget",
  "quantities" : [ 10, 22, 30 ]
}
```

Code examples

To view a code example for using the $ positional operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
```

```
  const collection = db.collection('inventory');

  await collection.updateOne(
    { _id: 1, quantities: 20 },
    { $set: { "quantities.$": 22 } }
  );

  const updatedDocument = await collection.findOne({ _id: 1 });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.inventory

    collection.update_one(
        {'_id': 1, 'quantities': 20},
        {'$set': {'quantities.$': 22}}
    )

    updated_document = collection.find_one({'_id': 1})
    print(updated_document)

    client.close()

update_document()
```

$[]

The `$[]` all positional operator updates all elements in an array. It is used when you need to modify every element in an array field.

Parameters

- `field.$[]`: The array field with the all positional operator to update all elements.

Example (MongoDB Shell)

The following example demonstrates using the `$[]` operator to update all array elements.

Create sample documents

```
db.products.insertOne({
  _id: 1,
  name: "Laptop",
  prices: [1000, 1100, 1200]
});
```

Query example

```
db.products.updateOne(
  { _id: 1 },
  { $inc: { "prices.$[]": 50 } }
);
```

Output

```
{
  "_id" : 1,
  "name" : "Laptop",
  "prices" : [ 1050, 1150, 1250 ]
}
```

Code examples

To view a code example for using the `$[]` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');
```

```
async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  await collection.updateOne(
    { _id: 1 },
    { $inc: { "prices.$[]": 50 } }
  );

  const updatedDocument = await collection.findOne({ _id: 1 });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.products

    collection.update_one(
        {'_id': 1},
        {'$inc': {'prices.$[]': 50}}
    )

    updated_document = collection.find_one({'_id': 1})
    print(updated_document)

    client.close()

update_document()
```

$[<identifier>]

The `$[<identifier>]` filtered positional operator updates all array elements that match the specified filter conditions. It is used with the `arrayFilters` option to selectively update array elements.

Parameters

- `field.$[identifier]`: The array field with the filtered positional operator.
- `arrayFilters`: An array of filter conditions that determine which elements to update.

Example (MongoDB Shell)

The following example demonstrates using the `$[<identifier>]` operator to update specific array elements based on a condition.

Create sample documents

```
db.students.insertOne({
  _id: 1,
  name: "Alice",
  grades: [
    { subject: "Math", score: 85 },
    { subject: "Science", score: 92 },
    { subject: "History", score: 78 }
  ]
});
```

Query example

```
db.students.updateOne(
  { _id: 1 },
  { $inc: { "grades.$[elem].score": 5 } },
  { arrayFilters: [{ "elem.score": { $gte: 80 } }] }
);
```

Output

```
{
  "_id" : 1,
```

```
  "name" : "Alice",
  "grades" : [
    { "subject" : "Math", "score" : 90 },
    { "subject" : "Science", "score" : 97 },
    { "subject" : "History", "score" : 78 }
  ]
}
```

Code examples

To view a code example for using the `$[<identifier>]` operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('students');

  await collection.updateOne(
    { _id: 1 },
    { $inc: { "grades.$[elem].score": 5 } },
    { arrayFilters: [{ "elem.score": { $gte: 80 } }] }
  );

  const updatedDocument = await collection.findOne({ _id: 1 });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.students

    collection.update_one(
        {'_id': 1},
        {'$inc': {'grades.$[elem].score': 5}},
        array_filters=[{'elem.score': {'$gte': 80}}]
    )

    updated_document = collection.find_one({'_id': 1})
    print(updated_document)

    client.close()

update_document()
```

$addToSet

The `$addToSet` operator in Amazon DocumentDB is used to add a value to an array only if the value is not already present in the array. This is useful for ensuring that an array contains unique elements.

Parameters

- `field`: The field to update.
- `value`: The value to add to the array field. This can be a single value or an expression.

Example (MongoDB Shell)

The following example demonstrates how to use the `$addToSet` operator to add unique elements to an array.

Create sample documents

```
db.products.insertMany([
  { "_id": 1, "item": "apple", "tags": ["fruit", "red", "round"] },
  { "_id": 2, "item": "banana", "tags": ["fruit", "yellow"] },
```

```
  { "_id": 3, "item": "cherry", "tags": ["fruit", "red"] }
])
```

Query example

```
db.products.update(
  { "item": "apple" },
  { $addToSet: { "tags": "green" } }
)
```

Output

```
{ "_id": 1, "item": "apple", "tags": ["fruit", "red", "round", "green"] }
```

In this example, the `$addToSet` operator adds the "green" tag to the "tags" array of the document where the "item" field is "apple". Since "green" was not already in the array, it was added.

Code examples

To view a code example for using the `$addToSet` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('products');

  await collection.updateOne(
    { "item": "apple" },
    { $addToSet: { "tags": "green" } }
  );

  const updatedDoc = await collection.findOne({ "item": "apple" });
  console.log(updatedDoc);
```

```
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.products

    collection.update_one(
        {"item": "apple"},
        {"$addToSet": {"tags": "green"}}
    )

    updated_doc = collection.find_one({"item": "apple"})
    print(updated_doc)

    client.close()

example()
```

$bit

The `$bit` operator in Amazon DocumentDB allows you to perform bitwise operations on the bits of a given field. This can be useful for tasks like setting, clearing, or checking the state of individual bits within a number.

Parameters

- `field`: The field to perform bitwise operations on.
- `and`: An integer value that is used to perform a bitwise AND operation on the field.
- `or`: An integer value that is used to perform a bitwise OR operation on the field.
- `xor`: An integer value that is used to perform a bitwise XOR operation on the field.

Example (MongoDB Shell)

The following example demonstrates how to use the $bit operator to perform bitwise operations on a numerical field.

Create sample documents

```
db.numbers.insert([
  { "_id": 1, "number": 5 },
  { "_id": 2, "number": 12 }
])
```

Query example

```
db.numbers.update(
  { "_id": 1 },
  { "$bit": { "number": { "and": 3 } } }
)
```

Output

```
{
  "_id": 1,
  "number": 1
}
```

In this example, the $bit operator is used to perform a bitwise AND operation on the "number" field of the document with the _id of 1. The result is that the value of the "number" field is set to 1, which is the result of the bitwise AND operation between the original value of 5 and the value 3.

Code examples

To view a code example for using the $bit command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('numbers');

  await collection.updateOne(
    { "_id": 1 },
    { "$bit": { "number": { "and": 3 } } }
  );

  const result = await collection.findOne({ "_id": 1 });
  console.log(result);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
db = client['test']
collection = db['numbers']

collection.update_one(
    {"_id": 1},
    {"$bit": {"number": {"and": 3}}}
)

result = collection.find_one({"_id": 1})
print(result)

client.close()
```

$currentDate

The `$currentDate` operator is used to set the value of a field to the current date and time. This operator is useful for automatically updating a field with the current timestamp when a document is inserted or updated.

Parameters

- `field`: The field to update with the current date and time.
- `type`: (optional) Specifies the BSON type to use for the current date. Can be either `date` or `timestamp`.

Example (MongoDB Shell)

The following example demonstrates how to use the `$currentDate` operator to set the `lastModified` field to the current date and time when a new document is inserted.

Create sample documents

```
db.users.insert({
  name: "John Doe",
  email: "john.doe@example.com"
})
```

Query example

```
db.users.updateOne(
  { name: "John Doe" },
  { $currentDate: { lastModified: true } }
)
```

View updated document

```
db.users.findOne({ name: "John Doe" })
```

Output

```
{
```

```
  _id: ObjectId('...'),
  name: 'John Doe',
  email: 'john.doe@example.com',
  lastModified: ISODate('2025-10-25T22:50:29.963Z')
}
```

Code examples

To view a code example for using the `$currentDate` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateUserWithCurrentDate() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  await users.updateOne(
    { name: 'John Doe' },
    { $currentDate: { lastModified: true } }
  );

  console.log('User updated with current date');
  client.close();
}

updateUserWithCurrentDate();
```

Python

```
from pymongo import MongoClient

def update_user_with_current_date():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    users = db.users
```

```
    result = users.update_one(
        {'name': 'John Doe'},
        {'$currentDate': {'lastModified': True}}
    )

    print('User updated with current date')
    client.close()

update_user_with_current_date()
```

$each

The $each operator is used in conjunction with other update operators, such as $push and $addToSet, to add multiple values to an array field. It allows adding multiple elements to an array in a single operation, rather than having to execute multiple update operations.

Parameters

- value: The array of values to add to the array field.

Example (MongoDB Shell)

The following example demonstrates using the $each operator with the $push operator to add multiple elements to an array field.

Create sample documents

```
db.fruits.insertOne({
  _id: 1,
  fruits: ["apple", "banana"]
})
```

Query example

```
db.fruits.updateOne(
  { _id: 1 },
  { $push: { fruits: { $each: ["cherry", "durian", "elderberry"] } } }
)
```

View updated document

```
db.fruits.findOne({ _id: 1 })
```

Output

```
{
  _id: 1,
  fruits: [ 'apple', 'banana', 'cherry', 'durian', 'elderberry' ]
}
```

Code examples

To view a code example for using the $each command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('fruits');

  await collection.updateOne(
    { _id: 1 },
    { $push: { fruits: { $each: ["cherry", "durian", "elderberry"] } } }
  );

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['fruits']

    collection.update_one(
        {'_id': 1},
        {'$push': {'fruits': {'$each': ['cherry', 'durian', 'elderberry']}}}
    )

    client.close()

example()
```

$inc

The `$inc` operator is used to increment the value of a field by a specified amount. It is used to update a numeric field, such as a counter or a rating, without having to retrieve the current value, calculate the new value, and then update the field.

Parameters

- `field`: The name of the field to increment.
- `amount`: The amount by which to increment the field. This can be a positive or negative value.

Example (MongoDB Shell)

The following example demonstrates how to use the `$inc` operator to increment the `age` field of a document.

Create sample documents

```
db.users.insertOne({_id: 123, name: "John Doe", age: 30})
```

Query example

```
db.users.updateOne({_id: 123}, {$inc: {age: 1}})
```

View updated document

```
db.users.findOne({_id: 123})
```

Output

```
{ "_id" : 123, "name" : "John Doe", "age" : 31 }
```

Code examples

To view a code example for using the $inc command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateWithInc() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const result = await collection.updateOne(
    { _id: 123 },
    { $inc: { age: 1 } }
  );

  console.log(result);

  await client.close();
}

updateWithInc();
```

Python

```
from pymongo import MongoClient

def update_with_inc():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
    db = client['test']
    collection = db['users']

    result = collection.update_one(
        {'_id': 123},
        {'$inc': {'age': 1}}
    )

    print(result.modified_count)

    client.close()

update_with_inc()
```

$max

The $max update operator updates a field's value only if the specified value is greater than the current field value. This operator is useful for maintaining maximum values across updates.

Parameters

- field: The field to update.
- value: The value to compare with the current field value.

Example (MongoDB Shell)

The following example demonstrates using the $max operator to update the highest recorded score for a player.

Create sample documents

```
db.scores.insertMany([
  { _id: 1, player: "Alice", highScore: 85 },
  { _id: 2, player: "Bob", highScore: 92 },
  { _id: 3, player: "Charlie", highScore: 78 }
])
```

Update example

```
db.scores.updateOne(
```

```
  { _id: 1 },
  { $max: { highScore: 95 } }
)
```

Result

The `highScore` field for Alice is updated to 95 because 95 is greater than the current value of 85.

```
{ "_id": 1, "player": "Alice", "highScore": 95 }
```

Code examples

To view a code example for using the `$max` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('scores');

  const result = await collection.updateOne(
    { _id: 1 },
    { $max: { highScore: 95 } }
  );

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['scores']

    result = collection.update_one(
        { '_id': 1 },
        { '$max': { 'highScore': 95 } }
    )

    print(result)
    client.close()

example()
```

$min

The $min update operator updates a field's value only if the specified value is less than the current field value. This operator is useful for maintaining minimum values across updates.

Parameters

- field: The field to update.
- value: The value to compare with the current field value.

Example (MongoDB Shell)

The following example demonstrates using the $min operator to update the lowest recorded temperature for a weather station.

Create sample documents

```
db.weather.insertMany([
  { _id: 1, station: "Station A", lowestTemp: 15 },
  { _id: 2, station: "Station B", lowestTemp: 20 },
  { _id: 3, station: "Station C", lowestTemp: 18 }
])
```

Update example

```
db.weather.updateOne(
  { _id: 1 },
  { $min: { lowestTemp: 12 } }
)
```

Result

The `lowestTemp` field for Station A is updated to 12 because 12 is less than the current value of 15.

```
{ "_id": 1, "station": "Station A", "lowestTemp": 12 }
```

Code examples

To view a code example for using the `$min` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('weather');

  const result = await collection.updateOne(
    { _id: 1 },
    { $min: { lowestTemp: 12 } }
  );

  console.log(result);
  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient
```

```
def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['weather']

    result = collection.update_one(
        { '_id': 1 },
        { '$min': { 'lowestTemp': 12 } }
    )

    print(result)
    client.close()

example()
```

$mul

The `$mul` operator in Amazon DocumentDB is used to multiply the value of a field by a specified number. This can be useful for updating multiple documents atomically and consistently, such as updating flight miles based on a credit card status.

Parameters

- `field`: The field to be multiplied.
- `multiplier`: The number to multiply the field value by.

Example (MongoDB Shell)

This example demonstrates how to use the `$mul` operator to double the `flight_miles` value for all documents where the `credit_card` field is `true`.

Create sample documents

```
db.miles.insertMany([
  { "_id": 1, "member_since": new Date("1987-01-01"), "credit_card": false,
 "flight_miles": [1205, 2560, 880] },
  { "_id": 2, "member_since": new Date("1982-01-01"), "credit_card": true,
 "flight_miles": [2410, 5120, 1780, 5560] },
```

```
  { "_id": 3, "member_since": new Date("1999-01-01"), "credit_card": true,
 "flight_miles": [2410, 1760] }
]);
```

Query example

```
db.miles.update(
  { "credit_card": { "$eq": true } },
  { "$mul": { "flight_miles.$[]": NumberInt(2) } },
  { "multi": true }
);
```

Output

```
{ "_id" : 1, "member_since" : ISODate("1987-01-01T00:00:00Z"), "credit_card" : false,
 "flight_miles" : [ 1205, 2560, 880 ] }
{ "_id" : 2, "member_since" : ISODate("1982-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 4820, 10240, 3560, 11120 ] }
{ "_id" : 3, "member_since" : ISODate("1999-01-01T00:00:00Z"), "credit_card" : true,
 "flight_miles" : [ 4820, 3520 ] }
```

For the customers that have a credit card, their flight miles have been doubled.

The `$[]` positional array operator is used to apply the `$mul` operation to each element in the `flight_miles` array.

Code examples

To view a code example for using the `$mul` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateFlightMiles() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('miles');
```

```
  await collection.updateMany(
    { credit_card: true },
    { $mul: { 'flight_miles.$[]': 2 } }
  );

  const documents = await collection.find().toArray();
  console.log(documents);

  await client.close();
}

updateFlightMiles();
```

Python

```
from pymongo import MongoClient

def update_flight_miles():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.miles

    collection.update_many(
        {'credit_card': True},
        {'$mul': {'flight_miles.$[]': 2}}
    )

    documents = list(collection.find())
    print(documents)

    client.close()

update_flight_miles()
```

$pop

The $pop operator in Amazon DocumentDB is used to remove the first or last element from an array field. It is particularly useful when you need to maintain a fixed-size array or implement a queue-like data structure within a document.

Parameters

- `field`: The name of the array field to remove an element from.
- `value`: An integer value that determines the position of the element to remove. A value of `1` removes the last element, while a value of `-1` removes the first element.

Example (MongoDB Shell)

This example demonstrates how to use the $pop operator to remove the first and last elements from an array field.

Create sample documents

```
db.users.insertMany([
  { "_id": 1, "name": "John Doe", "hobbies": ["reading", "swimming", "hiking"] },
  { "_id": 2, "name": "Jane Smith", "hobbies": ["cooking", "gardening", "painting"] }
])
```

Query example

```
// Remove the first element from the "hobbies" array
db.users.update({ "_id": 1 }, { $pop: { "hobbies": -1 } })

// Remove the last element from the "hobbies" array
db.users.update({ "_id": 2 }, { $pop: { "hobbies": 1 } })
```

Output

```
{ "_id" : 1, "name" : "John Doe", "hobbies" : [ "swimming", "hiking" ] }
{ "_id" : 2, "name" : "Jane Smith", "hobbies" : [ "cooking", "gardening" ] }
```

Code examples

To view a code example for using the $pop command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  // Remove the first element from the "hobbies" array
  await collection.updateOne({ "_id": 1 }, { $pop: { "hobbies": -1 } });

  // Remove the last element from the "hobbies" array
  await collection.updateOne({ "_id": 2 }, { $pop: { "hobbies": 1 } });

  const users = await collection.find().toArray();
  console.log(users);

  await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['users']

    # Remove the first element from the "hobbies" array
    collection.update_one({"_id": 1}, {"$pop": {"hobbies": -1}})

    # Remove the last element from the "hobbies" array
    collection.update_one({"_id": 2}, {"$pop": {"hobbies": 1}})

    users = list(collection.find())
    print(users)

    client.close()

example()
```

$position

The $position modifier in Amazon DocumentDB specifies the location in the array at which the $push operator inserts elements. Without the $position modifier, the $push operator inserts elements to the end of the array.

Parameters

- field: The array field to update.
- num: The position in the array where elements should be inserted, based on zero-based indexing.

Note: To use the $position modifier, it must appear with the $each modifier.

Example (MongoDB Shell)

The following example demonstrates how to use the $position operator to insert tasks at specific positions in a project management system.

Create sample documents

```
db.projects.insertOne({ "_id": 1, "name": "Website Redesign", "tasks": ["Design
 mockups"] })
```

Query example 1 - Add urgent tasks at the beginning

```
db.projects.updateOne(
   { _id: 1 },
   {
     $push: {
        tasks: {
           $each: ["Security audit", "Performance review"],
           $position: 0
        }
     }
   }
)
```

Output 1

```
{ "_id": 1, "name": "Website Redesign", "tasks": ["Security audit", "Performance
 review", "Design mockups"] }
```

Query example 2 - Add tasks at specific position

```
db.projects.insertOne({ "_id": 2, "name": "Mobile App", "tasks": ["Setup project",
 "Create wireframes", "Deploy to store"] })

db.projects.updateOne(
   { _id: 2 },
   {
     $push: {
        tasks: {
           $each: ["Code review", "Testing phase"],
           $position: 2
        }
     }
   }
)
```

Output 2

```
{ "_id": 2, "name": "Mobile App", "tasks": ["Setup project", "Create wireframes", "Code
 review", "Testing phase", "Deploy to store"] }
```

Code examples

To view a code example for using the $position command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function insertTasksAtPosition() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('projects');

  await collection.updateOne(
    { _id: 1 },
    {
      $push: {
```

```
        tasks: {
          $each: ["Security audit", "Performance review"],
          $position: 0
        }
      }
    }
  );

  const updatedProject = await collection.findOne({ _id: 1 });
  console.log(updatedProject);

  await client.close();
}

insertTasksAtPosition();
```

Python

```
from pymongo import MongoClient

def insert_tasks_at_position():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['projects']

    result = collection.update_one(
        {'_id': 1},
        {
            '$push': {
                'tasks': {
                    '$each': ['Security audit', 'Performance review'],
                    '$position': 0
                }
            }
        }
    )

    updated_project = collection.find_one({'_id': 1})
    print(updated_project)

    client.close()
```

```
insert_tasks_at_position()
```

$pull

The `$pull` operator is used to remove from an array all instances of a value or values that match a specified condition. This operator is useful when you need to remove specific elements from an array field within a document.

Parameters

- `field`: The name of the array field from which to remove the value(s).
- `value`: The value or condition that determines which element(s) to remove from the array.

Example (MongoDB Shell)

The following example demonstrates how to use the `$pull` operator to remove elements from an array field.

Create sample documents

```
db.restaurants.insertMany([
  {
    name: "Pizza Hut",
    cuisine: "Italian",
    features: ["Delivery", "Takeout", "Dine-in"]
  },
  {
    name: "Sushi Saito",
    cuisine: "Japanese",
    features: ["Dine-in", "Private Dining"]
  },
  {
    name: "Taco Bell",
    cuisine: "Mexican",
    features: ["Delivery", "Takeout", "Drive-thru"]
  }
])
```

Query example

```
db.restaurants.updateMany(
  { cuisine: "Italian" },
  { $pull: { features: "Takeout" } }
)
```

Output

```
{
  "acknowledged" : true,
  "matchedCount" : 1,
  "modifiedCount" : 1
}
```

The above query removes the "Takeout" feature from all documents where the `cuisine` field is "Italian".

Code examples

To view a code example for using the `$pull` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateRestaurants() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const restaurants = db.collection('restaurants');

  await restaurants.updateMany(
    { cuisine: 'Italian' },
    { $pull: { features: 'Takeout' } }
  );

  const updatedRestaurants = await restaurants.find({}).toArray();
  console.log(updatedRestaurants);

  await client.close();
}
```

```
updateRestaurants();
```

Python

```
from pymongo import MongoClient

def update_restaurants():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    restaurants = db.restaurants

    result = restaurants.update_many(
        { 'cuisine': 'Italian' },
        { '$pull': { 'features': 'Takeout' } }
    )

    updated_restaurants = list(restaurants.find({}))
    print(updated_restaurants)

    client.close()

update_restaurants()
```

$pullAll

The `$pullAll` operator in Amazon DocumentDB is used to remove all instances of the specified values from an array field. This is particularly useful when you need to remove multiple elements from an array in a single operation.

Parameters

- `field`: The name of the array field from which to remove the elements.
- `value`: An array of values to remove from the array field.

Example (MongoDB Shell)

The following example demonstrates how to use the `$pullAll` operator to remove multiple elements from an array field.

Create sample documents

```
db.restaurants.insert([
  {
    "name": "Taj Mahal",
    "cuisine": "Indian",
    "features": ["Private Dining", "Live Music"]
  },
  {
    "name": "Golden Palace",
    "cuisine": "Chinese",
    "features": ["Private Dining", "Takeout"]
  },
  {
    "name": "Olive Garden",
    "cuisine": "Italian",
    "features": ["Private Dining", "Outdoor Seating"]
  }
])
```

Query example

```
db.restaurants.update(
  { "name": "Taj Mahal" },
  { $pullAll: { "features": ["Private Dining", "Live Music"] } }
)
```

Output

```
{
  "name": "Taj Mahal",
  "cuisine": "Indian",
  "features": []
}
```

Code examples

To view a code example for using the `$pullAll` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function main() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('restaurants');

  await collection.updateMany(
    { "name": "Taj Mahal" },
    { $pullAll: { "features": ["Private Dining", "Live Music"] } }
  );

  const updatedDocument = await collection.findOne({ "name": "Taj Mahal" });
  console.log(updatedDocument);

  await client.close();
}

main();
```

Python

```
from pymongo import MongoClient

def main():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['restaurants']

    collection.update_many(
        {"name": "Taj Mahal"},
        {"$pullAll": {"features": ["Private Dining", "Live Music"]}}
    )

    updated_document = collection.find_one({"name": "Taj Mahal"})
    print(updated_document)
```

```
    client.close()

if __name__ == '__main__':
    main()
```

$push

The $push operator in Amazon DocumentDB is used to add an item to an array field in a document. This operator is particularly useful when you need to append new data to an existing array without overwriting the entire array.

Parameters

- `field`: The name of the array field to which the new element should be added.
- `value`: The value to be added to the array.
- `position`: (optional) A modifier that specifies the position in the array where the new element should be added. Supported modifiers include $ (add to the end of the array) and `$[]` (add to the end of the array, ignoring any array filters).

Example (MongoDB Shell)

The following example demonstrates how to use the $push operator to add new elements to an array field in a document.

Create sample documents

```
db.users.insert([
  { _id: 1, name: "John Doe", hobbies: ["reading", "swimming"] },
  { _id: 2, name: "Jane Smith", hobbies: ["gardening", "cooking"] }
])
```

Query example

```
db.users.updateOne(
  { _id: 1 },
  { $push: { hobbies: "hiking" } }
)
```

Output

```
{
  "acknowledged" : true,
  "matchedCount" : 1,
  "modifiedCount" : 1
}
```

After running the update, the document with `_id: 1` will have the `hobbies` array updated to `["reading", "swimming", "hiking"]`.

Code examples

To view a code example for using the `$push` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('users');

  const result = await collection.updateOne(
    { _id: 1 },
    { $push: { hobbies: "hiking" } }
  );

  console.log(result);
  client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['users']

    result = collection.update_one(
        {'_id': 1},
        {'$push': {'hobbies': 'hiking'}}
    )

    print(result.raw_result)
    client.close()

update_document()
```

$rename

The `$rename` operator in Amazon DocumentDB is used to rename a field in a document. This operator can be particularly useful when you need to update the structure of your documents or align them with new data models.

Parameters

- `field`: The field to be renamed.
- `newName`: The new name for the field.

Example (MongoDB Shell)

The following example demonstrates how to use the `$rename` operator to rename the `"Date.DoW"` field to `"Date.DayOfWeek"` in a document with the `"DocName"` field set to `"Document 1"`.

Create sample documents

```
db.example.insertOne({
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
```

```
        "DoW": "Saturday"
    },
    "Words": 2482
})
```

Query example

```
db.example.update(
    { "DocName": "Document 1" },
    { $rename: { "Date.DoW": "Date.DayOfWeek" } }
)
```

Output

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
        "DayOfWeek": "Saturday"
    },
    "Words": 2482
}
```

Code examples

To view a code example for using the `$rename` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function example() {
    const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
    const db = client.db('test');
    const collection = db.collection('example');

    await collection.updateOne(
```

```
        { "DocName": "Document 1" },
        { $rename: { "Date.DoW": "Date.DayOfWeek" } }
    );

    const updatedDoc = await collection.findOne({ "DocName": "Document 1" });
    console.log(updatedDoc);

    await client.close();
}

example();
```

Python

```
from pymongo import MongoClient

def example():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['example']

    collection.update_one(
        {"DocName": "Document 1"},
        {"$rename": {"Date.DoW": "Date.DayOfWeek"}}
    )

    updated_doc = collection.find_one({"DocName": "Document 1"})
    print(updated_doc)

    client.close()

example()
```

$set

The `$set` operator in Amazon DocumentDB is used to update the value of a specified field in a document. This operator allows you to add new fields or modify existing ones within a document. It is a fundamental update operator in the MongoDB Java driver, which is compatible with Amazon DocumentDB.

Parameters

- `field`: The field to update.
- `value`: The new value for the field.

Example (MongoDB Shell)

The following example demonstrates how to use the `$set` operator to update the `Item` field in a document.

Create sample documents

```
db.example.insert([
  {
    "Item": "Pen",
    "Colors": ["Red", "Green", "Blue", "Black"],
    "Inventory": {
      "OnHand": 244,
      "MinOnHand": 72
    }
  },
  {
    "Item": "Poster Paint",
    "Colors": ["Red", "Green", "Blue", "White"],
    "Inventory": {
      "OnHand": 120,
      "MinOnHand": 36
    }
  }
])
```

Query example

```
db.example.update(
  { "Item": "Pen" },
  { $set: { "Item": "Gel Pen" } }
)
```

Output

```
{
```

```
  "Item": "Gel Pen",
  "Colors": ["Red", "Green", "Blue", "Black"],
  "Inventory": {
    "OnHand": 244,
    "MinOnHand": 72
  }
}
```

Code examples

To view a code example for using the `$set` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('example');

  await collection.updateOne(
    { "Item": "Pen" },
    { $set: { "Item": "Gel Pen" } }
  );

  const updatedDocument = await collection.findOne({ "Item": "Gel Pen" });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
```

```
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.example

    collection.update_one(
        {"Item": "Pen"},
        {"$set": {"Item": "Gel Pen"}}
    )

    updated_document = collection.find_one({"Item": "Gel Pen"})
    print(updated_document)

    client.close()

update_document()
```

$setOnInsert

The `$setOnInsert` operator in Amazon DocumentDB is used to set the value of a field if a document is being inserted, but it has no effect if the document is being updated.

Parameters

- `field`: The field to be set.
- `value`: The value to assign to the field.

Example (MongoDB Shell)

The following example demonstrates the usage of the `$setOnInsert` operator in Amazon DocumentDB. It creates a new document if the document does not already exist, but it has no effect if the document is being updated.

Create sample documents

```
db.users.insertOne({
  _id: 1,
  name: "John Doe",
  age: 30
```

```
})
```

Query example 1 - Update existing document

```
db.users.update(
  { _id: 1 },
  {
    $set: { age: 31 },
    $setOnInsert: { createdAt: new Date() }
  },
  { upsert: true }
)
```

Output 1

```
{
  _id: 1,
  name: "John Doe",
  age: 31
}
```

The output shows that the document was updated, but the `createdAt` field was **NOT added** since the document already existed. The `$setOnInsert` operator only applies when inserting new documents.

Query example 2 - Insert new document (upsert)

```
db.users.update(
  { _id: 2 },
  {
    $set: { name: "Jane Smith", age: 25 },
    $setOnInsert: { createdAt: new Date() }
  },
  { upsert: true }
)
```

Output 2

```
{
  _id: 2,
```

```
  name: "Jane Smith",
  age: 25,
  createdAt: ISODate("2025-10-31T09:57:52.459Z")
}
}
```

Code examples

To view a code example for using the `$setOnInsert` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateWithSetOnInsert() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const users = db.collection('users');

  await users.updateOne(
    { _id: 1 },
    {
      $set: { age: 31 },
      $setOnInsert: { createdAt: new Date() }
    },
    { upsert: true }
  );

  const updatedUser = await users.findOne({ _id: 1 });
  console.log(updatedUser);

  await client.close();
}

updateWithSetOnInsert();
```

Python

```
from pymongo import MongoClient
```

```
def update_with_set_on_insert():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    users = db['users']

    result = users.update_one(
        {'_id': 1},
        {
            '$set': {'age': 31},
            '$setOnInsert': {'createdAt': datetime.datetime.now()}
        },
        upsert=True
    )

    updated_user = users.find_one({'_id': 1})
    print(updated_user)

    client.close()

update_with_set_on_insert()
```

$slice

The `$slice` update operator modifies an array by limiting its size. When used with the `$push` operator, it restricts the number of elements in an array, keeping only the specified number of most recent or oldest elements.

Parameters

- `field`: The array field to modify.
- `count`: Maximum number of elements to keep. Positive values keep the first N elements, negative values keep the last N elements.

Example (MongoDB Shell)

The following example demonstrates how to use the `$slice` update operator with `$push` to maintain a fixed-size array of recent scores.

Create sample documents

```
db.students.insertOne({
  _id: 1,
  name: "Alice",
  scores: [85, 90, 78]
});
```

Query example

```
db.students.updateOne(
  { _id: 1 },
  {
    $push: {
      scores: {
        $each: [92, 88],
        $slice: -3
      }
    }
  }
)
```

Output

```
{
  "_id" : 1,
  "name" : "Alice",
  "scores" : [ 78, 92, 88 ]
}
```

In this example, the `$slice: -3` modifier keeps only the last three elements after pushing new values to the array.

Code examples

To view a code example for using the `$slice` update operator, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
```

```
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('students');

  await collection.updateOne(
    { _id: 1 },
    {
      $push: {
        scores: {
          $each: [92, 88],
          $slice: -3
        }
      }
    }
  );

  const updatedDocument = await collection.findOne({ _id: 1 });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.students

    collection.update_one(
        {'_id': 1},
        {
            '$push': {
                'scores': {
                    '$each': [92, 88],
```

```
                    '$slice': -3
                }
            }
        }
    )

    updated_document = collection.find_one({'_id': 1})
    print(updated_document)

    client.close()

update_document()
```

$sort

The `$sort` update modifier orders array elements when used with the `$push` operator. It arranges array elements in ascending or descending order based on specified field values or the elements themselves.

Parameters

- `field`: The array field to modify.
- `order`: Use `1` for ascending order or `-1` for descending order.

Example (MongoDB Shell)

The following example demonstrates using the `$sort` modifier with `$push` to add new quiz scores and keep them sorted in descending order.

Create sample documents

```
db.students.insertOne({
  _id: 1,
  name: "Bob",
  quizzes: [
    { score: 85, date: "2024-01-15" },
    { score: 92, date: "2024-02-10" }
  ]
});
```

Query example

```
db.students.updateOne(
  { _id: 1 },
  {
    $push: {
      quizzes: {
        $each: [{ score: 78, date: "2024-03-05" }],
        $sort: { score: -1 }
      }
    }
  }
)
```

Output

```
{
  "_id" : 1,
  "name" : "Bob",
  "quizzes" : [
    { "score" : 92, "date" : "2024-02-10" },
    { "score" : 85, "date" : "2024-01-15" },
    { "score" : 78, "date" : "2024-03-05" }
  ]
}
```

Code examples

To view a code example for using the `$sort` update modifier, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function updateDocument() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('students');

  await collection.updateOne(
    { _id: 1 },
```

```
    {
      $push: {
        quizzes: {
          $each: [{ score: 78, date: "2024-03-05" }],
          $sort: { score: -1 }
        }
      }
    }
  );

  const updatedDocument = await collection.findOne({ _id: 1 });
  console.log(updatedDocument);

  await client.close();
}

updateDocument();
```

Python

```
from pymongo import MongoClient

def update_document():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client.test
    collection = db.students

    collection.update_one(
        {'_id': 1},
        {
            '$push': {
                'quizzes': {
                    '$each': [{'score': 78, 'date': '2024-03-05'}],
                    '$sort': {'score': -1}
                }
            }
        }
    )

    updated_document = collection.find_one({'_id': 1})
    print(updated_document)
```

```
    client.close()

update_document()
```

$unset

The `$unset` operator in Amazon DocumentDB is used to remove a specified field from a document. When a field is removed using `$unset`, the field is deleted from the document, and the document size is reduced accordingly. This can be useful when you want to remove unnecessary data from your documents.

Parameters

- `field`: The field to remove from the document. This can be a single field or a dotted path to a nested field.

Example (MongoDB Shell)

The following example demonstrates how to use the `$unset` operator to remove the `Words` field from a document in the `example` collection.

Create sample documents

```
db.example.insert({
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
        "DoW": "Saturday"
    },
    "Words": 2482
})
```

Query example

```
db.example.update(
    { "DocName" : "Document 1" },
    { $unset: { Words:1 } }
```

```
)
```

Output

```
{
    "DocName": "Document 1",
    "Date": {
        "Month": 4,
        "Day": 18,
        "Year": 1987,
        "DoW": "Saturday"
    }
}
```

In this example, the `$unset` operator is used to remove the `Words` field from the document with `DocName` equal to "Document 1". The resulting document no longer contains the `Words` field.

Code examples

To view a code example for using the `$unset` command, choose the tab for the language that you want to use:

Node.js

```
const { MongoClient } = require('mongodb');

async function removeField() {
  const client = await MongoClient.connect('mongodb://
<username>:<password>@<cluster-endpoint>:27017/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false');
  const db = client.db('test');
  const collection = db.collection('example');

  const result = await collection.updateOne(
    { "DocName": "Document 1" },
    { $unset: { "Words": 1 } }
  );

  console.log(`Modified ${result.modifiedCount} document(s)`);
  client.close();
}
```

```
removeField();
```

Python

```
from pymongo import MongoClient

def remove_field():
    client = MongoClient('mongodb://<username>:<password>@<cluster-endpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
    db = client['test']
    collection = db['example']

    result = collection.update_one(
        {"DocName": "Document 1"},
        {"$unset": {"Words": 1}}
    )

    print(f"Modified {result.modified_count} document(s)")
    client.close()

remove_field()
```

Managing Amazon DocumentDB clusters and instances

These sections cover the various components and their related tasks for managing your Amazon DocumentDB (with MongoDB compatibility) implementation.

Topics

- Amazon DocumentDB operational tasks overview
- Overview of Amazon DocumentDB global clusters
- Managing Amazon DocumentDB clusters
- Managing Amazon DocumentDB instances
- Managing Amazon DocumentDB subnet groups
- Amazon DocumentDB high availability and replication
- Managing Amazon DocumentDB events
- Managing Amazon DocumentDB cluster parameter groups
- Understanding Amazon DocumentDB endpoints
- Understanding Amazon DocumentDB Amazon Resource Names (ARNs)
- Tagging Amazon DocumentDB resources
- Maintaining Amazon DocumentDB
- Understanding service-linked roles

Amazon DocumentDB operational tasks overview

This section covers operational tasks for your Amazon DocumentDB cluster, and how to accomplish these tasks using the AWS CLI.

Topics

- Adding a replica to an Amazon DocumentDB cluster
- Describing clusters and instances
- Creating a cluster snapshot
- Restoring from a snapshot
- Removing an instance from a cluster

- Deleting a cluster

Adding a replica to an Amazon DocumentDB cluster

After you create the primary instance for your Amazon DocumentDB cluster, you can add one or more *replicas*. A replica is a read-only instance that serves two purposes:

- **Scalability** — If you have a large number of clients that require concurrent access, you can add more replicas for read-scaling.
- **High availability** — If the primary instance fails, Amazon DocumentDB automatically fails over to a replica instance and designates it as the new primary. If a replica fails, other instances in the cluster can still serve requests until the failed node can be recovered.

Each Amazon DocumentDB cluster can support up to 15 replicas.

> **Note**
>
> For maximum fault tolerance, you should deploy replicas in separate Availability Zones. This helps ensure that your Amazon DocumentDB cluster can continue to function, even if an entire Availability Zone becomes unavailable.

The following AWS CLI example shows how to add a new replica. The `--availability-zone` parameter places the replica in the specified Availability Zone.

```
aws docdb create-db-instance \
    --db-instance-identifier sample-instance \
    --db-cluster-identifier sample-cluster \
    --engine docdb \
    --db-instance-class db.r5.large \
    --availability-zone us-east-1a
```

Describing clusters and instances

The following AWS CLI example lists all Amazon DocumentDB clusters in a Region. For certain management features such as cluster and instance lifecycle management, Amazon

DocumentDB leverages operational technology that is shared with Amazon RDS. The `filterName=engine,Values=docdb` filter parameter returns only Amazon DocumentDB clusters.

For more information on describing and modifying clusters, see the Amazon DocumentDB cluster lifecycle.

```
aws docdb describe-db-clusters --filter Name=engine,Values=docdb
```

Output from this operation looks something like the following.

```
{
    "DBClusters": [
        {
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-1",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        },
        {
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-2",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        },
        {
            "AvailabilityZones": [
                "us-east-1c",
```

```
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-3",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        }
    ]
}
```

The following AWS CLI example lists the instances in an Amazon DocumentDB cluster. For more information on describing and modifying clusters, see the Amazon DocumentDB instance lifecycle.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].[DBClusterMembers]'
```

The output looks like something like below. In this output, there are two instances. The primary instance is `sample-instance-1` (`"IsClusterWriter": true`). There is also a replica instance, `sample-instance2` (`"IsClusterWriter: false"`).

```
[
    [
        [
            {
                "DBInstanceIdentifier": "sample-instance-1",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            },
            {
                "DBInstanceIdentifier": "sample-cluster-2",
                "IsClusterWriter": false,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            }
        ]
    ]
]
```

Creating a cluster snapshot

A *cluster snapshot* is a complete backup of the data in your Amazon DocumentDB cluster. When the snapshot is being created, Amazon DocumentDB reads your data directly from the cluster volume. Because of this, you can create a snapshot even if your cluster doesn't have any instances running at the time. The amount of time it takes to create a snapshot depends on the size of your cluster volume.

Amazon DocumentDB supports automatic backups, which occur daily during the preferred backup window — a 30-minute period of time during the day. The following AWS CLI example shows how to view the backup window for your cluster:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].PreferredBackupWindow'
```

The output shows the backup window (in UTC):

```
[
    "00:18-00:48"
]
```

You can define the backup window when you create your Amazon DocumentDB cluster. You can also change the backup window, as shown in the following example. If you don't define a backup window, Amazon DocumentDB automatically assigns one to your cluster.

```
aws docdb modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --preferred-backup-window "02:00-02:30"
```

In addition to automatic backups, you can manually create a cluster snapshot at any time. When you do this, you specify which cluster you want to back up, and a unique name for your snapshot so that you can restore from it later.

The following AWS CLI example shows how to create a snapshot of your data.

```
aws docdb create-db-cluster-snapshot \
    --db-cluster-identifier sample-cluster \
```

```
    --db-cluster-snapshot-identifier sample-cluster-snapshot
```

Restoring from a snapshot

You can restore a cluster snapshot to a new Amazon DocumentDB cluster. To do this, you provide the name of the snapshot and the name of a new cluster. You can't restore from a snapshot to an existing cluster; instead, Amazon DocumentDB creates a new cluster when you restore and then populates it with your snapshot data.

The following example shows all the snapshots for the cluster `sample-cluster`.

```
aws docdb describe-db-cluster-snapshots \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusterSnapshots[*].[DBClusterSnapshotIdentifier,SnapshotType,Status]'
```

The output looks something like the following. A manual snapshot is one that you created manually, whereas an automated snapshot is created by Amazon DocumentDB within the cluster backup window.

```
[
        "sample-cluster-snapshot",
        "manual",
        "available"
    ],
    [
        "rds:sample-cluster",
        "automated",
        "available"
    ]
]
```

The following example shows how to restore an Amazon DocumentDB cluster from a snapshot.

```
aws docdb restore-db-cluster-from-snapshot \
    --engine docdb \
    --db-cluster-identifier new-sample-cluster \
    --snapshot-identifier sample-cluster-snapshot
```

The new cluster does not have any instances associated with it; so if you want to interact with the cluster, you must add an instance to it.

```
aws docdb create-db-instance \
    --db-instance-identifier new-sample-instance \
    --db-instance-class db.r5.large \
    --engine docdb \
    --db-cluster-identifier new-sample-cluster
```

You can use the following AWS CLI operations to monitor the progress of cluster and instance creation. When the cluster and instance statuses are available, you can connect to the new cluster's endpoint and access your data.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier new-sample-cluster  \
    --query 'DBClusters[*].[Status,Endpoint]'
```

```
aws docdb describe-db-instances \
    --db-instance-identifier new-sample-instance \
    --query 'DBInstances[*].[DBInstanceStatus]'
```

Removing an instance from a cluster

Amazon DocumentDB stores all of your data in the cluster volume. The data persists in that cluster volume, even if you remove all the instances from your cluster. If you need to access the data again, you can add an instance to the cluster at any time, and pick up where you left off.

The following example shows how to remove an instance from your Amazon DocumentDB cluster.

```
aws docdb delete-db-instance \
    --db-instance-identifier sample-instance
```

Deleting a cluster

Before you can delete an Amazon DocumentDB cluster, you must first remove all of its instances. The following AWS CLI example returns information about the instances in a cluster. If this operation returns any instance identifiers, you have to delete each of the instances. For more information, see Removing an instance from a cluster.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
```

```
    --query 'DBClusters[*].DBClusterMembers[*].DBInstanceIdentifier'
```

When there are no more instances remaining, you can delete the cluster. At that time, you must choose one of the following options:

- **Create a final snapshot** — Capture all the cluster data in a snapshot so that you can re-create a new instance with that data later. The following example shows how to do this:

```
aws docdb delete-db-cluster \
    --db-cluster-identifier sample-cluster \
    --final-db-snapshot-identifier sample-cluster-snapshot
```

- **Skip the final snapshot** — Permanently discard all the cluster data. This cannot be reversed. The following example shows how to do this:

```
aws docdb delete-db-cluster \
    --db-cluster-identifier sample-cluster \
    --skip-final-snapshot
```

Overview of Amazon DocumentDB global clusters

What is a global cluster?

A global cluster consists of one primary Region and up to 10 read-only secondary Regions. You issue write operations directly to the primary cluster in the primary Region and Amazon DocumentDB automatically replicates the data to the secondary Regions using dedicated infrastructure. Latency is typically under a second.

How are global clusters useful?

- **Recovery from Region-wide outages** — In the event of a Region-wide outage, you can promote one of the secondary clusters to a primary cluster within minutes, with a typical Recovery Time Objective (RTO) of under a minute. The Recovery Point Objective (RPO) is typically measured in seconds, but this depends on the lag across the network at the time of the failure.
- **Global reads with local latency** — If you have offices around the world, you can use a global cluster to keep your main sources of information updated in the primary Region. Offices in your other Regions can access the information in their own Region, with local latency.

- **Scalable secondary clusters** — You can scale your secondary clusters by adding more read-only instances to a secondary Region. The secondary cluster is read-only, so it can support up to 16 read-only replica instances rather than the usual limit of 15 for a single cluster.
- **Fast replication from primary to secondary clusters** — The replication performed by a global cluster has little performance impact on the primary database cluster. The resources of the DB instances are fully devoted to serve application read and write workloads.
- **Change streams** — You can read change streams from a secondary cluster in global clusters, reducing the load on the primary cluster and allowing you to scale read workloads from secondary Regions.

What are the current limitations of global clusters?

- Global clusters are not supported on Amazon DocumentDB v3.6.
- Global clusters are supported on all instance types except db.t3, db.t4g, and db.r4.
- Global clusters are not available in the following Regions: South America (São Paulo), Europe (Milan), China (Beijing), and China (Ningxia).
- Switchover and global failover are not supported when Regions are on different engine versions. Manual failover is supported when there's an engine version mismatch.
- Only the primary cluster performs write operations. Clients that perform write operations connect to the cluster endpoint of the primary cluster.
- You can have a maximum of 10 secondary Regions and one primary Region for your cluster.
- A secondary cluster cannot be stopped. A primary cluster cannot be stopped if it has secondary clusters associated with it. Only a regional cluster that has no secondary clusters can be stopped.
- Replicas attached to the secondary cluster can restart under certain circumstances. If the primary Region's instance restarts or fails over, replicas in the secondary Region also restart. The cluster is then unavailable until all replicas are back in sync with the primary database cluster's writer instance. This behavior is expected. Be sure that you understand the impact to your global cluster before making changes to your primary cluster.

Topics

- Quick start guide: global clusters
- Managing an Amazon DocumentDB global cluster
- Connect to an Amazon DocumentDB global cluster

- Monitoring Amazon DocumentDB global clusters
- Disaster recovery and Amazon DocumentDB global clusters

Quick start guide: global clusters

Topics

- Configuration
- Creating an Amazon DocumentDB global cluster
- Adding an AWS Region to an Amazon DocumentDB global cluster
- Using a snapshot for your Amazon DocumentDB global cluster

Configuration

Amazon DocumentDB global cluster spans at least two AWS Regions. The primary Region supports a cluster that has one primary (writer) instance and up to 15 replica instances, while a secondary Region runs a read-only cluster made up entirely of up to 16 replica instances. A global cluster can have up to 10 secondary Regions. The table lists the maximum clusters, instances, and replicas allowed in a global cluster.

Description	Primary AWS Region	Secondary AWS Region
Clusters	1	10 (maximum)
Writer instances	1	0
Read-only instances (Amazon DocumentDB replicas), per cluster	15 (max)	16 (total)
Read-only instances (max allowed, given actual number of secondary Regions)	15 - *s*	*s* = total number of secondary AWS Regions

The clusters have the following specific requirements:

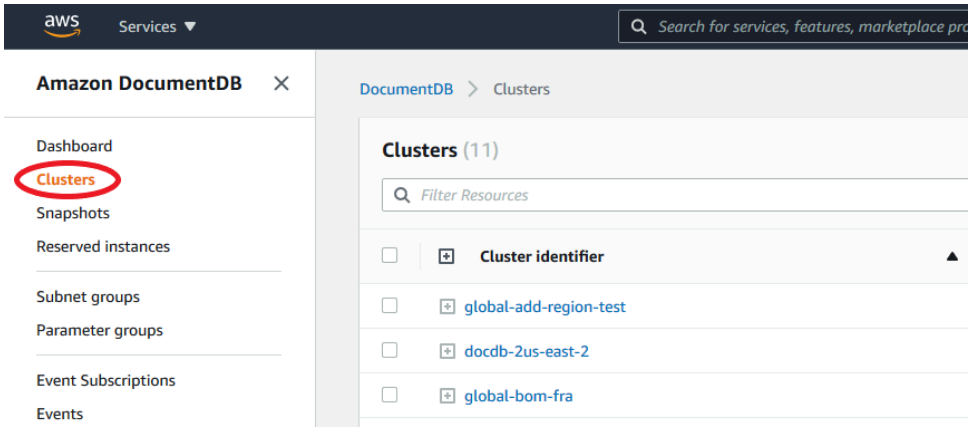
- **Database instance class requirements** — You can only use the `db.r5` and `db.r6g` instance classes.
- **AWS Region requirements** — The primary cluster must be in one Region, and at least one secondary cluster must be in a different Region of the same account. You can create up to 10 secondary (read-only) clusters, and each must be in a different Region. In other words, no two clusters can be in the same Region.
- **Naming requirements** — The names you choose for each of your clusters must be unique, across all Regions. You can't use the same name for different clusters even though they're in different Regions.

Creating an Amazon DocumentDB global cluster

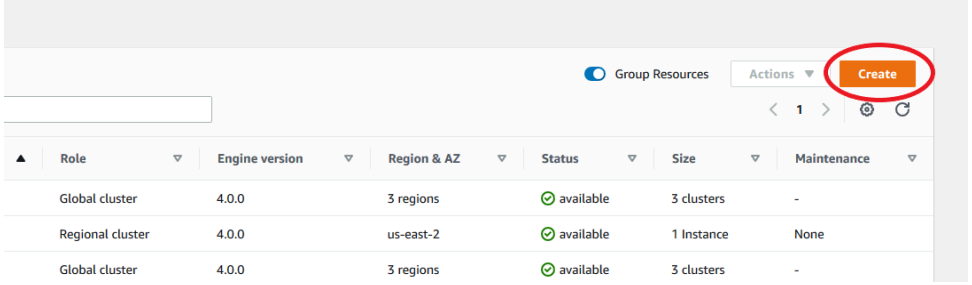
Are you ready to build your first global cluster? In this section we will explain how to create a brand new global cluster with new database clusters and instances, using either the AWS Management Console or AWS CLI with the following instructions.

Using the AWS Management Console

1. In the AWS Management Console, navigate to **Amazon DocumentDB**.
2. When you get to the Amazon DocumentDB console, choose **Clusters**.

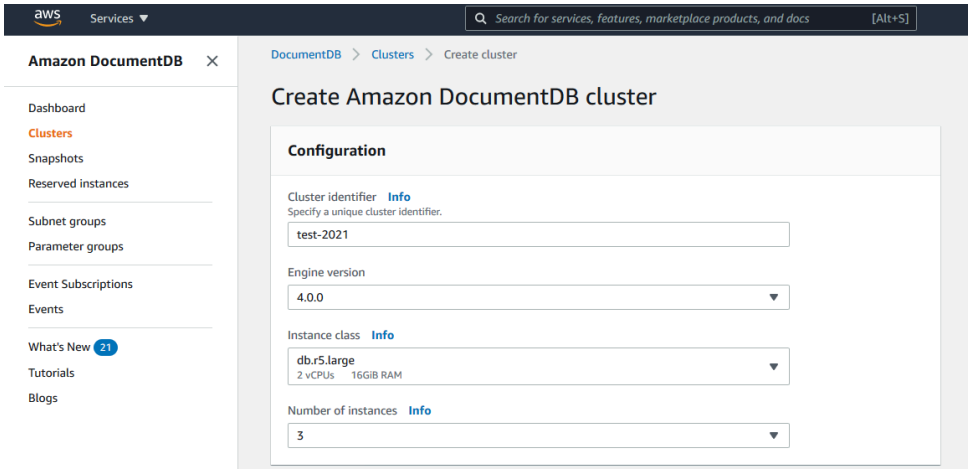


3. Choose **Create**.

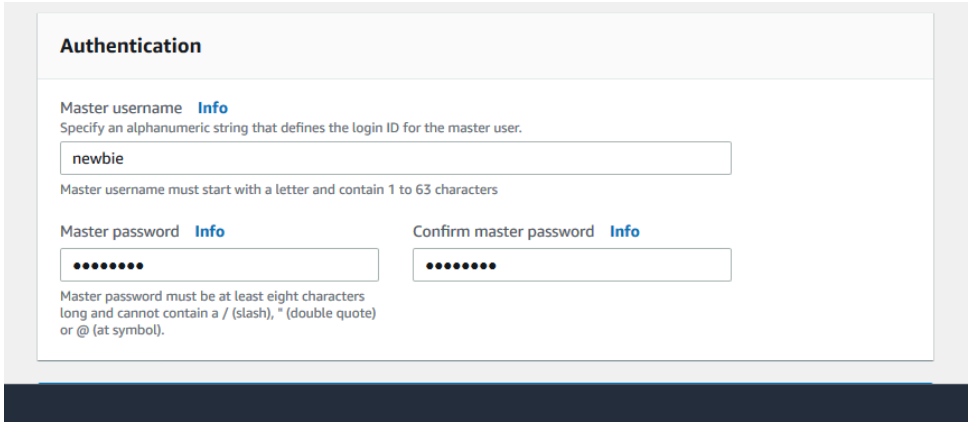


4. Fill out the **Configuration** section of the **Create Amazon DocumentDB Cluster** form accordingly:

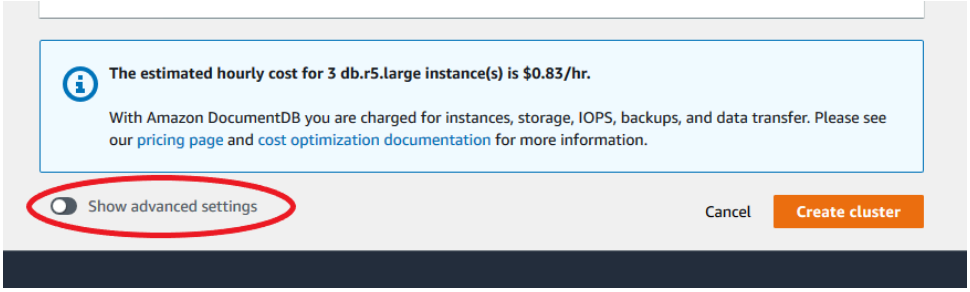
   - **Cluster identifier**: You can either enter a unique identifier for this instance or allow Amazon DocumentDB to provide the instance identifier based on the cluster identifier.
   - Engine version: Choose **4.0.0**
   - Instance class: Choose **db.r5.large**
   - Number of instances: Choose **3**.



5. In the **Authentication** section, fill in a master username and master password.

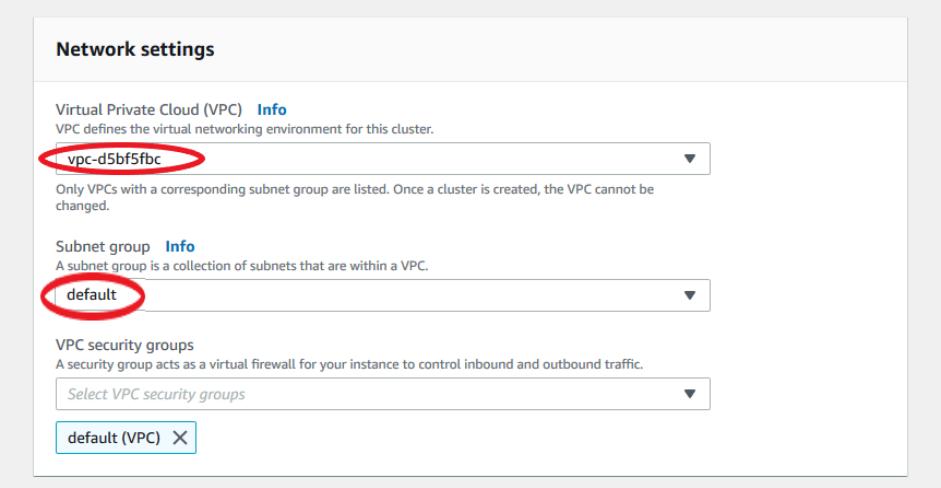


6. Choose **Show advanced settings**.

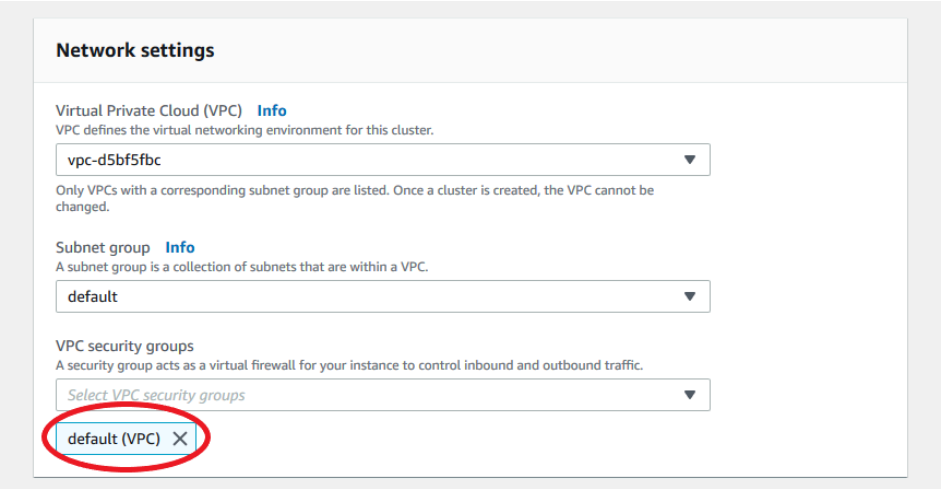


7. In the **Network settings** section:

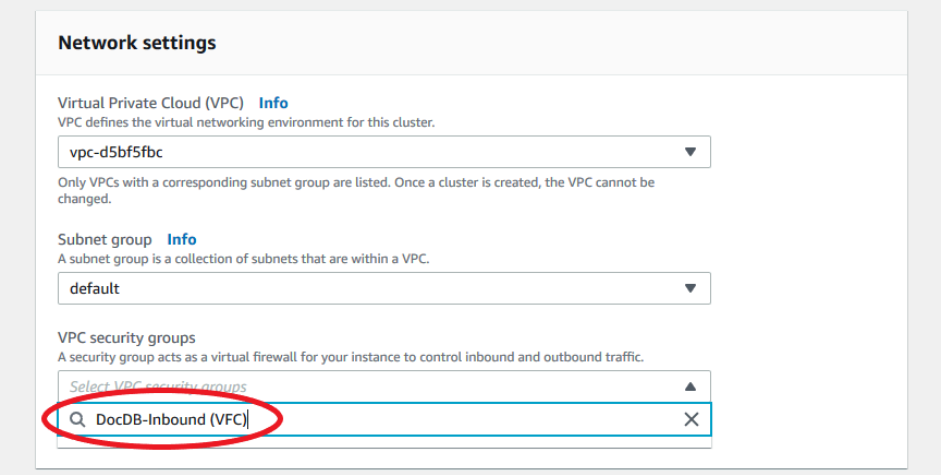
   - Keep default options for **Virtual Private Cloud (VPC)** and **Subnet group**.



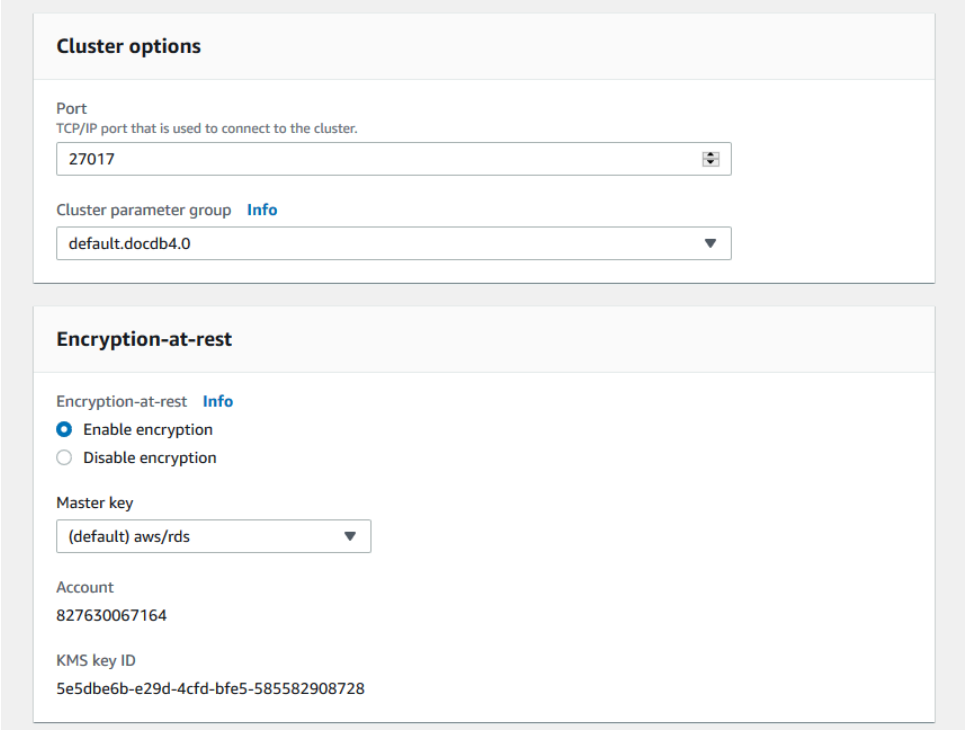
   - For **VPC security groups**, **default (VPC)** should already be added.



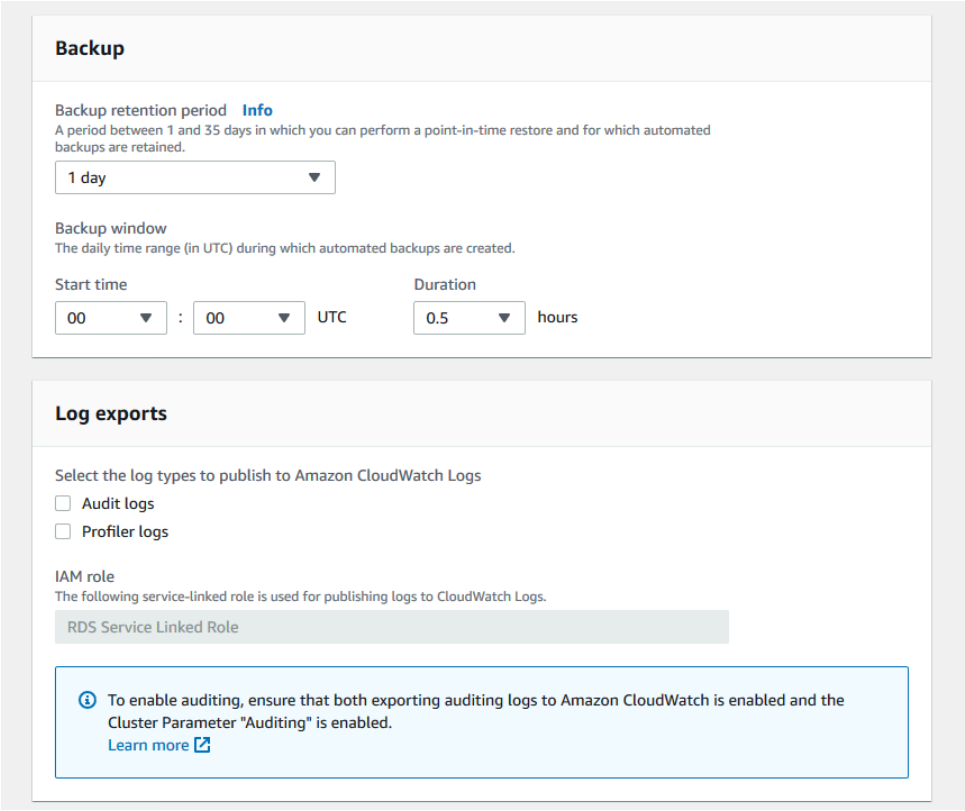
   - Type DocDB into the **VPC security groups** field and select **DocDB-Inbound (VFC)**.



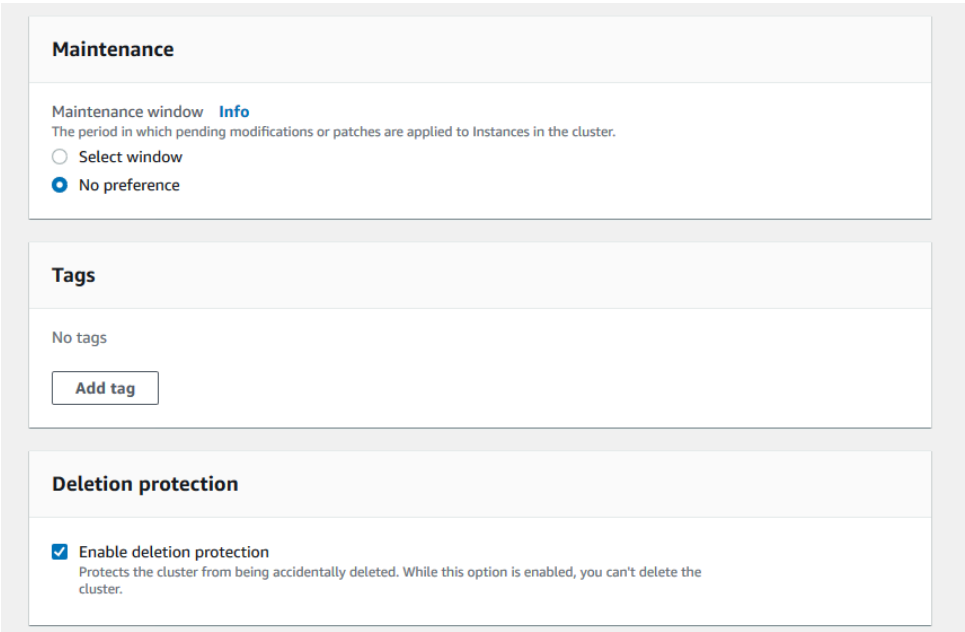
8. For **Cluster options** and **Encryption-at-rest**, leave at default selections.



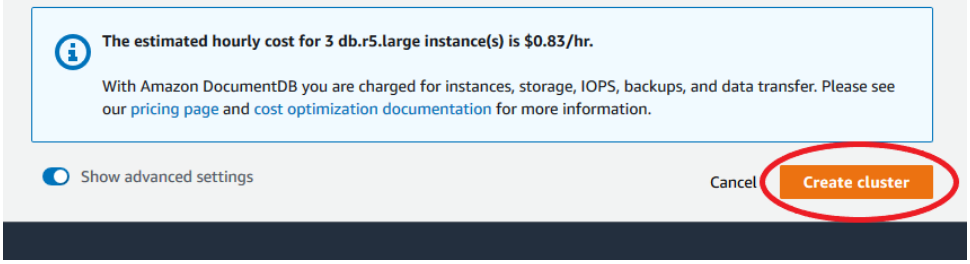
9. For **Backup** and **Log exports**, leave at default selections.



10. For **Maintenance**, **Tags**, and **Deletion protection**, leave at default selections.



11. Now choose **Create cluster**.



Using the AWS CLI

To create an Amazon DocumentDB Regional cluster, call the create-global-cluster AWS CLI. The following AWS CLI command creates an Amazon DocumentDB cluster named `global-cluster-id`. For more information on deletion protection, see Deleting an Amazon DocumentDB cluster.

Also, `--engine-version` is an optional parameter. If you omit it, Amazon DocumentDB uses the newest 8.0 minor version available in the AWS Region. To use a specific minor version, specify it with `--engine-version`. For more information, see Amazon DocumentDB cluster settings.

If a `db-subnet-group-name` or `vpc-security-group-id` is not specified, Amazon DocumentDB will use the default subnet group and Amazon VPC security group for the given Region.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
      --global-cluster-identifier global-cluster-id \
      --source-db-cluster-identifier arn:aws:rds:us-east-1:111122223333:cluster-id
```

For Windows:

```
aws docdb create-db-cluster ^
      --global-cluster-identifier global-cluster-id ^
      --source-db-cluster-identifier arn:aws:rds:us-east-1:111122223333:cluster-id
```

Output from this operation looks something like the following (JSON format).

```
{
```

```
    "DBCluster": {
        "StorageEncrypted": false,
        "DBClusterMembers": [],
        "Engine": "docdb",
        "DeletionProtection" : "enabled",
        "ClusterCreateTime": "2018-11-26T17:15:19.885Z",
        "DBSubnetGroup": "default",
        "EngineVersion": "4.0.0",
        "MasterUsername": "masteruser",
        "BackupRetentionPeriod": 1,
        "DBClusterArn": "arn:aws:rds:us-east-1:123456789012:cluster:cluster-id",
        "DBClusterIdentifier": "cluster-id",
        "MultiAZ": false,
        "DBClusterParameterGroup": "default.docdb4.0",
        "PreferredBackupWindow": "09:12-09:42",
        "DbClusterResourceId": "cluster-KQSGI4MHU4NTDDRVNLNTU7XVAY",
        "PreferredMaintenanceWindow": "tue:04:17-tue:04:47",
        "Port": 27017,
        "Status": "creating",
        "ReaderEndpoint": "cluster-id.cluster-ro-sfcrlcjcoroz.us-
east-1.docdb.amazonaws.com",
        "AssociatedRoles": [],
        "HostedZoneId": "ZNKXTT8WH85VW",
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-77186e0d",
                "Status": "active"
            }
        ],
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1c",
            "us-east-1e"
        ],
        "Endpoint": "cluster-id.cluster-sfcrlcjcoroz.us-east-1.docdb.amazonaws.com"
    }
}
```


It takes several minutes to create the cluster. You can use the AWS Management Console or AWS CLI to monitor the status of your cluster. For more information, see Monitoring an Amazon DocumentDB cluster's status.

> **Important**
>
> When you use the AWS CLI to create an Amazon DocumentDB Regional cluster, no instances are created. Consequently, you must explicitly create a primary instance and any replica instances that you need. You can use either the console or AWS CLI to create the instances. For more information, see Adding an Amazon DocumentDB instance to a cluster and CreateDBCluster in the *Amazon DocumentDB API Reference*.

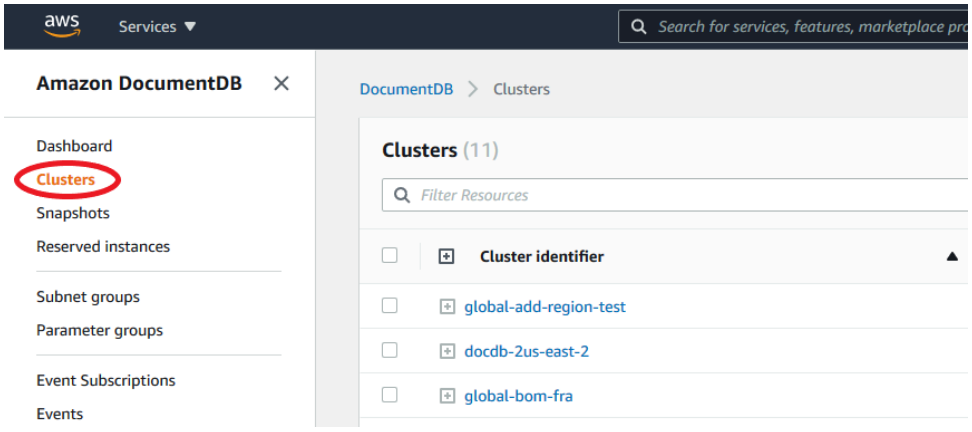
Once your Regional cluster is available, you can add a secondary cluster in another Region with the following instructions: Adding an AWS Region to an Amazon DocumentDB global cluster. When you add a Region, your Regional cluster becomes your primary cluster, and you have a new secondary cluster in the Region you chose.

Adding an AWS Region to an Amazon DocumentDB global cluster

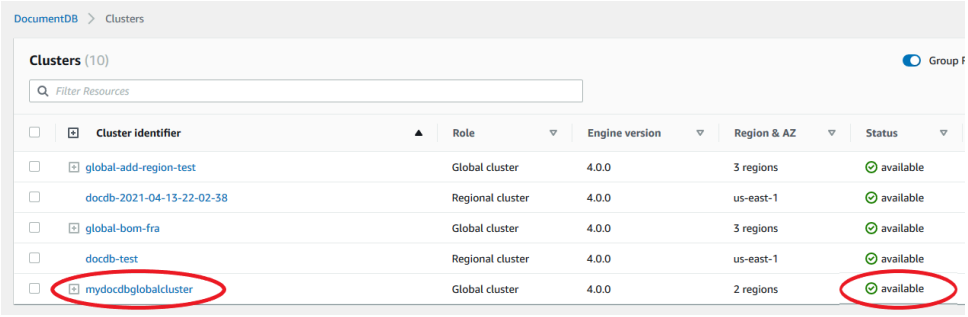
A global cluster needs at least one secondary cluster in a different Region than the primary cluster, and you can add up to 10 secondary clusters. Note that for each secondary cluster that you add, you must reduce the number of replicas allowed in the primary cluster by one. For example, if your global cluster has 10 secondary Regions, your primary cluster can have only 5 (rather than 15) replicas. For more information, see Configuration.

Using the AWS Management Console

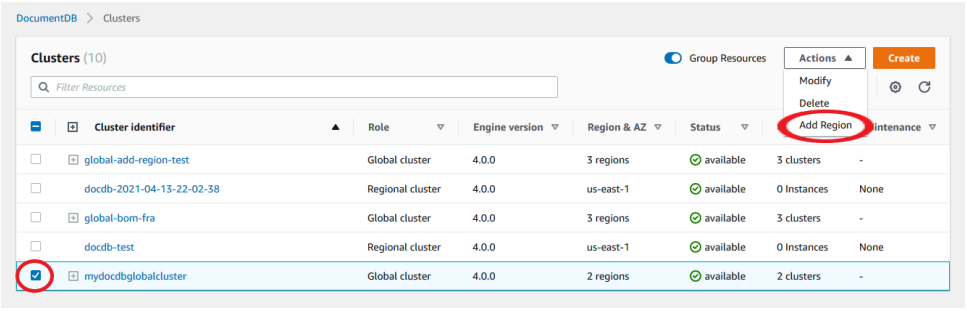
1. Sign in to the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.



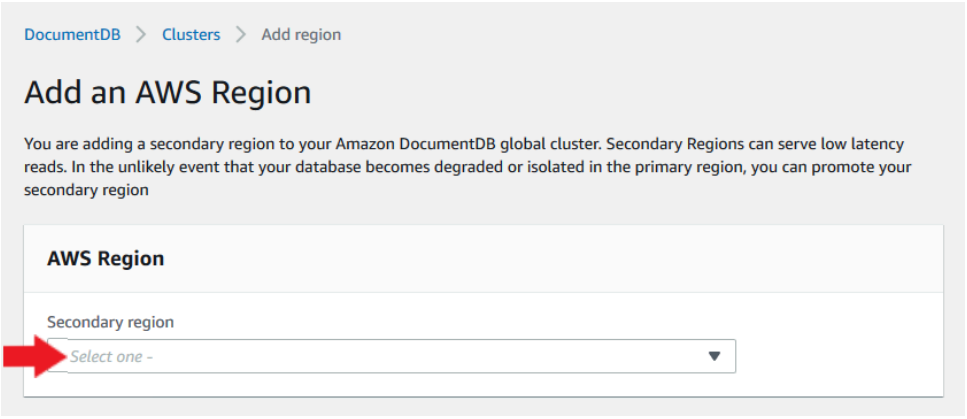
3. Choose the cluster that you would like to add a secondary cluster to. Ensure that the cluster is `Available`.



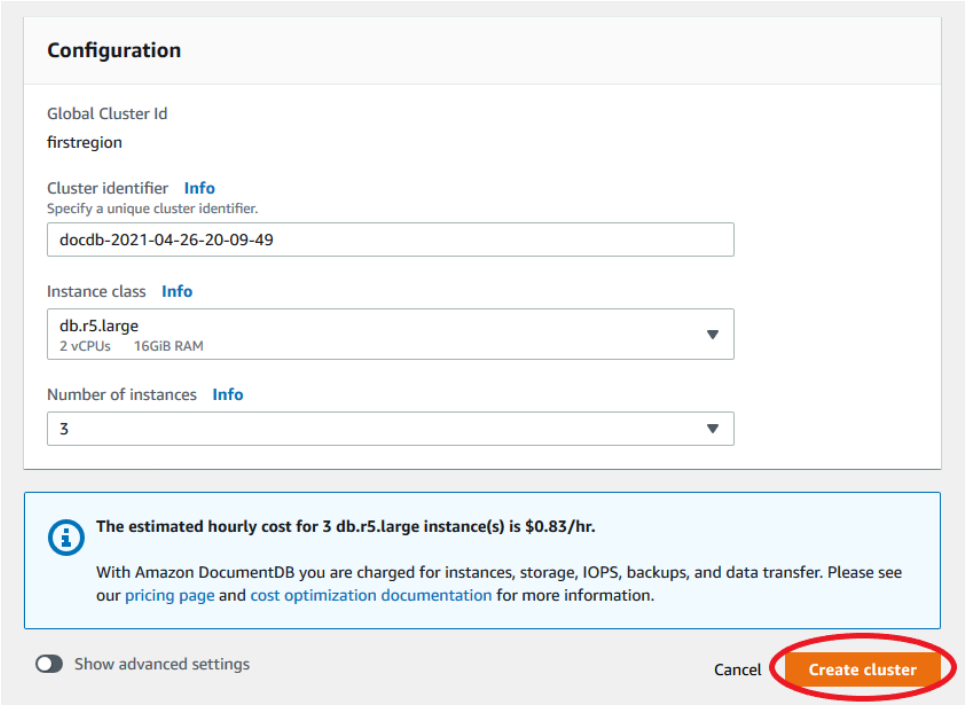
4. Select the dropdown list for **Actions** and then choose **Add Region**.



5. On the **Add an AWS Region** page, choose the secondary Region. Note that you can't choose a Region that already has a secondary cluster for the same global cluster. Also, it can't be the same Region as the primary cluster. If this is the first Region you are adding, you will also have to specify a global cluster identifier of your choice.



6. Complete the remaining fields for the secondary cluster in the new Region, then select **Create cluster**. After you finish adding the Region, you can see it in the list of **Clusters** in the AWS Management Console.



Using the AWS CLI

- Use the `create-db-cluster` CLI command with the name (`--global-cluster-identifier`) of your global cluster. For other parameters, do the following:
  - For `--region`, choose a different AWS Region than that of your primary Region.
  - Choose specific values for the `--engine` and `--engine-version` parameters.
  - For an encrypted cluster, specify your primary AWS Region as the `--source-region` for encryption.

The following example creates a new Amazon DocumentDB cluster and attaches it to the global cluster as a read-only secondary cluster. In the last step, the instance is added to the new cluster.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb --region secondary-region-id \
  create-db-cluster \
    --db-cluster-identifier cluster-id \
    --global-cluster-identifier global-cluster-id \
    --engine-version version \
```

```
    --engine docdb

aws docdb --region secondary-region-id \
  create-db-instance \
    --db-cluster-identifier cluster-id \
    --global-cluster-identifier global-cluster-id \
    --engine-version version \
    --engine docdb
```

For Windows:

```
aws docdb --region secondary-region-id ^
  create-db-cluster ^
    --db-cluster-identifier cluster-id ^
    --global-cluster-identifier global-cluster-id ^
    --engine-version version ^
    --engine docdb

aws docdb --region secondary-region-id ^
  create-db-instance ^
    --db-cluster-identifier cluster-id ^
    --global-cluster-identifier global-cluster-id ^
    --engine-version version ^
    --engine docdb
```

Using a snapshot for your Amazon DocumentDB global cluster

You can restore a snapshot of an Amazon DocumentDB cluster to use as the starting point for your global cluster. To do this, you must restore the snapshot and create a new cluster. This will serve as the primary cluster of your global cluster. You can then add another Region to the restored cluster, thus converting it into a global cluster.

Managing an Amazon DocumentDB global cluster

You perform most management operations on the individual clusters that make up a global cluster. When you choose **Group related resources** on the **Clusters** page in the console, you see the primary cluster and secondary clusters grouped under the associated global cluster.

The **Configuration** tab for a global cluster shows the AWS Regions where the clusters are running, the version, and the global cluster identifier.

Topics

- Modifying an Amazon DocumentDB global cluster
- Modifying parameters an Amazon DocumentDB global cluster
- Removing a cluster from an Amazon DocumentDB global cluster
- Deleting a cluster from an Amazon DocumentDB global cluster
- Creating a headless Amazon DocumentDB cluster in a secondary Region

Modifying an Amazon DocumentDB global cluster

The **Clusters** page in the AWS Management Console lists all your global clusters, showing the primary cluster and secondary clusters for each one. The global cluster has its own configuration settings. Specifically, it has Regions associated with its primary and secondary clusters.

When you make changes to the global cluster, you have a chance to cancel changes.

When you choose Continue, you confirm the changes.

Modifying parameters an Amazon DocumentDB global cluster

You can configure the cluster parameter groups independently for each cluster within the global cluster. Most parameters work the same as for other kinds of Amazon DocumentDB clusters. Keep settings consistent among all the clusters in a global database to avoid unexpected behavior changes if you promote a secondary cluster to be the primary.

For example, use the same settings for time zones and character sets to avoid inconsistent behavior if a different cluster takes over as the primary cluster.

Removing a cluster from an Amazon DocumentDB global cluster

There are several situations when you may want to remove clusters from your global cluster. For example, you might want to remove a cluster from a global cluster if the primary cluster becomes degraded or isolated. It then becomes a standalone provisioned cluster that could be used to create a new global cluster. To learn more, see Performing a manual failover for an Amazon DocumentDB global cluster.

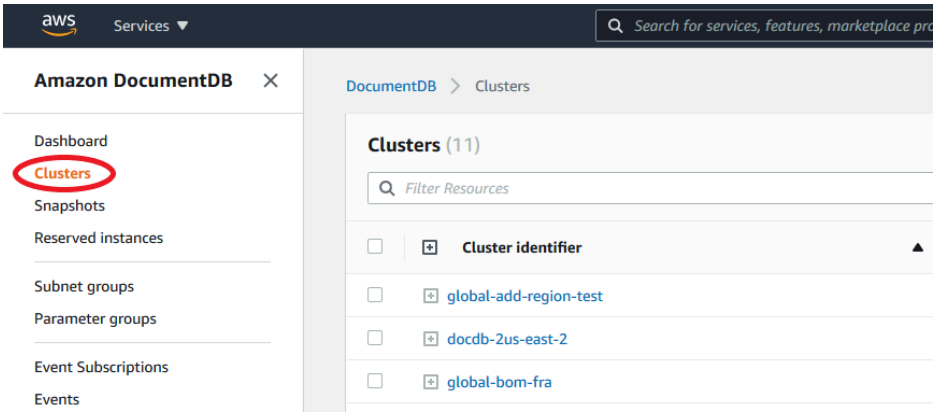
You also might want to remove clusters because you want to delete a global cluster that you no longer need. You can't delete the global cluster until after you detach all associated clusters, leaving the primary for last. For more information, see Deleting a cluster from an Amazon DocumentDB global cluster.

> **Note**
>
> When a cluster is detached from the global cluster, it's no longer synchronized with the primary. It becomes a standalone provisioned cluster with full read/write capabilities. Additionally, it is no longer visible in the Amazon DocumentDB console. It is only visible when you select the Region in the console that the cluster was located in.

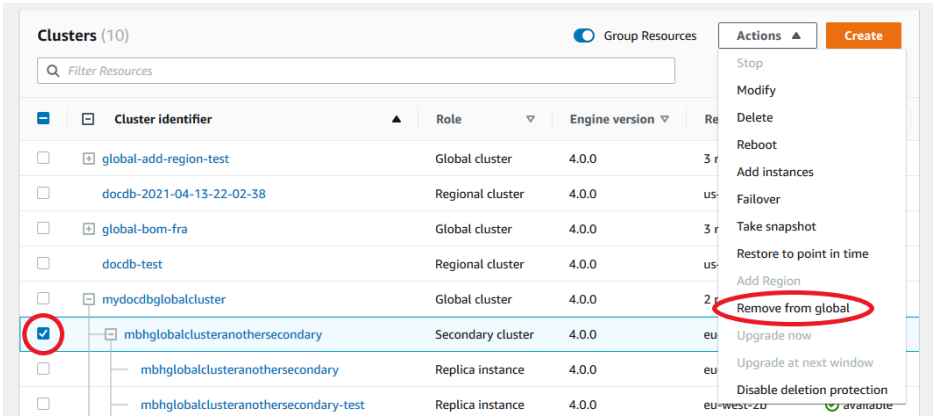
You can remove clusters from your global cluster using the AWS Management Console, the AWS CLI, or the RDS API.

Using the AWS Management Console

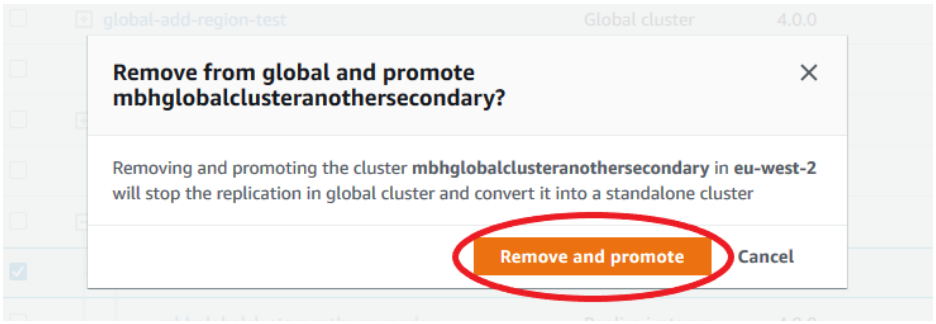
1. Sign in to the AWS Management Console and navigate to the Amazon DocumentDB console.
2. Choose **Clusters** on the left side navigation.



3. Expand the global cluster so you can see all the secondary clusters. Select the secondary clusters you wish to remove. Choose **Actions**, and in the menu that drops down, choose **Remove from Global**.



4. A prompt will appear, asking you to confirm that you want to detach the secondary from the global cluster. Choose **Remove and promote** to remove the cluster from the global cluster.



Now that cluster is no longer serving as a secondary and no longer synchronized with the primary cluster. It is a standalone cluster with full read/write capability.

After you remove or delete all secondary clusters, then you can remove the primary cluster the same way. You can't detach or remove the primary cluster from the global cluster until after you have removed all secondary clusters. The global cluster might remain in the Clusters list, with zero Regions and AZs. You can delete if you no longer want to use this global cluster.

Using the AWS CLI

To remove a cluster from a global cluster, run the `remove-from-global-cluster` CLI command with the following parameters:

- `--global-cluster-identifier` — The name (identifier) of your global cluster.
- `--db-cluster-identifier` — The name of each cluster to remove from the global cluster.

The following examples first remove a secondary cluster and then the primary cluster from a global cluster.

For Linux, macOS, or Unix:

```
aws docdb --region secondary_region \
  remove-from-global-cluster \
    --db-cluster-identifier secondary_cluster_ARN \
    --global-cluster-identifier global_cluster_id

aws docdb --region primary_region \
  remove-from-global-cluster \
```

```
    --db-cluster-identifier primary_cluster_ARN \
    --global-cluster-identifier global_cluster_id
```

Repeat the `remove-from-global-cluster --db-cluster-identifier secondary_cluster_ARN` command for each secondary Region in your global cluster.

For Windows:

```
aws docdb --region secondary_region ^
  remove-from-global-cluster ^
    --db-cluster-identifier secondary_cluster_ARN ^
    --global-cluster-identifier global_cluster_id

aws docdb --region primary_region ^
  remove-from-global-cluster ^
    --db-cluster-identifier primary_cluster_ARN ^
    --global-cluster-identifier global_cluster_id
```

Repeat the `remove-from-global-cluster --db-cluster-identifier secondary_cluster_ARN` command for each secondary Region in your global cluster.

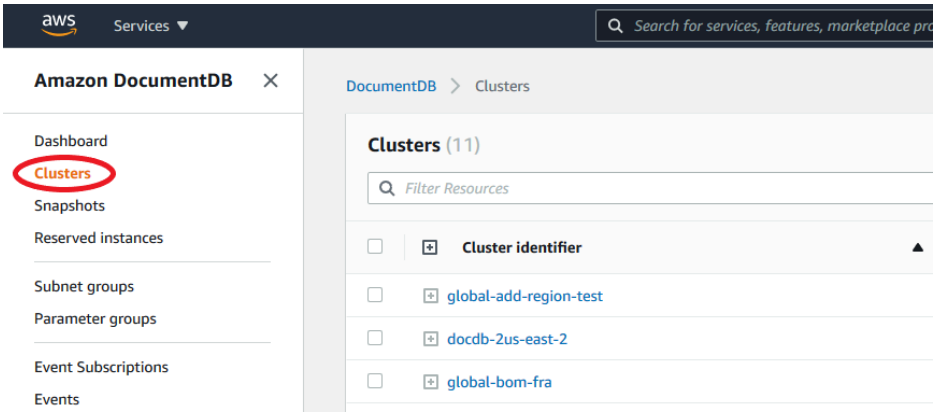
Deleting a cluster from an Amazon DocumentDB global cluster

To delete a global cluster, do the following:

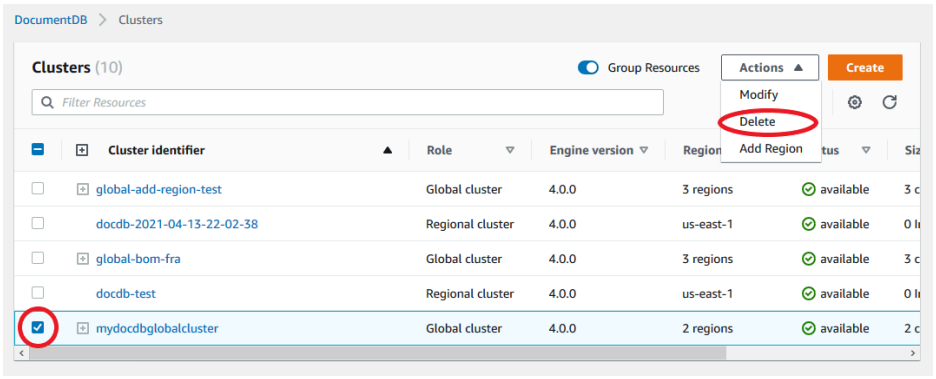
- Remove all secondary clusters from the global cluster. Each cluster becomes a standalone cluster. See the previous section, Removing a cluster from an Amazon DocumentDB global cluster.
- From each standalone cluster, delete all replicas.
- Remove the primary cluster from the global cluster. This becomes a standalone cluster.
- From the primary cluster, first delete all replicas, then delete the primary instance. Deleting the primary instance from the newly standalone cluster also typically removes both the cluster and the global cluster.

Using the AWS Management Console

1. Sign in to the AWS Management Console and navigate to the Amazon DocumentDB console.
2. Choose **Clusters** and find the global cluster you want to delete.



3. With your global cluster selected, choose **Delete** from the **Actions** menu.



Confirm that all clusters are removed from the global cluster. The global cluster should show zero Regions and AZs and a size of zero clusters. If the global cluster contains any clusters, you can't delete it yet. You'll first have to follow the instructions in the previous step, **Removing a cluster from an Amazon DocumentDB global cluster**.

Using the AWS CLI

To delete a global cluster, run the `delete-global-cluster` CLI command with the name of the AWS Region and the global cluster identifier, as shown in the following example.

For Linux, macOS, or Unix:

```
aws docdb --region primary_region delete-global-cluster \
   --global-cluster-identifier global_cluster_id
```

For Windows:

```
aws docdb --region primary_region delete-global-cluster ^
   --global-cluster-identifier global_cluster_id
```

Creating a headless Amazon DocumentDB cluster in a secondary Region

Although an Amazon DocumentDB global cluster requires at least one secondary cluster in a different AWS Region than the primary, you can use a headless configuration for the secondary cluster. A headless secondary Amazon DocumentDB cluster is one without an instance. This type of configuration can lower expenses for a global cluster. In an Amazon DocumentDB cluster, compute and storage are decoupled. Without the instance, you're not charged for compute, only for storage. If it's set up correctly, a headless secondary's storage volume is kept in sync with the primary cluster.

You add the secondary cluster as you normally do when creating an Amazon DocumentDB global cluster. However, after the primary cluster begins replication to the secondary, you delete the read-only instance from the secondary cluster. This secondary cluster is now considered "headless" because it no longer has an instance. Yet, the storage volume is kept in sync with the primary Amazon DocumentDB cluster.

> **Important**
>
> Use headless clusters only if you can tolerate Region-wide failures for 15 minutes or more. Recovering from a Region-wide failure with a headless secondary cluster requires you to create a new instance after failing over. A new instance can take 10–15 minutes to become available.

How to add a headless secondary cluster to your global cluster

1. Sign in to the AWS Management Console and open the Amazon DocumentDB console.
2. Choose **Clusters** on the left side navigation.
3. Choose the global cluster that needs a secondary cluster. Ensure that the primary cluster is `Available`.
4. For **Actions**, choose **Add Region**.
5. On the **Add a region** page, choose the secondary Region.

> **Note**
>
> You can't choose a Region that already has a secondary cluster for the same global cluster. Also, it can't be the same Region as the primary cluster.

6. Complete the remaining fields for the secondary cluster in the new Region. These are the same configuration options as for any cluster instance.
7. Add a Region. After you finish adding the Region to your global cluster, it appears in the list of `Clusters` in the AWS Management Console.
8. Check the status of the secondary cluster and its reader instance before continuing, by using the AWS Management Console or the AWS CLI. Here is a sample command if you use the AWS CLI:

   ```
   $ aws docdb describe-db-clusters --db-cluster-identifier secondary-cluster-id --
   query '*[].[Status]' --output text
   ```

   It can take several minutes for the status of a newly added secondary cluster to change from creating to available. When the cluster is available, you can delete the reader instance.
9. Select the reader instance in the secondary cluster, and then choose **Delete**.
10. After deleting the reader instance, the secondary cluster remains part of the global cluster. It should have no instance associated with it.

> **Note**
>
> You can use this headless secondary Amazon DocumentDB cluster to manually recover your Amazon DocumentDB global cluster from an unplanned outage in the primary Region if such an outage occurs.

Connect to an Amazon DocumentDB global cluster

How you connect to a global cluster depends on whether you need to write to the cluster or read from the cluster:

- For read-only requests or queries, you connect to the reader endpoint for the cluster in your AWS Region.
- To run data manipulation language (DML) or data definition language (DDL) statements, you connect to the cluster endpoint for the primary cluster. This endpoint might be in a different AWS Region than your application.

When you view a global cluster in the console, you can see all the general-purpose endpoints associated with all of its clusters.

How you connect to a global cluster depends on whether you need to write to the database or read from the database. For DDL, DML and read operations that you would like to serve from the primary Region, connect to your primary cluster using the cluster endpoint in replica set mode, with a read preference of `secondaryPreferred=true`. This routes write traffic to your primary cluster's writer instance and read traffic to your primary cluster's replica instance.

For cross-Region, read-only traffic, connect to one of your secondary clusters using the cluster endpoint in replica set mode. Because all instances are read-only replica instances, specify a read preference other than `primary` (for example, `secondary`, `secondaryPreferred`, or `nearest`). To minimize latency, choose whichever reader endpoint is in your Region or the Region closest to you.

Monitoring Amazon DocumentDB global clusters

Amazon DocumentDB (with MongoDB compatibility) integrates with CloudWatch so that you can gather and analyze operational metrics for your clusters. You can monitor these metrics using the CloudWatch console, the Amazon DocumentDB console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API.

To monitor a global cluster, use the following CloudWatch metrics.

Metric	Description
`GlobalClusterReplicatedWriteIO`	The average number of billed write I/O operations replicated from the cluster volume in the primary AWS Region to the cluster volume in a secondary AWS Region, reported at 5-minute intervals. The number of replicated `ReplicatedWriteIOs` to each secondary Region is the same as the number of in-Region `VolumeWriteIOPs` performed by the primary Region.
`GlobalClusterDataTransferBytes`	The amount of data transferred from the primary cluster's AWS Region to a secondary cluster's AWS Region, measure in bytes.

Metric	Description
GlobalClusterReplicationLag	The amount of lag, in milliseconds, when replicating change events from the primary cluster's AWS Region to a secondary cluster's AWS Region

For more information on how to view these metrics, see Viewing CloudWatch data.

Disaster recovery and Amazon DocumentDB global clusters

Topics

- Performing a managed failover for an Amazon DocumentDB global cluster
- Performing a manual failover for an Amazon DocumentDB global cluster
- Performing a switchover for an Amazon DocumentDB global cluster
- Unblocking a global cluster switchover or failover
- Managing RPOs for Amazon DocumentDB global clusters

By using a global cluster, you can recover from disasters such as Region failures quickly. Recovery from disaster is typically measured using values for RTO and RPO.

- **Recovery time objective (RTO)** — The time it takes a system to return to a working state after a disaster. In other words, RTO measures downtime. For a global cluster, RTO in minutes.
- **Recovery point objective (RPO)** — The amount of data that can be lost (measured in time). For a global cluster, RPO is typically measured in seconds.
- To recover from an unplanned outage, you can perform a cross-Region failover to one of the secondaries in your global cluster. When your global cluster has multiple secondary Regions, make sure that you detach all the secondary Regions that you wish to promote as primaries. Then, you promote one of those secondary Regions to be the new primary AWS Region. Finally, you create new clusters in each of the other secondary Regions and attach those clusters to your global cluster.

Performing a managed failover for an Amazon DocumentDB global cluster

This approach is intended for business continuity in the event of a true Regional disaster or complete service-level outage.

During a managed failover, your primary cluster is failed over to your choice of secondary Region while your Amazon DocumentDB global cluster's existing replication topology is maintained. The chosen secondary cluster promotes one of its read-only nodes to full writer status. This step allows the cluster to assume the role of primary cluster. Your database is unavailable for a short time while this cluster is assuming its new role. Data that wasn't replicated from the old primary to the chosen secondary cluster may be missing when this secondary becomes the new primary. The old primary volume makes a best effort attempt to take a snapshot before synchronizing with the new primary so unreplicated data is preserved on the snapshot.

> **Note**
>
> You can only perform a managed cross-Region cluster failover on an Amazon DocumentDB global cluster if the primary and all secondary clusters have the same engine versions. If your engine versions are incompatible, you can perform the failover manually by following the steps in Performing a manual failover for an Amazon DocumentDB global cluster.
> If the Region's engine versions do not match, the failover will be blocked. Check for any pending upgrades and apply them to ensure all Region's engine versions match and the global cluster failover is unblocked. For more information, see Unblocking a global cluster switchover or failover.

To minimize data loss, do the following before using this feature:

- Take applications offline to prevent writes from being sent to the primary cluster of the Amazon DocumentDB global cluster.
- Check lag times for all Amazon DocumentDB secondary clusters. Choosing the secondary Region with the least replication lag can minimize data loss with the current failed primary Region. Check lag times for all Amazon DocumentDB secondary clusters in the global cluster by viewing the `GlobalClusterReplicationLag` metric in Amazon CloudWatch. These metrics show you how far behind (in milliseconds) replication to a secondary cluster is to the primary cluster.

  For more information about CloudWatch metrics for Amazon DocumentDB, see Amazon DocumentDB metrics.

During a managed failover, the chosen secondary cluster is promoted to its new role as primary. However, it doesn't inherit the various configuration options of the primary cluster. A mismatch in configuration can lead to performance issues, workload incompatibilities, and other anomalous behavior. To avoid such issues, resolve differences between your Amazon DocumentDB global clusters for the following:

- **Configure an Amazon DocumentDB cluster parameter group for the new primary, if necessary** — You can configure your Amazon DocumentDB cluster parameter groups independently for each cluster in your Amazon DocumentDB global cluster. Therefore, when you promote a secondary cluster to take over the primary role, the parameter group from the secondary might be configured differently than for the primary. If so, modify the promoted secondary cluster's parameter group to conform to your primary cluster's settings. To learn how, see Modifying Amazon DocumentDB cluster parameter groups.
- **Configure monitoring tools and options, such as Amazon CloudWatch events and alarms** — Configure the promoted cluster with the same logging ability, alarms, and so on as needed for the global cluster. As with parameter groups, configuration for these features isn't inherited from the primary during the failover process. Some CloudWatch metrics, such as replication lag, are only available for secondary Regions. Thus, a failover changes how to view those metrics and set alarms on them, and could require changes to any predefined dashboards. For more information about Amazon DocumentDB clusters and monitoring, see Monitoring and logging in Amazon DocumentDB.

Typically, the chosen secondary cluster assumes the primary role within a minute. As soon as the new primary Region's writer node is available, you can connect your applications to it and resume your workloads. After Amazon DocumentDB promotes the new primary cluster, it automatically rebuilds all additional secondary Region clusters.

Because Amazon DocumentDB global clusters use asynchronous replication, the replication lag in each secondary Region can vary. Amazon DocumentDB rebuilds these secondary Regions to have the exact same point-in-time data as the new primary Region cluster. The duration of the complete rebuilding task can take a few minutes to several hours, depending on the size of the storage volume and the distance between the Regions. When the secondary Region clusters finish rebuilding from the new primary Region, they become available for read access. As soon as the new primary writer is promoted and available, the new primary Region's cluster can handle read and write operations for the Amazon DocumentDB global cluster.

To restore the global cluster's original topology, Amazon DocumentDB monitors the availability of the old primary Region. As soon as that Region is healthy and available again, Amazon DocumentDB automatically adds it back to the global cluster as a secondary Region. Before creating the new storage volume in the old primary Region, Amazon DocumentDB tries to take a snapshot of the old storage volume at the point of failure. It does this so that you can use it to recover any of the missing data. If this operation is successful, Amazon DocumentDB places this snapshot named "rds:docdb-unplanned-global-failover-name-of-old-primary-DB-cluster-timestamp" in the snapshot section of the AWS Management Console. You can also see this snapshot listed in the information returned by the `DescribeDBClusterSnapshots` API operation.

> **Note**
>
> The snapshot of the old storage volume is a system snapshot that's subject to the backup retention period configured on the old primary cluster. To preserve this snapshot outside of the retention period, you can copy it to save it as a manual snapshot. To learn more about copying snapshots, including pricing, see Copying a cluster snapshot.

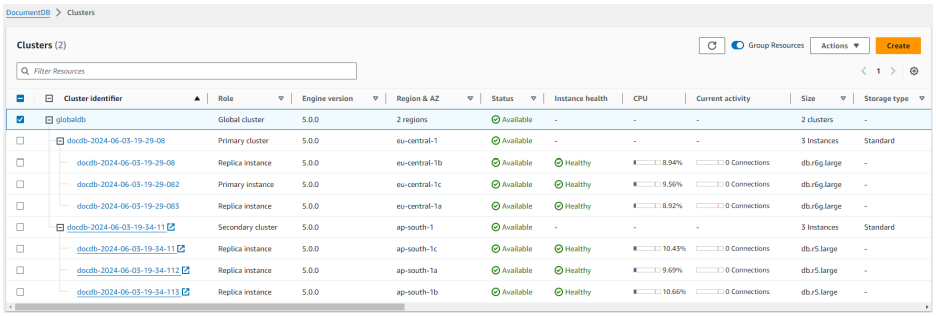
After the original topology is restored, you can fail back your global cluster to the original primary Region by performing a switchover operation when it makes the most sense for your business and workload. To do so, follow the steps in Performing a switchover for an Amazon DocumentDB global cluster.

You can fail over your Amazon DocumentDB global cluster using the AWS Management Console, the AWS CLI, or the Amazon DocumentDB API.

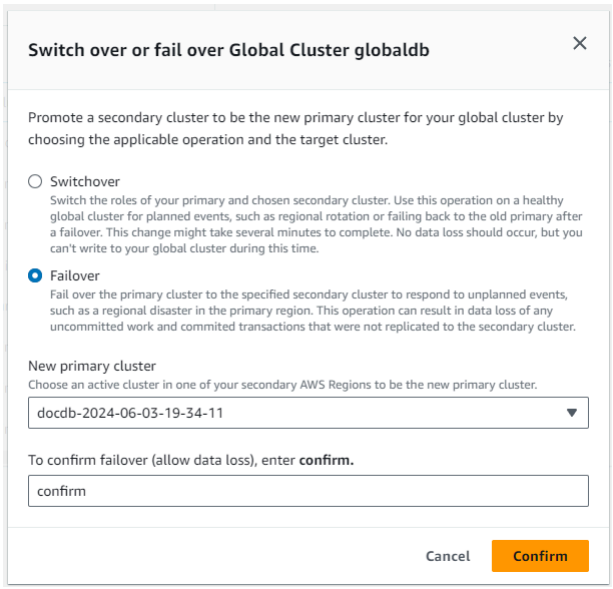
Using the AWS Management Console

Perform a managed failover on your Amazon DocumentDB global cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. Find and choose the Amazon DocumentDB global cluster you want to fail over.

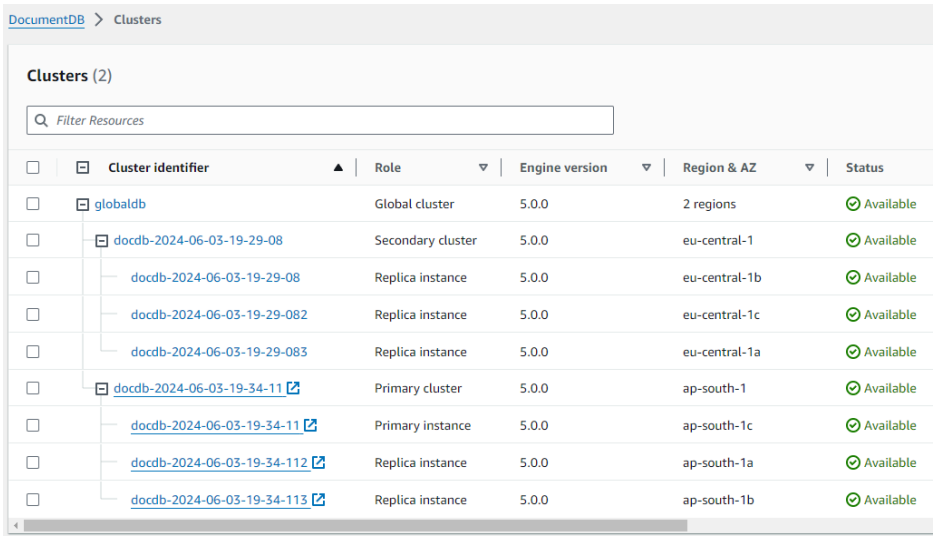


4. Choose **Switchover or Failover** from the **Actions** menu.
5. On the dialog box that appears, choose **Failover**, then choose the secondary cluster from the **New primary cluster** field drop down list.



6. Type "confirm" in the last field. Then choose **Confirm**.

   The status of the primary cluster changes to "**Failing-over**". This condition should take approximately one minute. During this time, the status of the new primary cluster shows "**Modifying...**". Once the new primary is promoted, it will show "**Available**" and will be able to serve read and write transactions. The secondary Regions including the old primary will show "**Resyncing...**" while it resynchronizes to the new primary. Similar to the new primary, it will only be able to serve transaction once the status changes to "**Available**".
7. When complete, the original primary cluster becomes the secondary cluster. The selected secondary cluster becomes the primary cluster.



Using the AWS CLI

Perform a managed failover on your Amazon DocumentDB global cluster

Run the `failover-global-cluster` CLI command to fail over your Amazon DocumentDB global cluster. With the command, pass values for the following options:

- `--region`
- `--global-cluster-identifier`
- `--target-db-cluster-identifier`
- `--allow-data-loss`

In the following examples, replace each *`user input placeholder`* with your cluster's information.

For Linux, macOS, or Unix:

```
aws docdb failover-global-cluster \
   --region region_of_selected_secondary \
   --global-cluster-identifier global_cluster_id \
   --target-db-cluster-identifier arn_of_secondary_to_promote \
   --allow-data-loss
```

For Windows:

```
aws docdb failover-global-cluster ^
   --region region_of_selected_secondary ^
   --global-cluster-identifier global_cluster_id ^
   --target-db-cluster-identifier arn_of_secondary_to_promote ^
   --allow-data-loss
```

Performing a manual failover for an Amazon DocumentDB global cluster

If an entire cluster in one AWS Region becomes unavailable, you can promote another cluster in the global cluster to have read/write capability.

You can manually activate the global cluster failover mechanism if a cluster in a different AWS Region is a better choice to be the primary cluster. For example, you might increase the capacity of one of the secondary clusters and then promote it to be the primary cluster. Or the balance of activity among the AWS Regions might change, so that switching the primary cluster to a different AWS Region might give lower latency for write operations.

The following procedure outlines what to do to promote one of the secondary clusters in an Amazon DocumentDB global cluster.

To promote a secondary cluster:

1. Stop issuing DML statements and other write operations to the primary cluster in the AWS Region with the outage.
2. Identify a cluster from a secondary AWS Region to use as a new primary cluster. If you have two (or more) secondary AWS Regions in your global cluster, choose the secondary cluster that has the least lag time.
3. Detach your chosen secondary cluster from the global cluster.

   Removing a secondary cluster from a global cluster immediately stops the replication from the primary to this secondary and promotes it to standalone provisioned cluster with full read/write capabilities. Any other secondary cluster associated with the primary cluster in the Region with the outage are still available and can accept calls from your application. They also consume resources. Since you are recreating the global cluster, to avoid split-brain and other issues, remove the other secondary clusters before creating the new global cluster in the steps that follow.

For detailed steps for detaching, see Removing a cluster from an Amazon DocumentDB global cluster.

4. This cluster becomes the primary cluster of a new global cluster when you start adding Regions to it, in the next step.
5. Add an AWS Region to the cluster. When you do this, the replication process from primary to secondary begins.
6. Add more AWS Regions as needed to re-create the topology needed to support your application. Make sure that application writes are sent to the correct cluster before, during, and after making changes such as these, to avoid data inconsistencies among the clusters in the global cluster (split-brain issues).
7. When the outage is resolved and you're ready to assign your original AWS Region as the primary cluster again, perform the same steps in reverse.
8. Remove one of the secondary clusters from the global cluster. This will enable it to serve read/write traffic.
9. Redirect all the write traffic to the primary cluster in the original AWS Region.
10. Add an AWS Region to set up one or more secondary clusters in the same AWS Region as before.

Amazon DocumentDB global clusters can be managed using AWS SDKs, enabling you to create solutions to automate global cluster failover process for Disaster Recovery and Business Continuity Planning use cases. One such solution is made available for our customers under Apache 2.0 licensing and can be accessed from our tools repository here. This solution leverages Amazon Route 53 for endpoint management and provides AWS Lambda functions that can be triggered based appropriate events.

Performing a switchover for an Amazon DocumentDB global cluster

By using switchovers, you can change the Region of your primary cluster on a routine basis. This approach is intended for controlled scenarios, such as operational maintenance and other planned operational procedures.

There are three common use cases for using switchovers:

- For "regional rotation" requirements imposed on specific industries. For example, financial service regulations might want tier-0 systems to switch to a different Region for several months to ensure that disaster recovery procedures are regularly exercised.

- For multi-Region "follow-the-sun" applications. For example, a business might want to provide lower latency writes in different Regions based on business hours across different time zones.
- As a zero-data-loss method to fail back to the original primary Region after a failover.

> **Note**
>
> Switchovers are designed to be used on a healthy Amazon DocumentDB global cluster. To recover from an unplanned outage, follow the appropriate procedure in Performing a manual failover for an Amazon DocumentDB global cluster.
> To perform a switchover, all secondary Regions must be running the exact same engine version as the primary. If the Region's engine versions do not match, the switchover will be blocked. Check for any pending upgrades and apply them to ensure all Region's engine versions match and the global cluster switchover is unblocked. For more information, see Unblocking a global cluster switchover or failover.

During a switchover, Amazon DocumentDB switches over your primary cluster to your chosen secondary Region while it maintains your global cluster's existing replication topology. Before it starts the switchover process, Amazon DocumentDB waits for all secondary Region clusters to be fully synchronized with the primary Region cluster. Then, the DB cluster in the primary Region becomes read-only and the chosen secondary cluster promotes one of its read-only nodes to full writer status. Promoting this node to a writer allows that secondary cluster to assume the role of primary cluster. Because all secondary clusters were synchronized with the primary at the beginning of the process, the new primary continues operations for the Amazon DocumentDB global cluster without losing any data. Your database is unavailable for a short time while the primary and selected secondary clusters are assuming their new roles.

To optimize application availability, do the following before using this feature:

- Perform this operation during nonpeak hours or at another time when writes to the primary cluster are minimal.
- Take applications offline to prevent writes from being sent to the primary cluster of the Amazon DocumentDB global cluster.
- Check lag times for all Amazon DocumentDB secondary clusters in the global cluster by viewing the `GlobalClusterReplicationLag` metric in Amazon CloudWatch. This metric shows you how far behind (in milliseconds) replication to a secondary cluster is to the primary cluster. This

value is directly proportional to the time it takes for Amazon DocumentDB to complete the switchover. Therefore, the larger the lag value, the longer the switchover will take.

For more information about CloudWatch metrics for Amazon DocumentDB, see Amazon DocumentDB metrics.

During a switchover, the chosen secondary DB cluster is promoted to its new role as primary. However, it doesn't inherit the various configuration options of the primary DB cluster. A mismatch in configuration can lead to performance issues, workload incompatibilities, and other anomalous behavior. To avoid such issues, resolve differences between your Amazon DocumentDB global clusters for the following:

- **Configure Amazon DocumentDB DB cluster parameter group for the new primary, if necessary** — You can configure your Amazon DocumentDB cluster parameter groups independently for each cluster in your Amazon DocumentDB global cluster. That means that when you promote a secondary DB cluster to take over the primary role, the parameter group from the secondary might be configured differently than for the primary. If so, modify the promoted secondary DB cluster's parameter group to conform to your primary cluster's settings. To learn how, see Managing Amazon DocumentDB cluster parameter groups.
- **Configure monitoring tools and options, such as Amazon CloudWatch Events and alarms** — Configure the promoted cluster with the same logging ability, alarms, and so on as needed for the global cluster. As with parameter groups, configuration for these features isn't inherited from the primary during the switchover process. Some CloudWatch metrics, such as replication lag, are only available for primary Regions. Thus, a switchover changes how to view those metrics and set alarms on them, and could require changes to any predefined dashboards. For more information, see Monitoring and logging in Amazon DocumentDB.

> **Note**
>
> Typically, the role switchover can take up to several minutes.

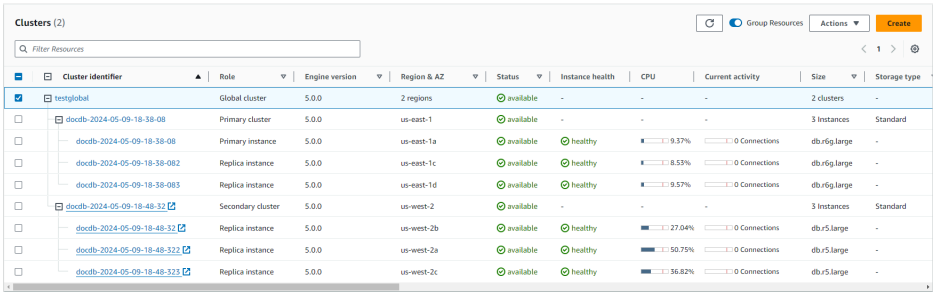
When the switchover process completes, the promoted Amazon DocumentDB cluster can handle write operations for the global cluster.

You can switch over your Amazon DocumentDB global cluster using the AWS Management Console or the AWS CLI:

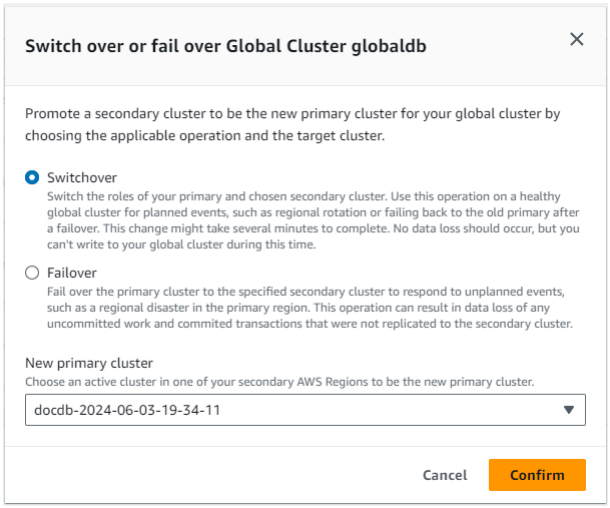
Using the AWS Management Console

Perform a switchover on your Amazon DocumentDB global cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. Find and select the Amazon DocumentDB global cluster you want to switch over.

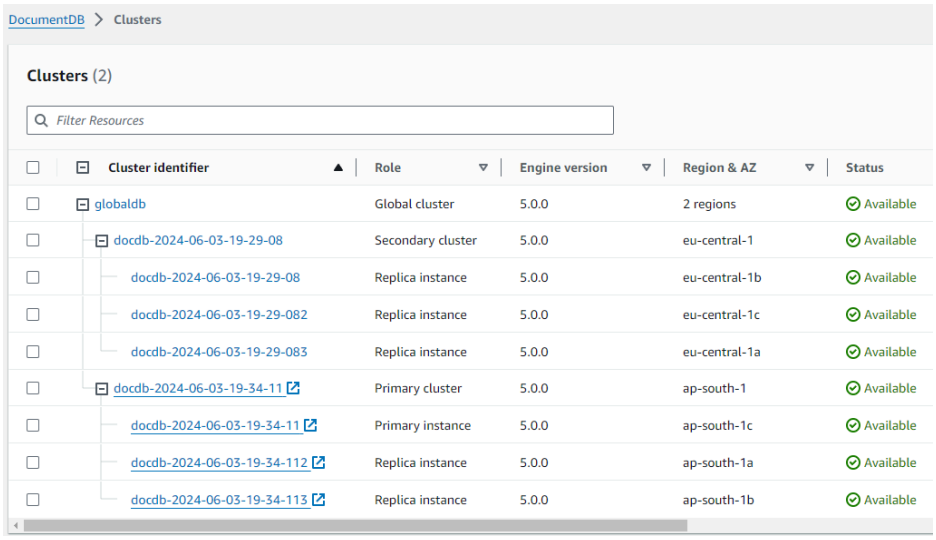


4. Choose **Switchover or Failover** from the **Actions** menu.
5. On the dialog box that appears, choose **Switchover**, then choose the secondary cluster from the **New primary cluster** field drop down list.



6. Choose **Confirm**.

   The status of the primary cluster changes to "**Switching-over**". This condition should take approximately three minutes. During this time, the status of all regional clusters show "**Modifying...**". Once the Regions are synchronized and the new primary is promoted, it will show "**Available**" for all status fields and will be able to serve transactions.
7. When complete, the original primary cluster becomes the secondary cluster. The selected secondary cluster becomes the primary cluster.



Using the AWS CLI

Perform a switchover on your Amazon DocumentDB global cluster

Run the `switchover-global-cluster` CLI command to switch over your Amazon DocumentDB global cluster. With the command, pass values for the following options:

- `--region`
- `--global-cluster-identifier`
- `--target-db-cluster-identifier`

In the following examples, replace each *`user input placeholder`* with your cluster's information.

For Linux, macOS, or Unix:

```
aws docdb switchover-global-cluster \
   --region region_of_primary \
   --global-cluster-identifier global_cluster_id \
   --target-db-cluster-identifier arn_of_secondary_to_promote
```

For Windows:

```
aws docdb switchover-global-cluster ^
   --region region_of_primary ^
   --global-cluster-identifier global_cluster_id ^
```

```
    --target-db-cluster-identifier arn_of_secondary_to_promote
```

Unblocking a global cluster switchover or failover

Regions on different engine versions

Global cluster switchovers and failovers are blocked when not all regional clusters in the global cluster are on the same engine version. If the versions don't match, you might see this error when calling a switchover or failover: The target DB cluster specified is running an engine version with a different patch level than the source DB cluster. Routinely apply the latest engine versions to keep your global clusters in a healthy state.

To resolve this error, update all secondary Regions first, and then the primary Region to the same engine version by applying any pending maintenance action items. To view pending maintenance action items, and to apply any needed changes to correct the issue, perform the instructions in one of the following tabs:

Using the AWS Management Console

To unblock a global cluster switchover or failover, you must determine if there are any pending maintenance actions for your clusters and apply them. Follow these steps to view and apply maintenance actions:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. In the **Clusters** table, locate your global cluster in the **Cluster identifier** column. Under your global cluster, take note of each secondary cluster and the primary cluster for the given global cluster, and perform the following steps for each.
4. For each secondary cluster:

    a. If an update is available for your cluster, it is indicated as **Available**, **Required**, or **Next Window** in the **Maintenance** column.

    b. To take an action, choose the cluster to show it's details, then choose **Maintenance & backups**. The **Pending Maintenance** items appear.

    c. Under **Description**, if it indicates that a "New maintenance update is available", select it and then choose **Apply now**.

5. For your primary cluster:

   a. If an update is available for your cluster, it is indicated as **Available**, **Required**, or **Next Window** in the **Maintenance** column.

   b. To take an action, choose the cluster to show it's details, then choose **Maintenance & backups**. The **Pending Maintenance** items appear.

   c. Under **Description**, if it indicates that a "New maintenance update is available", select it and then choose **Apply now**.

Using the AWS CLI

To unblock a global cluster switchover or failover, you must determine if there are any pending maintenance actions for the cluster and apply them. Follow these steps to view and apply maintenance actions first on the secondary clusters then on the primary cluster for your global cluster:

1. Run the following on each secondary Region's regional cluster first and then for the primary Regions regional cluster.
2. Run the describe-pending-maintenance-actions CLI command with the `--resource-identifier` option to determine if any maintenance actions are available for your Amazon DocumentDB regional cluster.

   In the following examples, replace each *`user input placeholder`* with your cluster's information.

   For Linux, macOS, or Unix:

```
aws docdb describe-pending-maintenance-action \
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15
```

   For Windows:

```
aws docdb describe-pending-maintenance-action ^
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15
```

   The result looks similar to this:

```
{
    "PendingMaintenanceActions": [
        {
            "ResourceIdentifier": "arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "system-update",
                    "CurrentApplyDate": "2025-04-11T03:01:00Z",
                    "Description": "db-version-upgrade",
                    "ForcedApplyDate": "2025-06-18T03:01:00Z",
                    "AutoAppliedAfterDate": "2025-05-11T03:01:00Z"
                    "OptInStatus": "pending"
                }
            ]
        }
    ]
}
```

3. If a maintenance action is needed, run the `apply-pending-maintenance-action` CLI command with the following options:

   - `--resource-identifier`
   - `--apply-action`
   - `--opt-in-type`
   - `--region`

   In the following examples, replace each *`user input placeholder`* with your cluster's information.

   For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 \
   --apply-action system-update \
   --opt-in-type immediate \
   --region us-east-1
```

For Windows:

```
aws docdb apply-pending-maintenance-action ^
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 ^
   --apply-action system-update ^
   --opt-in-type immediate ^
   --region us-east-1
```

4. Once the maintenance action has completed, run the describe-pending-maintenance-actions command again to ensure that there are no other actions pending for your cluster.

   The result you want is:

```
{
    "PendingMaintenanceActions": []
}
```

Using the Amazon DocumentDB API

To unblock a global cluster switchover or failover, you must determine if there are any pending maintenance actions for the cluster and apply them. Use the following APIs to view and apply maintenance actions:

1. Run the following on each secondary Region's regional cluster first and then for the primary Regions regional cluster.
2. Call the PendingMaintenanceAction API to determine if any maintenance actions are available for your Amazon DocumentDB global cluster.
3. Apply any changes by calling the ApplyPendingMaintenanceAction API.

Pending changes on the target cluster

Switchovers and failovers are also blocked when the target secondary cluster has a scheduled change that hasn't been applied yet, such as a modification or maintenance action you requested for the next maintenance window. In this case, you might see this error when calling a switchover or failover: You can't fail over to the cluster with ARN *arn:aws:rds:us-*

east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 because it's being modified or because modifications are pending. Try again when the cluster becomes available. To unblock the operation, remove the scheduled change on the target cluster. The steps depend on the type of change. After the scheduled change is removed and the cluster status returns to `available`, retry the switchover or failover.

Cancel a scheduled maintenance action

You can't cancel a scheduled maintenance action (undo an opt-in) from the AWS Management Console, so use the AWS CLI.

Using the AWS CLI

1. Run the describe-pending-maintenance-actions CLI command with the `--resource-identifier` option, and look for a maintenance action whose `OptInStatus` is `next-maintenance`.

   In the following examples, replace each *user input placeholder* with your cluster's information.

   For Linux, macOS, or Unix:

   ```
   aws docdb describe-pending-maintenance-actions \
      --resource-identifier arn:aws:rds:us-
   east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 \
      --region us-east-1
   ```

   For Windows:

   ```
   aws docdb describe-pending-maintenance-actions ^
      --resource-identifier arn:aws:rds:us-
   east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 ^
      --region us-east-1
   ```

   The result looks similar to the following. Note the `Action` value (`os-upgrade` in this example), which you use in the next step.

   ```
   {
       "PendingMaintenanceActions": [
           {
   ```

```
            "ResourceIdentifier": "arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "os-upgrade",
                    "OptInStatus": "next-maintenance",
                    "CurrentApplyDate": "2026-09-02T03:02:00Z",
                    "Description": "New Operating System update is available"
                }
            ]
        }
    ]
}
```

2. Cancel the scheduled action by running the apply-pending-maintenance-action CLI command with `--opt-in-type undo-opt-in`, passing the `Action` value from the previous step to `--apply-action`.

   For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 \
   --apply-action os-upgrade \
   --opt-in-type undo-opt-in \
   --region us-east-1
```

   For Windows:

```
aws docdb apply-pending-maintenance-action ^
   --resource-identifier arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15 ^
   --apply-action os-upgrade ^
   --opt-in-type undo-opt-in ^
   --region us-east-1
```

   In the response, the maintenance action no longer has an `OptInStatus`, which confirms the scheduled opt-in was cancelled.

```
{
    "ResourcePendingMaintenanceActions": {
```

```
        "ResourceIdentifier": "arn:aws:rds:us-
east-1:001234567890:cluster:docdb-2025-03-27-19-21-15",
        "PendingMaintenanceActionDetails": [
            {
                "Action": "os-upgrade",
                "Description": "New Operating System update is available"
            }
        ]
    }
}
```

Revert a scheduled cluster modification

Using the AWS CLI

1. Check for scheduled modifications by running the describe-db-clusters CLI command and inspecting the `PendingModifiedValues` output field.

   > **Note**
   >
   > Use the Amazon RDS CLI (`aws rds`) for this command, not the Amazon DocumentDB CLI (`aws docdb`), because the Amazon DocumentDB `describe-db-clusters` API doesn't return a cluster-level `PendingModifiedValues` field.

   In the following examples, replace each *`user input placeholder`* with your cluster's information.

   For Linux, macOS, or Unix:

   ```
   aws rds describe-db-clusters \
      --db-cluster-identifier docdb-2025-03-27-19-21-15 \
      --query 'DBClusters[0].PendingModifiedValues' \
      --region us-east-1
   ```

   For Windows:

   ```
   aws rds describe-db-clusters ^
      --db-cluster-identifier docdb-2025-03-27-19-21-15 ^
   ```

```
    --query "DBClusters[0].PendingModifiedValues" ^
    --region us-east-1
```

The result shows the scheduled changes. In this example, a change to the cluster's name (`DBClusterIdentifier`) is pending, from `docdb-2025-03-27-19-21-15` to `docdb-new-name`.

```
{
    "DBClusterIdentifier": "docdb-new-name"
}
```

2. Revert the change by running the modify-db-cluster CLI command with `--apply-immediately`, setting the modified value back to its original value. This example reverts the pending name change by setting `--new-db-cluster-identifier` back to the cluster's current name.

   For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
    --db-cluster-identifier docdb-2025-03-27-19-21-15 \
    --new-db-cluster-identifier docdb-2025-03-27-19-21-15 \
    --apply-immediately \
    --region us-east-1
```

   For Windows:

```
aws docdb modify-db-cluster ^
    --db-cluster-identifier docdb-2025-03-27-19-21-15 ^
    --new-db-cluster-identifier docdb-2025-03-27-19-21-15 ^
    --apply-immediately ^
    --region us-east-1
```

Managing RPOs for Amazon DocumentDB global clusters

With a Amazon DocumentDB global cluster, you can manage the recovery point objective (RPO) by using the `global_db_rpo` parameter. RPO represents the maximum amount of data that can be lost in the event of an outage.

When you set an RPO for your Amazon DocumentDB global cluster, Amazon DocumentDB monitors the *RPO lag time* of all secondary clusters. This monitoring ensures that at least one secondary cluster stays within the target RPO window.

The RPO setting controls how Amazon DocumentDB manages write transactions on the primary cluster to limit potential data loss if a failover occurs. Amazon DocumentDB evaluates RPO and RPO lag times to commit (or block) transactions on the primary as follows:

- Commits the transaction if at least one secondary DB cluster has an RPO lag time less than the RPO.
- Blocks the transaction if all secondary DB clusters have RPO lag times that are larger than the RPO.

In other words, if all secondary clusters are behind the target RPO, Amazon DocumentDB pauses transactions on the primary cluster. Amazon DocumentDB resumes and commits paused transactions as soon as the lag time of at least one secondary DB cluster drops below the RPO. The result is that no transactions can commit until the RPO is met.

The `global_db_rpo` parameter is dynamic. If you decide that you don't want all write transactions to stall until the lag decreases sufficiently, you can reset it quickly. In this case, Amazon DocumentDB applies the change after a short delay.

> **Important**
>
> In a global database with only two AWS Regions, we recommend keeping the `global_db_rpo` parameter's default value in the secondary Region's parameter group. Otherwise, performing a failover due to a loss of the primary AWS Region could cause Amazon DocumentDB to pause transactions. Instead, wait until Amazon DocumentDB completes rebuilding the cluster in the old failed AWS Region before changing this parameter to enforce a maximum RPO.

Topics

- Setting the recovery point objective
- Viewing the recovery point objective
- Disabling the recovery point objective

Setting the recovery point objective

The `global_db_rpo` parameter controls the RPO setting for a Amazon DocumentDB database. Valid values range from 20 seconds to 2,147,483,647 seconds (68 years). Choose a realistic value to meet your business need. For example, you might want to allow up to 10 minutes for your RPO, in which case you set the value to 600.

You can set this value for your Amazon DocumentDB global cluster by using the AWS Management Console, the AWS CLI, or the Amazon DocumentDB API.

Using the AWS Management Console

To set the RPO

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. Choose the primary cluster of your Amazon DocumentDB global cluster and open the **Configuration** tab to find its DB cluster parameter group.

   Parameter groups can't be edited directly. Instead, you do the following:

   - Create a custom DB cluster parameter group using the appropriate default parameter group as the starting point.
   - On your custom DB cluster parameter group, set the value of the **global_db_rpo** parameter to meet your use case. Valid values range from 20 seconds up to the maximum integer value of 2,147,483,647 (68 years).
   - Apply the modified DB cluster parameter group to your Amazon DocumentDB DB cluster.

For more information about modifying DB cluster parameter groups, see Modifying Amazon DocumentDB cluster parameter groups.

Using the AWS CLI

To set the `global_db_rpo` parameter, use the modify-db-cluster-parameter-group CLI command. In the command, specify the name of your primary cluster's parameter group and values for the RPO parameter.

The following example sets the RPO to 600 seconds (10 minutes) for the primary DB cluster's parameter group named `my_custom_global_parameter_group`.

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name my_custom_global_parameter_group \
    --parameters
 "ParameterName=global_db_rpo,ParameterValue=600,ApplyMethod=immediate"
```

For Windows:

```
aws docdb modify-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name my_custom_global_parameter_group ^
    --parameters
 "ParameterName=global_db_rpo,ParameterValue=600,ApplyMethod=immediate"
```

Using the Amazon DocumentDB API

To modify the `global_db_rpo` parameter, use the ModifyDBClusterParameterGroup API operation.

Viewing the recovery point objective

The recovery point objective (RPO) of a global cluster is stored in the `global_db_rpo` parameter for each DB cluster.

You can use the CLI to view the `global_db_rpo` parameter for a Amazon DocumentDB DB cluster. Use the `--query` option to return only the `global_db_rpo` parameter from the parameter group.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
    --db-cluster-parameter-group-name my_custom_global_parameter_group \
    --query "Parameters[?ParameterName=='global_db_rpo']"
```

For Windows:

```
aws docdb describe-db-cluster-parameters ^
    --db-cluster-parameter-group-name my_custom_global_parameter_group ^
    --query "Parameters[?ParameterName=='global_db_rpo']"
```

The command returns output similar to the following.

```
[
    {
        "ParameterName": "global_db_rpo",
        "Description": "(s) Recovery point objective threshold, in seconds, that blocks
 user commits when it is violated.",
        "Source": "engine-default",
        "ApplyType": "dynamic",
        "DataType": "integer",
        "AllowedValues": "20-2147483647",
        "IsModifiable": true,
        "ApplyMethod": "immediate"
    }
]
```

For more information about viewing parameters of the cluster parameter group, see Managing Amazon DocumentDB cluster parameter groups.

Disabling the recovery point objective

To disable the RPO, reset the `global_db_rpo` parameter. You can reset parameters using the AWS Management Console, the AWS CLI, or the Amazon DocumentDB API.

Using the AWS Management Console

To disable the RPO

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.
3. In the list, choose your primary DB cluster parameter group.
4. Choose the radio button next to the **global_db_rpo** parameter.
5. Choose **Reset to default** and confirm it.

For more information about how to reset a parameter with the console, see Modifying Amazon DocumentDB cluster parameter groups.

Using the AWS CLI

To reset the `global_db_rpo` parameter, use the reset-db-cluster-parameter-group command.

For Linux, macOS, or Unix:

```
aws docdb reset-db-cluster-parameter-group \
    --db-cluster-parameter-group-name my_custom_global_parameter_group \
    --parameters "ParameterName=global_db_rpo,ApplyMethod=immediate"
```

For Windows:

```
aws docdb reset-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name my_custom_global_parameter_group ^
    --parameters "ParameterName=global_db_rpo,ApplyMethod=immediate"
```

Using the Amazon DocumentDB API

To reset the `global_db_rpo` parameter, use the ResetDBClusterParameterGroup API operation.

Managing Amazon DocumentDB clusters

To manage an Amazon DocumentDB cluster, you must have an IAM policy with the appropriate Amazon DocumentDB control plane permissions. These permissions allow you to create, modify, and delete clusters and instances. The `AmazonDocDBFullAccess` policy provides all the required permissions for administering an Amazon DocumentDB cluster.

The following topics show how to perform various tasks when working with Amazon DocumentDB clusters, including creating, deleting, modifying, connecting to, and viewing clusters.

Topics

- Understanding clusters
- Amazon DocumentDB cluster settings
- Amazon DocumentDB cluster storage configurations
- Determining a cluster's status
- Amazon DocumentDB cluster lifecycle
- Scaling Amazon DocumentDB clusters

- Cloning a volume for an Amazon DocumentDB cluster
- Understanding Amazon DocumentDB cluster fault tolerance

Understanding clusters

Amazon DocumentDB separates compute and storage, and offloads data replication and backup to the cluster volume. A cluster volume provides a durable, reliable, and highly available storage layer that replicates data six ways across three Availability Zones. Replicas enable higher data availability and read scaling. Each cluster can scale up to 15 replicas.

Noun	Description	API Operations (Verbs)

Cluster	Consists of one or more instances and a cluster storage volume that manages the data for those instances.	`create-db-cluster`
		`delete-db-cluster`
		`describe-db-clusters`
		`modify-db-cluster`

Instance	Reading and writing data to the cluster storage volume is done via instances. In a given cluster, there are two types of instances: primary and replica. A cluster always has one primary instance and can have 0–15 replicas.	`create-db-instance`
		`delete-db-instance`
		`describe-db-instances`
		`modify-db-instance`
		`describe-orderable-db-instance-options`
		`reboot-db-instance`

Cluster volume	A virtual database storage volume that spans three Availability Zones, with each Availability Zone having two copies of the cluster data.	N/A

Noun	Description	API Operations (Verbs)
Primary instance	Supports both read and write operations, and performs all data modifications to the cluster volume. Each cluster has one primary instance.	N/A
Replica instance	Supports only read operations. Each Amazon DocumentDB cluster can have up to 15 replica instances in addition to the primary instance. Multiple replicas distribute the read workload. By locating replicas in separate Availability Zones, you can also increase database availability.	N/A
Cluster endpoint	An endpoint for an Amazon DocumentDB cluster that connects to the current primary instance for the cluster. Each Amazon DocumentDB cluster has a cluster endpoint and one primary instance.	N/A

Noun	Description	API Operations (Verbs)
Reader endpoint	An endpoint for an Amazon DocumentDB cluster that connects to one of the available replicas for that cluster. Each Amazon DocumentDB cluster has a reader endpoint. If there is more than one replica, the reader endpoint directs each connection request to one of the Amazon DocumentDB replicas.	N/A
Instance endpoint	An endpoint for an instance in an Amazon DocumentDB cluster that connects to a specific instance. Each instance in a cluster, regardless of instance type, has its own unique instance endpoint.	N/A

Amazon DocumentDB cluster settings

When you create or modify a cluster, it is important to understand which parameters are immutable and which are modifiable after the cluster has been created. The following table lists all the settings, or parameters, that are specific to a cluster. As specified in the table, some are modifiable, others are not.

> **Note**
>
> These settings should not be confused with Amazon DocumentDB cluster parameter groups and their parameters. For more information about cluster parameter groups, see Managing Amazon DocumentDB cluster parameter groups.

Parameter	Modifiable	Notes
DBClusterIdentifier	Yes	Naming constraints: • Length is [1—63] letters, numbers, or hyphens. • First character must be a letter. • Cannot end with a hyphen or contain two consecutive hyphens. • Must be unique for all clusters across Amazon RDS, Amazon Neptune, and Amazon DocumentDB per AWS account, per Region.
Engine	No	Must be `docdb`.
BackupRetentionPeriod	Yes	Must be between [1-35] days.
DBClusterParameterGroupName	Yes	Naming constraints: • Length is [1—255] alphanumeric characters. • First character must be a letter. • Cannot end with a hyphen or contain two consecutive hyphens.
DBSubnetGroupName	No	After a cluster has been created, you cannot modify the cluster's subnet.
EngineVersion	No	• If you omit `--engine-version` or specify `--engine-version 8.0` , Amazon DocumentDB uses the newest 8.0 minor version available in the AWS Region. This version advances as new 8.0 minor versions are released. • If you specify `--engine-version 5.0`, Amazon DocumentDB uses the 5.0

Parameter	Modifiable	Notes
		long-term support (LTS) minor version. This version changes only when Amazon DocumentDB declares a new 5.0 LTS minor version. • If you specify a fully-qualified version, such as `--engine-version 8.0.1` or `--engine-version 5.0.2` , Amazon DocumentDB uses exactly the version that you specify.
`KmsKeyId`	No	If you choose to encrypt your cluster, you cannot change the AWS KMS key that you used to encrypt your cluster.
`MasterUsername`	No	After a cluster has been created, you cannot modify the `MasterUsername` . Naming constraints: • Length is [1—63] alphanumeric characters. • First character must be a letter. • Cannot be a word reserved by the database engine.

`MasterUserPassword`	Yes	Constraints:
		• Length is [8—100] printable ASCII character s.
		• Can use any printable ASCII characters except for the following:
		  • **`/`** (forward slash)
		  • **`"`** (double quotation mark)
		  • **`@`** (at symbol)

Parameter	Modifiable	Notes
`Port`	Yes	The port number applies to all instances in the cluster.
`PreferredBackupWindow`	Yes	
`PreferredMaintenanceWindow`	Yes	
`StorageEncrypted`	No	If you choose to encrypt your cluster, it cannot be unencrypted.
`StorageType`	Yes	The storage type for the DB cluster: Standard (`standard`) or I/O-Optimized (`iopt1`). Default: `standard` This parameter can be configured with `CreateDBCluster` and `ModifyDBCluster` . For more information, see Amazon DocumentDB cluster storage configurations.
`Tags`	Yes	
`VpcSecurityGroupIds`	No	After a cluster has been created, you cannot modify the VPC that the cluster resides in.

Amazon DocumentDB cluster storage configurations

Starting from Amazon DocumentDB 5.0, instance-based clusters support two storage configuration types:

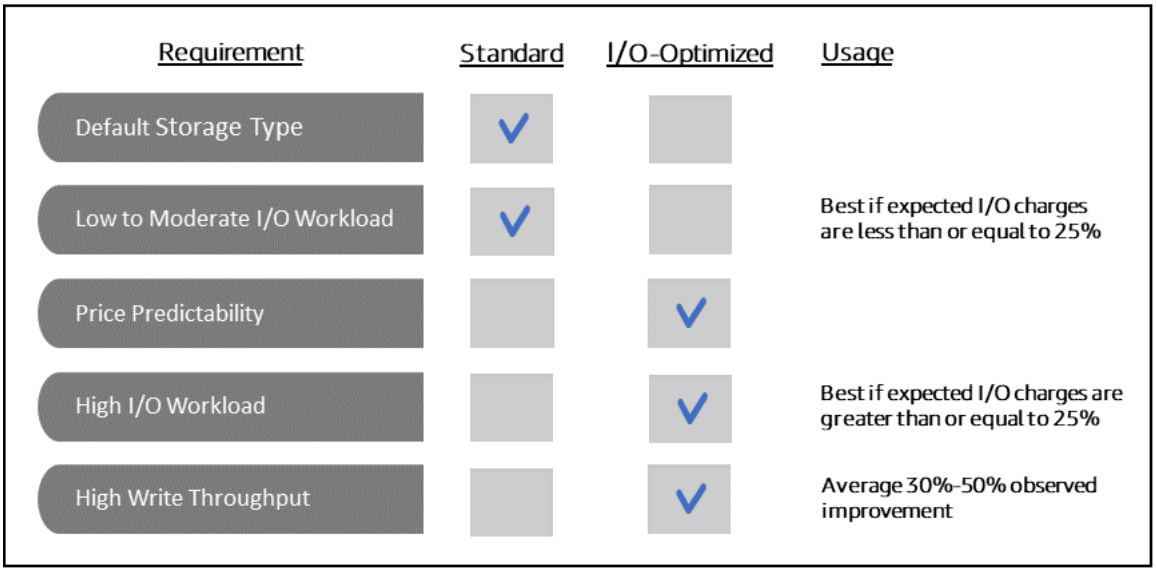
- **Amazon DocumentDB standard storage**: Designed for customers with low to moderate I/O consumption. If you expect your I/O costs to be less than 25% of your total Amazon DocumentDB cluster, this choice might be ideal for you. With the Amazon DocumentDB standard

storage configuration, you're billed on a pay-per-request I/O basis in addition to instance and storage charges. This means your billing might vary from one cycle to another based on usage. The configuration is tailored to accommodate fluctuating I/O demands of your application.

- **Amazon DocumentDB I/O-optimized storage**: Designed for customers who prioritize price predictability or have I/O intensive applications. The I/O-optimized configuration offers improved performance, increased throughput, and reduced latency for customers with I/O intensive workloads. If you expect your I/O costs to exceed 25% of your total Amazon DocumentDB cluster costs, this option offers enhanced price performance. With the Amazon DocumentDB I/O-optimized storage configuration, you won't be charged based on I/O operations, ensuring predictable costs each billing cycle. The configuration stabilizes costs while improving performance.

You can switch your existing database clusters once every 30 days to Amazon DocumentDB I/O-optimized storage. You can switch back to Amazon DocumentDB standard storage at any time. The next date to modify the storage configuration to I/O-optimized can be tracked with the `describe-db-clusters` command using the AWS CLI or through the AWS Management Console in the cluster's configuration page.

You can create a new database cluster including the Amazon DocumentDB I/O-optimized configuration or convert your existing database clusters with a few clicks in the AWS Management Console, a single parameter change in the AWS Command Line Interface (AWS CLI), or through AWS SDKs. There is no downtime or reboot of instances required during or after modifying the storage configuration.

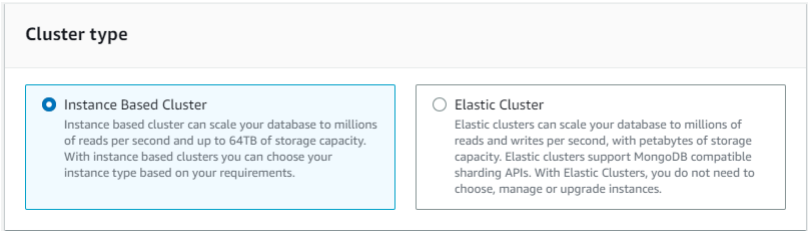


Creating an I/O-optimized cluster

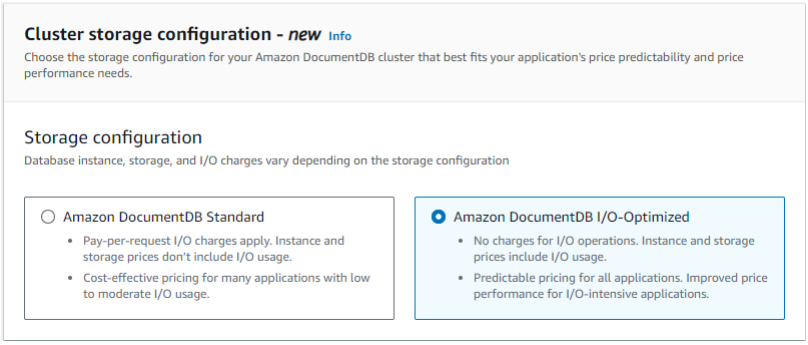
Using the AWS Management Console

To create or modify an I/O-optimized cluster using the AWS Management Console:

1. On the Amazon DocumentDB management console, under **Clusters**, choose either **Create** or select the cluster and choose **Actions**, and then choose **Modify**.
2. If you are creating a new cluster, make sure you choose **Instance Based Cluster** in the **Cluster type** section (this is the default option).



3. In the **Configuration** section, under **Cluster storage configuration**, choose **Amazon DocumentDB I/O-Optimized**.



4. Complete your cluster creation or modification and choose **Create cluster** or **Modify cluster**.

   For the complete Create cluster process, see Creating a cluster and primary instance using the AWS Management Console.

   For the complete Modify cluster process, see Modifying an Amazon DocumentDB cluster.

Using the AWS CLI

To create an I/O-optimized cluster using the AWS CLI:

In the following examples, replace each *user input placeholder* with your own information.

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
      --db-cluster-identifier sample-cluster \
      --engine docdb \
      --engine-version 5.0.0 \
      --storage-type iopt1 \
      --deletion-protection \
      --master-username username \
      --master-user-password password
```

For Windows:

```
aws docdb create-db-cluster ^
      --db-cluster-identifier sample-cluster ^
      --engine docdb ^
      --engine-version 5.0.0 ^
      --storage-type iopt1 ^
      --deletion-protection ^
      --master-username username ^
      --master-user-password password
```

Cost analysis for determining storage configuration

With Amazon DocumentDB, you have the flexibility to choose your storage configuration for every database cluster you have. In order to properly allocate your clusters between standard and I/O-optimized, you can track your Amazon DocumentDB costs cluster-wise. To do so, you can add tags to existing clusters, enable cost allocation tagging in your AWS Billing and Cost Management dashboard, and analyze your costs for a given cluster in the AWS Cost Explorer Service. For information on cost analysis, see our blog Using cost allocation tags.

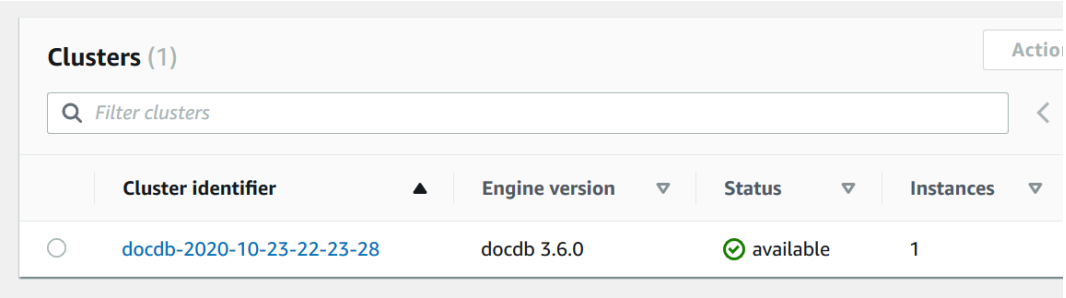
Determining a cluster's status

You can determine a cluster's status using the AWS Management Console or AWS CLI.

Using the AWS Management Console

Use the following procedure to see the status of your Amazon DocumentDB cluster using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. In the **Cluster identifier** column, find the name of the cluster that you are interested in. Then, to find the status of the cluster, read across that row to the **Status** column, as shown below.



Using the AWS CLI

Use the `describe-db-clusters` operation to see the the status of your Amazon DocumentDB cluster using the AWS CLI.

The following code finds the status of the cluster `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster  \
    --query 'DBClusters[*].[DBClusterIdentifier,Status]'
```

For Windows:

```
aws docdb describe-db-clusters ^
    --db-cluster-identifier sample-cluster  ^
    --query 'DBClusters[*].[DBClusterIdentifier,Status]'
```

Output from this operation looks something like the following (JSON format).

```
[
    [
        "sample-cluster",
        "available"
    ]
]
```

Amazon DocumentDB cluster lifecycle

The lifecycle of an Amazon DocumentDB cluster includes creating, describing, modifying, and deleting the cluster. This section provides information about how to complete these processes.

Topics

- Creating an Amazon DocumentDB cluster
- Describing Amazon DocumentDB clusters
- Modifying an Amazon DocumentDB cluster
- Determining pending maintenance
- Performing a patch update to a cluster's engine version
- Updating a cluster's operating system
- Stopping and starting an Amazon DocumentDB cluster
- Deleting an Amazon DocumentDB cluster

Creating an Amazon DocumentDB cluster

An Amazon DocumentDB cluster consists of instances and a cluster volume that represents the data for the cluster. The cluster volume is replicated six ways across three Availability Zones as a single, virtual volume. The cluster contains a primary instance and, optionally, up to 15 replica instances.

The following sections show how to create an Amazon DocumentDB cluster using either the AWS Management Console or the AWS CLI. You can then add additional replica instances for that cluster. When you use the console to create your Amazon DocumentDB cluster, a primary instance is automatically created for you at the same time. If you use the AWS CLI to create your Amazon DocumentDB cluster, after the cluster's status is *available*, you must then create the primary instance for that cluster.

Prerequisites

The following are prerequisites for creating an Amazon DocumentDB cluster.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://signin.aws.amazon.com/signup?request_type=register.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

VPC prerequisites

You can only create an Amazon DocumentDB cluster in an Amazon Virtual Private Cloud (Amazon VPC). Your Amazon VPC must have at least one subnet in each of at least two Availability Zones in order for you to use it with an Amazon DocumentDB cluster. By distributing your cluster instances across Availability Zones, you ensure that instances are available in your cluster in the unlikely case of an Availability Zone failure.

Subnet prerequisites

When creating an Amazon DocumentDB cluster, you must choose a VPC and corresponding subnet group within that VPC to launch your cluster. Subnets determine the Availability Zone and IP range within that Availability Zone that you want to use to launch an instance. For the purposes of this discussion, the terms *subnet* and *Availability Zone* are used interchangeably. A subnet group is a named set of subnets (or Availability Zones). What a subnet group allows you to do is specify the Availability Zones that you want to use to for launching Amazon DocumentDB instances. For example, in a cluster with three instances, it is recommended for high availability that each of those instances is provisioned in separate Availability Zones. Thus, if a single Availability Zone goes down, it only affects a single instance.

Amazon DocumentDB instances can currently be provisioned in up to three Availability Zones. Even if a subnet group has more than three subnets, you can only use three of those subnets to create an Amazon DocumentDB cluster. As a result, it is suggested that when you create a subnet group, only choose the three subnets that you want to deploy your instances to. In US East (N. Virginia), your subnet group can have six subnets (or Availability Zones). However, when an Amazon DocumentDB cluster is provisioned, Amazon DocumentDB chooses three of those Availability Zones that it uses to provision instances.

For example, suppose that when you are creating a cluster, Amazon DocumentDB chooses the Availability Zones {1A, 1B, and 1C}. If you try to create an instance in Availability Zone {1D}, the API call fails. However, if you choose to create an instance without specifying a particular Availability Zone, then Amazon DocumentDB chooses an Availability Zone on your behalf. Amazon DocumentDB uses an algorithm to load balance the instances across Availability Zones to help you achieve high availability. For example, if three instances are provisioned, by default, they are provisioned across three Availability Zones and are not provisioned all in a single Availability Zone.

Recommendations:

- Unless you have a specific reason, always create a subnet group with three subnets. Doing so helps ensure that clusters with three or more instances can achieve higher availability as instances are provisioned across three Availability Zones.
- Always spread instances across multiple Availability Zones to achieve high availability. Never place all instances for a cluster in a single Availability Zone.
- Because failover events can happen at any time, you should not assume that a primary instance or replica instances are always in a particular Availability Zone.

Additional prerequisites

The following are some additional prerequisites for creating an Amazon DocumentDB cluster:

- If you are connecting to AWS using AWS Identity and Access Management (IAM) credentials, your IAM account must have IAM policies that grant the permissions that are required to perform Amazon DocumentDB operations.

  If you are using an IAM account to access the Amazon DocumentDB console, you must first sign in to the AWS Management Console with your IAM account. Then go to the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

- If you want to tailor the configuration parameters for your cluster, you must specify a cluster parameter group and parameter group with the required parameter settings. For information about creating or modifying a cluster parameter group or parameter group, see Managing Amazon DocumentDB cluster parameter groups.
- You must determine the TCP/IP port number that you want to specify for your cluster. The firewalls at some companies block connections to the default ports for Amazon DocumentDB. If your company firewall blocks the default port, choose another port for your cluster. All instances in a cluster use the same port.

Creating a cluster and primary instance using the AWS Management Console

The following procedures describe how to use the console to launch an Amazon DocumentDB cluster with one or more instances.

Create a cluster: using default settings

To create a cluster with instances using the default settings using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. If you want to create your cluster in an AWS Region other than the US East (N. Virginia) Region, choose the Region from the list in the upper-right section of the console.
3. In the navigation pane, choose **Clusters**, and then choose **Create**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

4. On the **Create Amazon DocumentDB cluster** page, complete the **Configuration** pane.

   a. **Cluster identifier**—Accept the Amazon DocumentDB provided name, or enter a name for your cluster; for example, **`sample-cluster`**.

   Cluster naming constraints:

   - Length is [1—63] letters, numbers, or hyphens.

- First character must be a letter.
- Cannot end with a hyphen or contain two consecutive hyphens.
- Must be unique for all clusters across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.

b. **Engine version** – Keep the default version, or choose another available version. For more information, see Amazon DocumentDB cluster settings.

c. **Instance class**—Accept the default `db.r5.large`, or choose the instance class that you want from the list.

d. **Number of instances**—In the list, choose the number of instances that you want to be created with this cluster. The first instance is the primary instance, and all other instances are read-only replica instances. You can add and delete instances later if you need to. By default, an Amazon DocumentDB cluster launches with three instances (one primary and two replicas).

5. Complete the **Cluster storage configuration** section.

   Choose either **Amazon DocumentDB Standard** (default) or **Amazon DocumentDB I/O-Optimized**. For more information, see Amazon DocumentDB cluster storage configurations.

6. Complete the **Authentication** pane.

   a. **Username**—Enter a name for the primary user. To log in to your cluster, you must use the primary user name.

      Primary user naming constraints:

      - Length is [1—63] alphanumeric characters.
      - First character must be a letter.
      - Cannot be a word reserved by the database engine.

   b. Choose one of the following password options:

      - **Managed in AWS Secrets Manager**—Choose this option if you want AWS Secrets Manager to automatically manage your primary user password.

        If you choose this option, configure the KMS key by either creating your own or using a key that Secrets Manager creates.

- **Self managed**—Choose this option if you want to self-manage your primary user password. If you choose this option, enter a password for the primary user, and then confirm it. To log in to your cluster, you must use the password for the primary user.

  Password constraints:

  - Length is [8-100] printable ASCII characters.
  - Can use any printable ASCII characters except for the following:
    - **/** (forward slash)
    - **"** (double quotation mark)
    - **@** (at symbol)

7. At the bottom of the screen, choose one of the following:
   - To create the cluster now, choose **Create cluster**.
   - To not create the cluster, choose **Cancel**.
   - To further configure the cluster before creating, choose **Show additional configurations**, and then continue at Create a cluster: additional configurations.

     The configurations covered in the **Additional Configurations** section are as follows:

     - **Network settings**—The default is to use the `default` VPC security group.
     - **Cluster options**—The default is to use port is 27017 and the default parameter group.
     - **Encryption**—The default is to enable encryption using the `(default) aws/rds` key.

       > **Important**
       >
       > After a cluster is encrypted, it cannot be unencrypted.

     - **Backup**—The default is to retain backups for 1 day and let Amazon DocumentDB choose the backup window.
     - **Log exports**—The default is to not export audit logs to CloudWatch Logs.
     - **Maintenance**—The default is to let Amazon DocumentDB choose the maintenance window.
     - **Deletion protection**—Protect your cluster from accidental deletion. Default for cluster created using the console is *enabled*.

If you accept the default settings now, you can change most of them later by modifying the cluster.

8. Enable inbound connection for your cluster's security group.

   If you did not change the defaults settings for your cluster, you created a cluster using the default security group for the default VPC in the given Region. To connect to Amazon DocumentDB, you must enable inbound connections on port 27017 (or the port of your choice) for your cluster's security group.

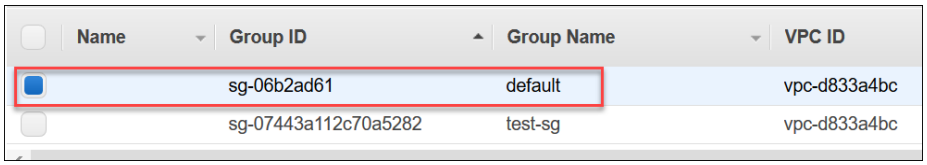
   To add an inbound connection to your cluster's security group

   a. Sign in to the AWS Management Console and open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.

   b. In the **Resources** section of the main window, choose **Security groups**.



   c. From the list of security groups locate the security group you used when creating your cluster (it is most likely the *default* security group) and choose the box to the left of the security group's name.



   d. From the **Actions** menu, choose **Edit inbound rules** then choose or enter the rule constraints.

      i. **Type**—From the list, choose the protocol to open to network traffic.

      ii. **Protocol**—From the list, choose the type of protocol.

      iii. **Port Range**—For a custom rule, enter a port number or port range. Be sure that the port number or range includes the port you specified when you created your cluster (default: 27017).

iv. **Source**—Specifies the traffic that can reach your instance. From the list, choose the traffic source. If you choose **Custom**, specify a single IP address or an IP address range in CIDR notation (e.g., 203.0.113.5/32).

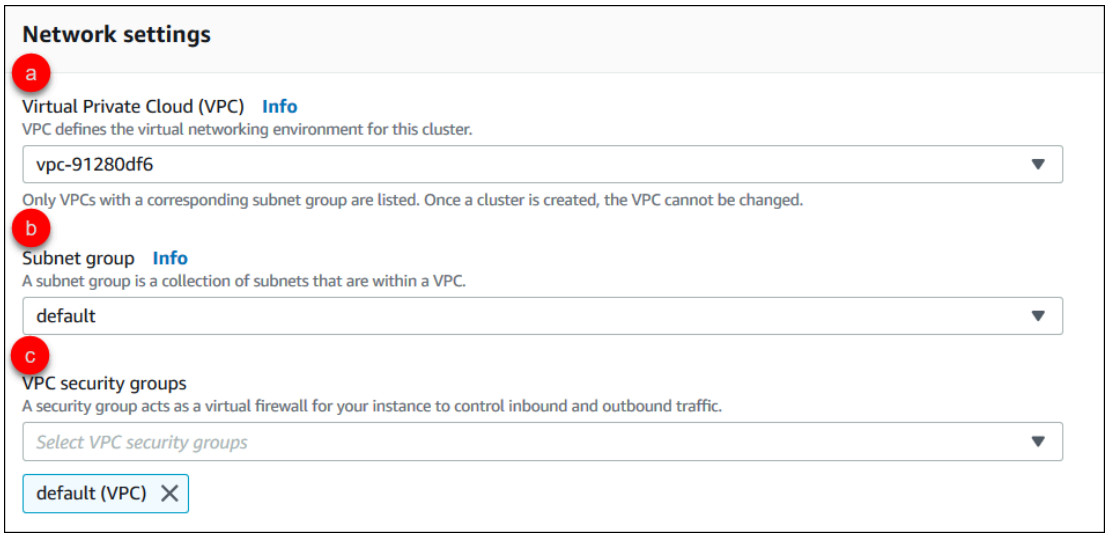
v. **Description**—Enter a description for this rule.

vi. When finished creating the rule, choose **Save**.

Create a cluster: additional configurations

If you want to accept the default settings for your cluster, you can skip the following steps and choose **Create cluster**.

1. Complete the **Network settings** pane.

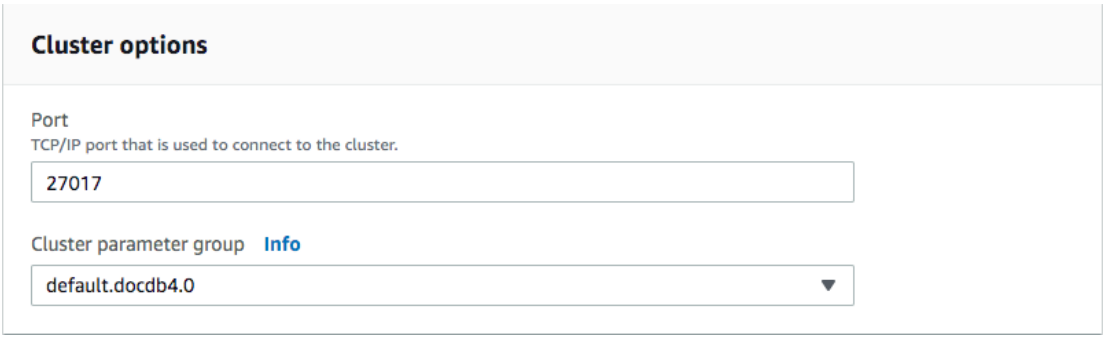


   a. **Virtual Private Cloud (VPC)**—In the list, choose the Amazon VPC that you want to launch this cluster in.

   b. **Subnet group**—In the list, choose the subnet group you want to use for this cluster.

   c. **VPC security groups**—In the list, choose the VPC security group for this cluster.

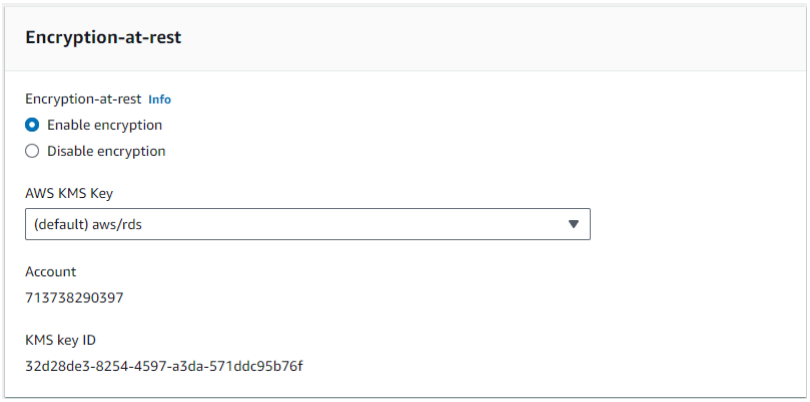
2. Complete the **Cluster options** pane.



a. **Data base port**—Use the up and down arrows to set the TCP/IP port that applications will use to connect to your instance.

b. **Cluster parameter group**—In the list of parameter groups, choose the cluster parameter group for this cluster.

3. Complete the **Encryption** pane.



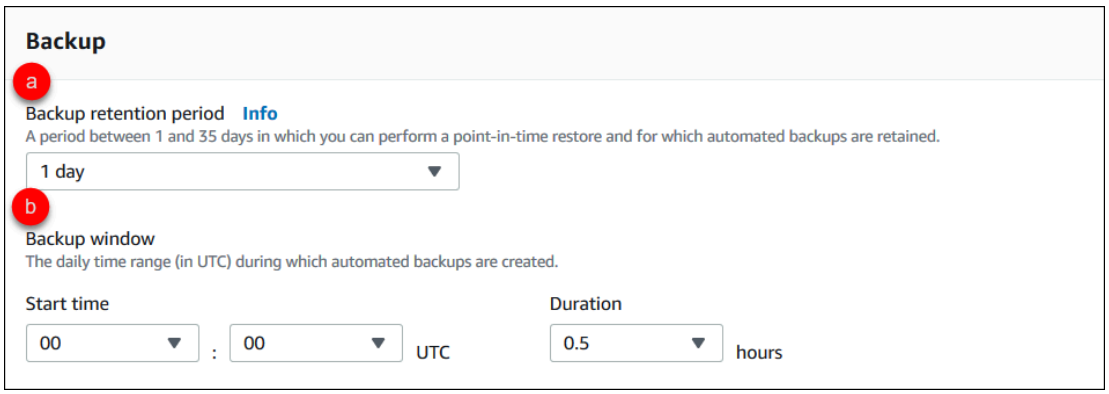
a. **Encryption-at-rest**—Choose one of the following:

- **Enable encryption**—Default. All data at rest is encrypted. If you choose to encrypt your data, you cannot undo this action.
- **Disable encryption**—Your data is not encrypted.

b. **AWS KMS key**—This is only available if you are encrypting your data. In the list, choose the key that you want to use for encrypting the data in this cluster. The default is `(default) aws/rds`.

   If you chose **Enter a key ARN**, you must enter an Amazon Resource Name (ARN) for the key.

4. Complete the **Backup** pane.



a. **Backup retention period**—In the list, choose the number of days to keep automatic backups of this cluster before deleting them.

b. **Backup window**—Set the daily time and duration during which Amazon DocumentDB is to make backups of this cluster.

   i. **Start time**—In the first list, choose the start time hour (UTC) for starting your automatic backups. In the second list, choose the minute of the hour that you want automatic backups to begin.

   ii. **Duration**—In the list, choose the number of hours to be allocated to creating automatic backups.

5. Complete the **Log exports** pane by selecting the types of logs you want to export to CloudWatch Logs.

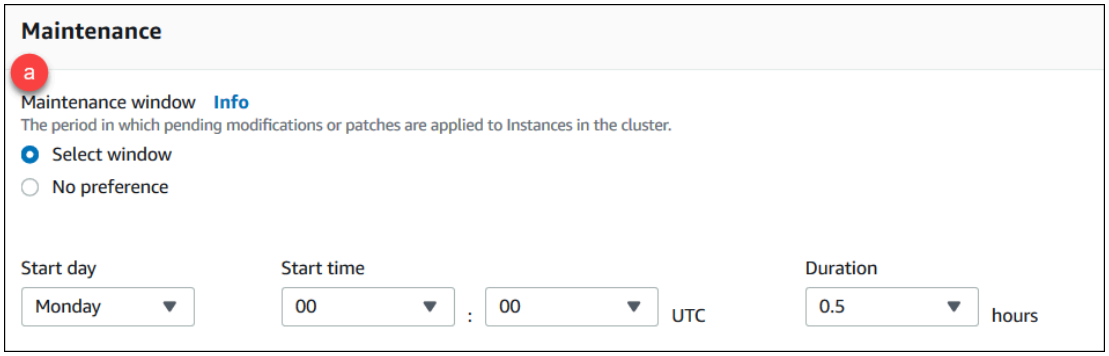


- **Audit logs**—Select this option to enable exporting audit logs to Amazon CloudWatch Logs. If you select **Audit logs**, you must enable `audit_logs` in the cluster's custom parameter group. For more information, see Auditing Amazon DocumentDB events.
- **Profiler logs**—Select this option to enable exporting operation profiler logs to Amazon CloudWatch Logs. If you select **Profiler logs**, you must also modify the following parameters in the cluster's custom parameter group:
  - `profiler`—Set to `enabled`.
  - `profiler_threshold_ms`—Set to a value `[0-INT_MAX]` to set the threshold for profiling operations.

- `profiler_sampling_rate`—Set to a value `[0.0-1.0]` to set the percentage of slow operations to profile.

  For more information, see Profiling Amazon DocumentDB operations.

6. Complete the **Maintenance** pane.



   - Choose one of the following

     - **Select window**—You can specify the day of the week, UTC start time, and duration for Amazon DocumentDB to perform maintenance on your cluster.

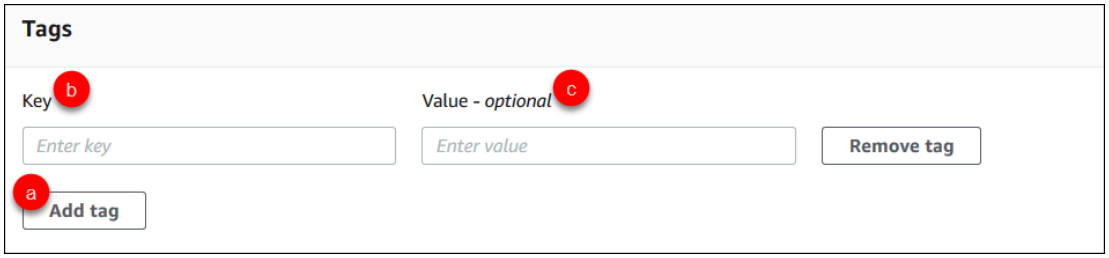
       a. **Start day**—In the list, choose the day of the week to start cluster maintenance.

       b. **Start time**—In the lists, choose the hour and minute (UTC) to start maintenance.

       c. **Duration**—In the list, choose how much time to allocate for cluster maintenance. If maintenance cannot be finished in the specified time, the maintenance process will continue past the specified time until finished.

     - **No preference**—Amazon DocumentDB chooses the day of the week, start time, and duration for performing maintenance.

7. If you want to add one or more tags to this cluster, complete the **Tags** pane.



   For each tag you want to add to the cluster, repeat the following steps. You may have up to 10 on a cluster.

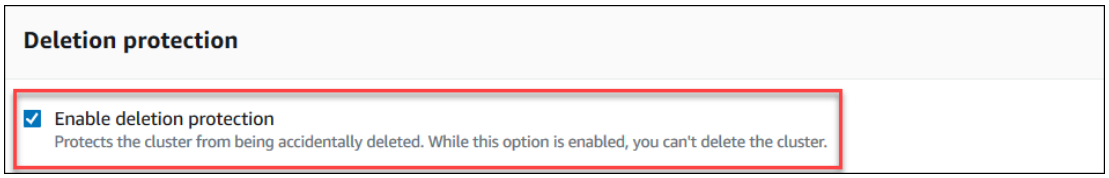
a. Choose **Add tags**.

b. Type the tag's **Key**.

c. Optionally type the tag's **Value**.

To remove a tag, choose **Remove tag**.

8. **Deletion Protection** is enabled by default when you create a cluster using the console. To disable deletion protection, clear **Enable deletion protection**. When enabled, deletion protection prevents a cluster from being deleted. To delete a deletion protected cluster, you must first modify the cluster to disable deletion protection.



For more information about deletion protection, see Deleting an Amazon DocumentDB cluster.

9. To create the cluster, choose **Create cluster**. Otherwise, choose **Cancel**.

Creating a cluster using the AWS CLI

The following procedures describe how to use the AWS CLI to launch an Amazon DocumentDB cluster and create an Amazon DocumentDB replica.

Parameters

- **`--db-cluster-identifier`**—Required. A lowercase string that identifies this cluster.

  Cluster Naming Constraints:

  - Length is [1–63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all clusters (across Amazon RDS, Amazon Neptune, and Amazon DocumentDB) per AWS account, per Region.

- **`--engine`**—Required. Must be **`docdb`**.

- **`--deletion-protection | --no-deletion-protection`**—Optional. When deletion protection is enabled, it prevents a cluster from being deleted. When you use the AWS CLI, the default setting is to have deletion protection disabled.

  For more information about deletion protection, see Deleting an Amazon DocumentDB cluster.

- **`--storage-type standard | iopt1`**—Optional. Default: **`standard`**. The cluster's storage configuration. Valid values are `standard` (Standard) or `iopt1` (I/O-optimized).
- **`--master-username`**—Required. The user name used to authenticate the user.

  Master User Naming Constraints:

  - Length is [1-63] alphanumeric characters.
  - First character must be a letter.
  - Cannot be a word reserved by the database engine.

- **`--master-user-password`**—Optional. The user's password used to authenticate the user.

  Master Password Constraints:

  - Length is [8-100] printable ASCII characters.
  - Can use any printable ASCII characters except for the following:
    - **`/`** (forward slash)
    - **`"`** (double quotation mark)
    - **`@`** (at symbol)

- **`--manage-master-user-password`**—Optional. Amazon DocumentDB generates the master user password and manages it throughout its lifecycle in Secrets Manager.

For additional parameters, see CreateDBCluster.

To launch an Amazon DocumentDB cluster using the AWS CLI

To create an Amazon DocumentDB cluster, call the `create-db-cluster` AWS CLI. The following AWS CLI command creates an Amazon DocumentDB cluster named `sample-cluster` with deletion protection enabled. For more information on deletion protection, see Deleting an Amazon DocumentDB cluster.

Also, `--engine-version` is an optional parameter. If you omit it, Amazon DocumentDB uses the newest 8.0 minor version available in the AWS Region. To use a specific minor version, specify it with `--engine-version`. For more information, see Amazon DocumentDB cluster settings.

> **Note**
>
> If a `db-subnet-group-name` or `vpc-security-group-id` is not specified, Amazon DocumentDB will use the default subnet group and Amazon VPC security group for the given Region.

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
      --db-cluster-identifier sample-cluster \
      --engine docdb \
      --engine-version 5.0.0 \
      --deletion-protection \
      --master-username masteruser \
      --master-user-password password
```

For Windows:

```
aws docdb create-db-cluster ^
      --db-cluster-identifier sample-cluster ^
      --engine docdb ^
      --engine-version 5.0.0 ^
      --deletion-protection ^
      --master-username masteruser ^
      --master-user-password password
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBCluster": {
        "StorageEncrypted": false,
        "DBClusterMembers": [],
        "Engine": "docdb",
        "DeletionProtection" : "enabled",
        "ClusterCreateTime": "2018-11-26T17:15:19.885Z",
        "DBSubnetGroup": "default",
```

```
        "EngineVersion": "5.0.0",
        "MasterUsername": "masteruser",
        "BackupRetentionPeriod": 1,
        "DBClusterArn": "arn:aws:rds:us-east-1:123456789012:cluster:sample-cluster",
        "DBClusterIdentifier": "sample-cluster",
        "MultiAZ": false,
        "DBClusterParameterGroup": "default.docdb5.0",
        "PreferredBackupWindow": "09:12-09:42",
        "DbClusterResourceId": "cluster-KQSGI4MHU4NTDDRVNLNTU7XVAY",
        "PreferredMaintenanceWindow": "tue:04:17-tue:04:47",
        "Port": 27017,
        "Status": "creating",
        "ReaderEndpoint": "sample-cluster.cluster-ro-sfcrlcjcoroz.us-
east-1.docdb.amazonaws.com",
        "AssociatedRoles": [],
        "HostedZoneId": "ZNKXTT8WH85VW",
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-77186e0d",
                "Status": "active"
            }
        ],
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1c",
            "us-east-1e"
        ],
        "Endpoint": "sample-cluster.cluster-sfcrlcjcoroz.us-east-1.docdb.amazonaws.com"
    }
}
```

It takes several minutes to create the cluster. You can use the AWS Management Console or AWS CLI to monitor the status of your cluster. For more information, see Monitoring an Amazon DocumentDB cluster's status.

Important

When you use the AWS CLI to create an Amazon DocumentDB cluster, no instances are created. Consequently, you must explicitly create a primary instance and any replica instances that you need. You can use either the console or AWS CLI to create the instances. For more information, see Adding an Amazon DocumentDB instance to a cluster.

For more information, see CreateDBCluster in the *Amazon DocumentDB API Reference*.

Describing Amazon DocumentDB clusters

You can use either Amazon DocumentDB Management Console or the AWS CLI to see details such as connection endpoints, security groups, VPCs, and parameter groups pertaining to your Amazon DocumentDB clusters.

For more information, see the following:

- Monitoring an Amazon DocumentDB cluster's status
- Finding a cluster's endpoints

Using the AWS Management Console

Use the following procedure to view the details of a specified Amazon DocumentDB cluster using the console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

   Tip

   If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the list of clusters, choose the name of the cluster that you want to see the details of. The information about the cluster is organized into the following groupings:

   - **Summary** — General information about the cluster, including the engine version, cluster status, pending maintenance, and the status of its parameter group.
   - **Connectivity & Security** —The **Connect** section lists connection endpoints to connect to this cluster with the mongo shell or with an application. The **Security Groups** section lists the security groups associated with this cluster and their VPC ID and descriptions.
   - **Configuration** — The **Cluster details** section lists details about the cluster, including the cluster's Amazon Resource Name (ARN), endpoint, and parameter group. It also lists the

cluster's backup information, maintenance details, and security and network settings. The **Cluster instances** section lists the instances that belong to this cluster with each instance's role and cluster parameter group status.

- **Monitoring** — The Amazon CloudWatch Logs metrics for this cluster. For more information, see Monitoring Amazon DocumentDB with CloudWatch.
- **Events & tags** — The **Recent events** section lists the recent events for this cluster. Amazon DocumentDB keeps a record of events that relate to your clusters, instances, snapshots, security groups, and cluster parameter groups. This information includes the date, time, and message associated with each event. The **Tags** section lists the tags attached to this cluster.

Using the AWS CLI

To view the details of your Amazon DocumentDB clusters using the AWS CLI, use the `describe-db-clusters` command as shown in the following examples. For more information, see DescribeDBClusters in the *Amazon DocumentDB Resource Management API Reference*.

> **Note**
>
> For certain management features such as cluster and instance lifecycle management, Amazon DocumentDB leverages operational technology that is shared with Amazon RDS. The `filterName=engine,Values=docdb` filter parameter returns only Amazon DocumentDB clusters.

Example

Example 1: List all Amazon DocumentDB clusters

The following AWS CLI code lists the details for all Amazon DocumentDB clusters in a Region.

```
aws docdb describe-db-clusters --filter Name=engine,Values=docdb
```

Output from this operation looks something like the following.

```
{
    "DBClusters": [
        {
```

```
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-1",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        },
        {
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-2",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        },
        {
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1b",
                "us-east-1a"
            ],
            "BackupRetentionPeriod": 1,
            "DBClusterIdentifier": "sample-cluster-3",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            ...
        }
    ]
}
```

Example

Example 2: List all details for a specified Amazon DocumentDB cluster

The following AWS CLI code lists the details for the cluster `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
   --filter Name=engine,Values=docdb \
   --db-cluster-identifier sample-cluster
```

For Windows:

```
aws docdb describe-db-clusters ^
   --filter Name=engine,Values=docdb ^
   --db-cluster-identifier sample-cluster
```

Output from this operation looks something like the following.

```
{
    "DBClusters": [
        {
            "AllocatedStorage": 1,
            "AvailabilityZones": [
                "us-east-1c",
                "us-east-1a",
                "us-east-1d"
            ],
            "BackupRetentionPeriod": 2,
            "DBClusterIdentifier": "sample-cluster",
            "DBClusterParameterGroup": "sample-parameter-group",
            "DBSubnetGroup": "default",
            "Status": "available",
            "EarliestRestorableTime": "2023-11-07T22:34:08.148000+00:00",
            "Endpoint": "sample-cluster.node.us-east-1.amazon.com",
            "ReaderEndpoint": "sample-cluster.node.us-east-1.amazon.com",
            "MultiAZ": false,
            "Engine": "docdb",
            "EngineVersion": "5.0.0",
            "LatestRestorableTime": "2023-11-10T07:21:16.772000+00:00",
            "Port": 27017,
            "MasterUsername": "chimeraAdmin",
            "PreferredBackupWindow": "22:22-22:52",
            "PreferredMaintenanceWindow": "sun:03:01-sun:03:31",
            "ReadReplicaIdentifiers": [],
            "DBClusterMembers": [
```

```
                {
                    "DBInstanceIdentifier": "sample-instance-1",
                    "IsClusterWriter": true,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "PromotionTier": 1
                },
                {
                    "DBInstanceIdentifier": "sample-instance-2",
                    "IsClusterWriter": true,
                    "DBClusterParameterGroupStatus": "in-sync",
                    "PromotionTier": 1
                },

            ],
            "VpcSecurityGroups": [
                {
                    "VpcSecurityGroupId": "sg-9084c2ec",
                    "Status": "active"
                }
            ],
            "HostedZoneId": "Z06853723JYKYBXTJ49RB",
            "StorageEncrypted": false,
            "DbClusterResourceId": "cluster-T4LGLANHVAPGQYYULWUDKLVQL4",
            "DBClusterArn": "arn:aws:rds:us-east-1:123456789012:cluster:sample-
cluster",
            "AssociatedRoles": [],
            "IAMDatabaseAuthenticationEnabled": false,
            "ClusterCreateTime": "2023-11-06T18:05:41.568000+00:00",
            "EngineMode": "provisioned",
            "DeletionProtection": false,
            "HttpEndpointEnabled": false,
            "CopyTagsToSnapshot": false,
            "CrossAccountClone": false,
            "DomainMemberships": [],
            "TagList": [],
            "StorageType": "iopt1",
            "AutoMinorVersionUpgrade": false,
            "NetworkType": "IPV4",
            "IOOptimizedNextAllowedModificationTime":
 "2023-12-07T18:05:41.580000+00:00"
        }
    ]
}
```

Example

Example 3: List specific details for a Amazon DocumentDB cluster

To list a subset of the clusters' details using the AWS CLI, add a `--query` that specifies which cluster members the `describe-db-clusters` operation is to list. The `--db-cluster-identifier` parameter is the identifier for the particular cluster that you want to display the details of. For more information on queries, see How to Filter the Output with the `--query` Option in the *AWS Command Line Interface User Guide*.

The following example lists the instances in an Amazon DocumentDB cluster.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
    --filter Name=engine,Values=docdb \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].[DBClusterMembers]'
```

For Windows:

```
aws docdb describe-db-clusters ^
    --filter Name=engine,Values=docdb ^
    --db-cluster-identifier sample-cluster ^
    --query 'DBClusters[*].[DBClusterMembers]'
```

Output from this operation looks something like the following.

```
[
    [
        [
            {
                "DBInstanceIdentifier": "sample-instance-1",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            },
            {
                "DBInstanceIdentifier": "sample-instance-2",
                "IsClusterWriter": false,
                "DBClusterParameterGroupStatus": "in-sync",
```

```
                "PromotionTier": 1
            }
        ]
    ]
]
```

Modifying an Amazon DocumentDB cluster

To modify a cluster, the cluster must be in the *available* state. You cannot modify a cluster that is stopped. If the cluster is stopped, first start the cluster, wait for the cluster to become *available*, and then make the desired modifications. For more information, see Stopping and starting an Amazon DocumentDB cluster.

Using the AWS Management Console

Use the following procedure to modify a specific Amazon DocumentDB cluster using the console.

To modify an Amazon DocumentDB cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. Specify the cluster that you want to modify by choosing the button to the left of the cluster's name.
4. Choose **Actions**, and then choose **Modify**.
5. In the **Modify Cluster: <cluster-name>** pane, make the changes that you want. You can make changes in the following areas:

   - **Cluster specifications**—The cluster's name, security groups, and credentials management.

- **Cluster storage configuration**—The cluster's data storage mode. Choose between Standard and I/O-Optimized configuration.
- **Cluster options**—The cluster's port and parameter group.
- **Backup**—The cluster's backup retention period and backup window.
- **Log exports**—Enable or disable exporting audit or profiler logs.
- **Maintenance**—Set the cluster's maintenance window.
- **Deletion protection**—Enable or disable deletion protection on the cluster. Deletion protection is enabled by default.

6. When you're finished, choose **Continue** to view a summary of your changes.
7. If you are satisfied with your changes, you can choose **Modify cluster** to modify your cluster. Alternatively, you can choose **Back** or **Cancel** to edit or cancel your changes, respectively.

It takes a few minutes for your changes to be applied. You can use the cluster only when its status is *available*. You can monitor the cluster's status using the console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB cluster's status.

Using the AWS CLI

Use the `modify-db-cluster` operation to modify the specified cluster using the AWS CLI. For more information, see ModifyDBCluster in the *Amazon DocumentDB API Reference*.

Parameters

- **`--db-cluster-identifier`**—Required. The identifier of the Amazon DocumentDB cluster that you are going to modify.
- **`--backup-retention-period`**—Optional. The number of days for which automated backups are retained. Valid values are 1–35.
- **`--storage-type`**—Optional. The cluster's storage configuration. Valid values are `standard` (Standard) or `iopt1` (I/O-optimized).
- **`--db-cluster-parameter-group-name`**—Optional. The name of the cluster parameter group to use for the cluster.
- **`--manage-master-user-password`**—Optional. Amazon DocumentDB generates the master user password and manages it throughout its lifecycle in Secrets Manager.
- **`--rotate-master-user-password`**—Optional. Secrets Manager generates a new secret version for the existing secret. The new version of the secret contains the new primary user

password. Amazon DocumentDB changes the primary user password for the cluster to match the password for the new secret version.

You must specify the **`--apply-immediately`** option when you rotate the primary password.

- **`--master-user-password`**—Optional. The new password for the primary database user.

  Password constraints:

  - Length is [8—100] printable ASCII characters.
  - Can use any printable ASCII characters except for the following:
    - **`/`** (forward slash)
    - **`"`** (double quotation mark)
    - **`@`** (at symbol)
- **`--new-db-cluster-identifier`**—Optional. The new cluster identifier for the cluster when renaming a cluster. This value is stored as a lowercase string.

  Naming constraints:

  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all clusters across Amazon RDS, Amazon Neptune, and Amazon DocumentDB per AWS account, per Region.
- **`--preferred-backup-window`**—Optional. The daily time range during which automated backups are created, in Universal Coordinated Time (UTC).
  - Format: `hh24:mm-hh24:mm`
- **`--preferred-maintenance-window`**—Optional. The weekly time range during which system maintenance can occur, in UTC.
  - Format: `ddd:hh24:mm-ddd:hh24:mm`
  - Valid days: `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, and `Sat`.
- **`--deletion-protection`** or **`--no-deletion-protection`**—Optional. Whether deletion protection should be enabled on this cluster. Deletion protection prevents a cluster from being accidentally deleted until the cluster is modified to disable deletion protection. For more information, see Deleting an Amazon DocumentDB cluster.

- **`--apply-immediately`** or **`--no-apply-immediately`**—Use `--apply-immediately` to make the change immediately. Use `--no-apply-immediately` to make the change during your cluster's next maintenance window.

Example

The following code changes the backup retention period for the cluster `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
       --db-cluster-identifier sample-cluster \
       --apply-immediately \
       --backup-retention-period 7
```

For Windows:

```
aws docdb modify-db-cluster ^
       --db-cluster-identifier sample-cluster ^
       --apply-immediately ^
       --backup-retention-period 7
```

Output from this operation looks something like the following.

```
{
    "DBCluster": {
        "BackupRetentionPeriod": 7,
        "DbClusterResourceId": "cluster-VDP53QEWST7YHM36TTXOPJT5YE",
        "Status": "available",
        "DBClusterMembers": [
            {
                "PromotionTier": 1,
                "DBClusterParameterGroupStatus": "in-sync",
                "DBInstanceIdentifier": "sample-cluster-instance",
                "IsClusterWriter": true
            }
        ],
        "ReadReplicaIdentifiers": [],
        "AvailabilityZones": [
            "us-east-1b",
            "us-east-1c",
            "us-east-1a"
```

```
        ],
        "ReaderEndpoint": "sample-cluster.cluster-ro-ctevjxdlur57.us-
east-1.rds.amazonaws.com",
        "DBClusterArn": "arn:aws:rds:us-east-1:123456789012:cluster:sample-cluster",
        "PreferredMaintenanceWindow": "sat:09:51-sat:10:21",
        "EarliestRestorableTime": "2018-06-17T00:06:19.374Z",
        "StorageEncrypted": false,
        "MultiAZ": false,
        "AssociatedRoles": [],
        "MasterUsername": "<your-master-user-name>",
        "DBClusterIdentifier": "sample-cluster",
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-77186e0d"
            }
        ],
        "HostedZoneId": "Z2SUY0A1719RZT",
        "LatestRestorableTime": "2018-06-18T21:17:05.737Z",
        "AllocatedStorage": 1,
        "Port": 27017,
        "Engine": "docdb",
        "DBClusterParameterGroup": "default.docdb3.4",
        "Endpoint": "sample-cluster.cluster-ctevjxdlur57.us-
east-1.rds.amazonaws.com",
        "DBSubnetGroup": "default",
        "PreferredBackupWindow": "00:00-00:30",
        "EngineVersion": "3.4",
        "ClusterCreateTime": "2018-06-06T19:25:47.991Z",
        "IAMDatabaseAuthenticationEnabled": false
    }
}
```

It takes a few minutes for your changes to be applied. You can use the cluster only when its status is *available*. You can monitor the cluster's status using the console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB cluster's status.

Determining pending maintenance

Maintenance most often involves updates to the following resources in your cluster:

- Operating system (OS) of instances in the cluster

- Engine version of the cluster

For more information about Amazon DocumentDB maintenance, see Maintaining Amazon DocumentDB.

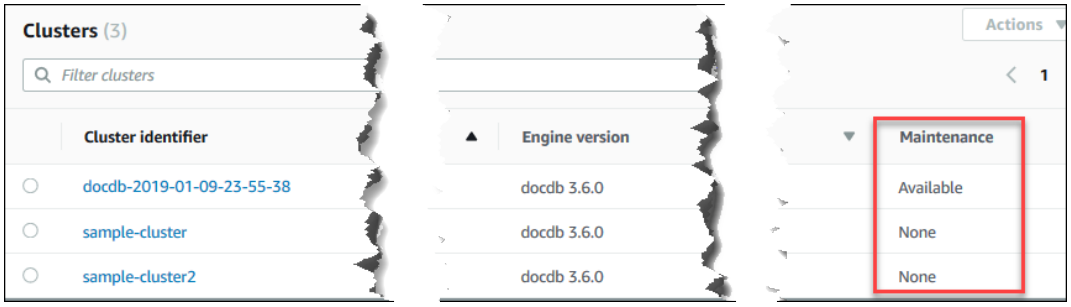
Using the AWS Management Console

You can use the AWS Management Console to determine whether a cluster has pending maintenance.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

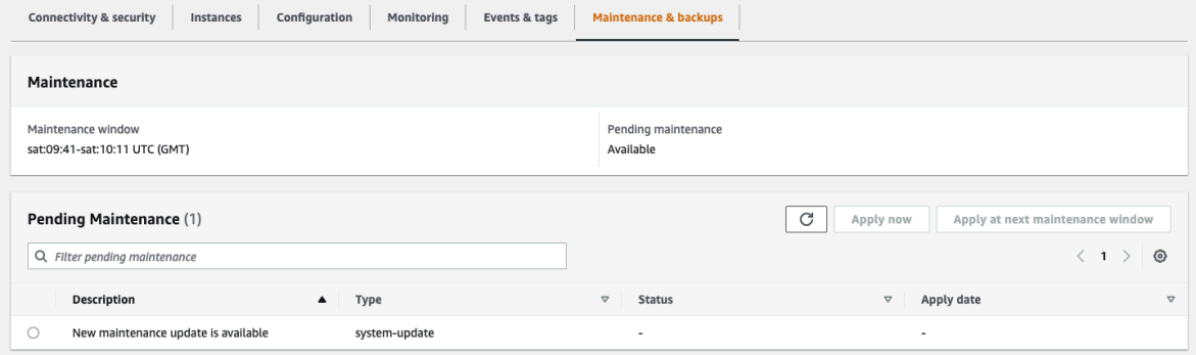
   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. Locate the **Maintenance** column to determine whether a cluster has pending maintenance.



   None indicates that the cluster is running the latest engine patch version and its instances have the latest OS updates. **Available** indicates that the cluster has pending maintenance, which might mean that an engine upgrade or OS update is needed.

4. If your cluster has pending maintenance, you can view the pending maintenance actions by choosing the cluster name and selecting the **Maintenance & backups** tab. The **Pending Maintenance** section lists the pending maintenance actions.



Using the AWS CLI

You can use the AWS CLI to determine whether a cluster has the latest engine version and OS updates by using the `describe-pending-maintenance-actions` operation with the following parameters.

Parameters

- **`--resource-identifier`**—Optional. The ARN for the resource (cluster). If this parameter is omitted, pending maintenance actions for all clusters are listed.
- **`--region`**—Optional. The AWS Region that you want to run this operation in, for example, `us-east-1`.

Example

For Linux, macOS, or Unix:

```
aws docdb describe-pending-maintenance-actions \
   --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster \
   --region aa-example-1
```

For Windows:

```
aws docdb describe-pending-maintenance-actions ^
   --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster ^
   --region aa-example-1
```

Output from this operation looks something like the following.

```
{
    "PendingMaintenanceActions": [
        {
            "ResourceIdentifier": "arn:aws:rds:aa-
example-1:111122223333:cluster:sample-cluster",
            "PendingMaintenanceActionDetails": [
                {
                    "Description": "New feature",
                    "Action": "system-update",
                    "ForcedApplyDate": "2019-02-25T21:46:00Z",
                    "AutoAppliedAfterDate": "2019-02-25T07:41:00Z",
                    "CurrentApplyDate": "2019-02-25T07:41:00Z"
                }
            ]
        }
    ]
}
```

The following maintenance action types apply to Amazon DocumentDB clusters:

- `system-update` — Upgrade the engine patch for the Amazon DocumentDB cluster.

  If your cluster has a pending engine patch upgrade available, continue with the steps at Performing a patch update to a cluster's engine version.
- `os-upgrade` — Update the operating systems of all the DB instances in the Amazon DocumentDB cluster, using rolling upgrades.

  For more information, see Amazon DocumentDB operating system updates.
- `serverless-platform-version-update` — Update the platform version of all the serverless instances in the Amazon DocumentDB cluster, using rolling upgrades.

  For more information, see Checking and upgrading the platform version of a serverless cluster.

Performing a patch update to a cluster's engine version

In this section, we will explain how to deploy a patch update using the AWS Management Console or the AWS CLI. A patch update is an update within the same engine version (for example, updating

a 3.6 engine version to a newer 3.6 engine version). You can update it immediately or during your cluster's next maintenance window. To determine whether your engine needs an update, see Determining pending maintenance. Note that when you apply the update, your cluster will experience some downtime.

> **Note**
>
> If you are trying to upgrade from a major engine version to another, such as 3.6 to 5.0 or 5.0 to 8.0, see either Amazon DocumentDB in-place major version upgrade or Upgrading your Amazon DocumentDB cluster using AWS Database Migration Service.

There are two configuration requirements to get the latest patch updates for a cluster's engine version:

- The cluster's status must be *available*.
- The cluster must be running an earlier engine version.

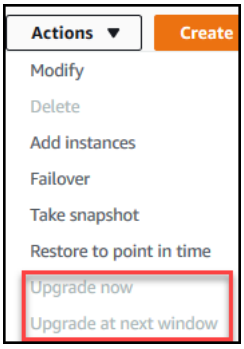
Using the AWS Management Console

The following procedure applies patch updates to your cluster's engine version using the console. You have the option to update immediately or during your cluster's next maintenance window.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**. In the list of clusters, choose the button to the left of the cluster that you want to upgrade. The status of the cluster must be *available*.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. From the **Actions** menu, choose one of the following options. These menu options are selectable only if the cluster you chose is not running the latest engine version.



- **Upgrade now**—Immediately initiates the upgrade process. Your cluster will be offline for a time while the cluster is upgraded to the latest engine version.
- **Upgrade at next window**—Initiates the upgrade process during the cluster's next maintenance window. Your cluster will be offline for a time while it is upgraded to the latest engine version.

4. When the confirmation window opens, choose one of the following:
   - **Upgrade**—To upgrade your cluster to the latest engine version according to the schedule chosen in the previous step.
   - **Cancel**—To cancel the cluster's engine upgrade and continue with the cluster's current engine version.

Using the AWS CLI

You can apply patch updates to your cluster using the AWS CLI and the `apply-pending-maintenance-action` operation with the following parameters.

Parameters

- **`--resource-identifier`**—Required. The ARN of the Amazon DocumentDB cluster that you are going to upgrade.
- **`--apply-action`**—Required. Use `system-update` to apply an engine patch update to the cluster.
- **`--opt-in-type`**—Required. The following values are permitted.
  - `immediate`—Apply the maintenance action immediately.
  - `next-maintenance`—Apply the maintenance action during the next maintenance window.
  - `undo-opt-in`—Cancel any existing `next-maintenance` opt-in requests.

Example

The following example applies the engine patch update to `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
   --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster \
   --apply-action system-update \
   --opt-in-type immediate
```

For Windows:

```
aws docdb apply-pending-maintenance-action ^
   --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster ^
   --apply-action system-update ^
   --opt-in-type immediate
```

Output from this operation looks like the following.

```
{
    "ResourcePendingMaintenanceActions": {
        "ResourceIdentifier": "arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster",
        "PendingMaintenanceActionDetails": [
            {
                "CurrentApplyDate": "2019-02-20T20:57:06.904Z",
                "Description": "Bug fixes",
                "ForcedApplyDate": "2019-02-25T21:46:00Z",
                "OptInStatus": "immediate",
                "Action": "system-update",
                "AutoAppliedAfterDate": "2019-02-25T07:41:00Z"
            }
        ]
    }
}
```

Updating a cluster's operating system

You can apply operating system (OS) updates across all instances in your cluster using the cluster-level `os-upgrade` maintenance action. Amazon DocumentDB updates instances in a rolling manner, a few at a time, and updates the primary instance last to minimize failovers. The update runs during the cluster maintenance window. After an instance receives an OS update, its buffer cache starts empty, and queries on that instance can experience higher latency until the working set is repopulated from the storage volume.

To determine whether your cluster has a pending OS update, see Determining pending maintenance. For full details on OS updates, including console and CLI procedures, see Amazon DocumentDB operating system updates.

Stopping and starting an Amazon DocumentDB cluster

Stopping and starting Amazon DocumentDB clusters can help you manage costs for development and test environments. Instead of creating and deleting clusters and instances each time you use Amazon DocumentDB, you can temporarily stop all the instances in your cluster when they aren't needed. You can then start them again when you resume your testing.

Topics

- Overview of stopping and starting a cluster
- Operations you can perform on a stopped cluster

Overview of stopping and starting a cluster

During periods where you don't need an Amazon DocumentDB cluster, you can stop all instances in that cluster at once. You can then start the cluster again anytime you need to use it. Starting and stopping simplifies the setup and teardown processes for clusters that are used for development, testing, or similar activities that don't require continuous availability. You can stop and start a cluster using the AWS Management Console or the AWS CLI with a single action, regardless of how many instances are in the cluster.

While your cluster is stopped, the cluster storage volume remains unchanged. You are charged only for storage, manual snapshots, and automated backup storage within your specified retention window. You aren't charged for any instance hours. Amazon DocumentDB automatically starts your cluster after seven days so that it doesn't fall behind any required maintenance updates. When your cluster starts after seven days, you will begin to be charged for the instances in the

cluster again. While your cluster is stopped, you can't query your storage volume because querying requires that instances are in the available state.

When an Amazon DocumentDB cluster is stopped, neither the cluster nor its instances can be modified in any way. This includes adding or removing instances, or deleting the cluster.

Using the AWS Management Console

The following procedure shows you how to stop a cluster with one or more instances in the available state, or start a stopped cluster.

To stop or start an Amazon DocumentDB cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

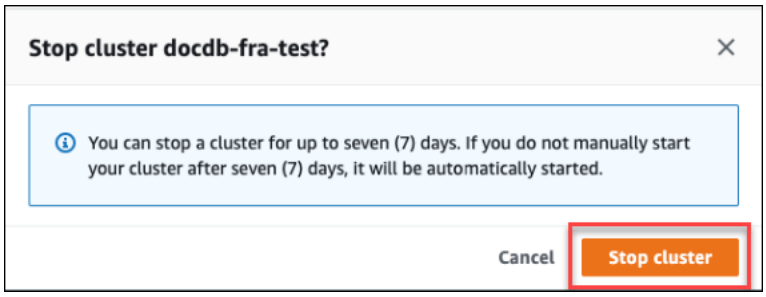
> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the list of clusters, choose the button to the left of the name of the cluster that you want to stop or start.
4. Choose **Actions**, and then choose the action that you want to perform on the cluster.
   - If you want to stop the cluster and the cluster is available:
     a. Choose **Stop**.

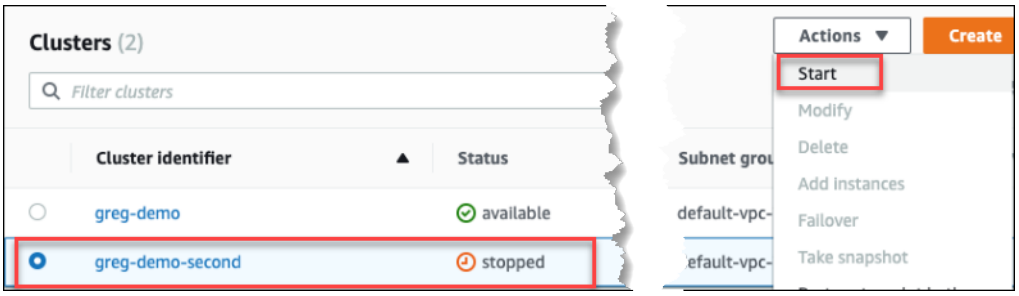


To avoid activating the failover mechanism, the stop operation stops the replica instances first, and then the primary instance.

b. On the confirmation dialog, confirm that you want to stop the cluster by choosing **Stop cluster**, or to keep the cluster running, choose **Cancel**.



- If you want to start the cluster, and the cluster is stopped, choose **Start**.



5. Monitor the status of the cluster and its instances. If you started the cluster, you can resume using the cluster when the cluster and its instances are *available*. For more information, see Determining a cluster's status.



Using the AWS CLI

The following code examples show you how to stop a cluster with one or more instances in the available state, or start a stopped cluster.

To stop a cluster with one or more available instances using the AWS CLI, use the `stop-db-cluster` operation. To start a stopped cluster, use the `start-db-cluster` operation. Both operations use the `--db-cluster-identifier` parameter.

Parameter:

- **`--db-cluster-identifier`**—Required. The name of the cluster to stop or start.

Example— To stop a cluster using the AWS CLI

The following code stops the cluster `sample-cluster`. The cluster must have one or more instances in the available state.

For Linux, macOS, or Unix:

```
aws docdb stop-db-cluster \
   --db-cluster-identifier sample-cluster
```

For Windows:

```
aws docdb stop-db-cluster ^
   --db-cluster-identifier sample-cluster
```

Example— To start a cluster using the AWS CLI

The following code starts the cluster `sample-cluster`. The cluster must currently be stopped.

For Linux, macOS, or Unix:

```
aws docdb start-db-cluster \
   --db-cluster-identifier sample-cluster
```

For Windows:

```
aws docdb start-db-cluster ^
   --db-cluster-identifier sample-cluster
```

Operations you can perform on a stopped cluster

While an Amazon DocumentDB cluster is stopped, you can do a point-in-time restore to any point within your specified automated backup retention window. For details about doing a point-in-time restore, see Restoring to a point in time.

You can't modify the configuration of an Amazon DocumentDB cluster, or any of its instances, while the cluster is stopped. You also can't add or remove instances from the cluster, or delete the cluster if it still has any associated instances. You must start the cluster before performing any such administrative actions.

Amazon DocumentDB applies any scheduled maintenance to your stopped cluster only after it's started again. After seven days, Amazon DocumentDB automatically starts a stopped cluster so that it doesn't fall too far behind in its maintenance status. When the cluster restarts, you will begin to be charged for the instances in the cluster again.

While a cluster is stopped, Amazon DocumentDB does not perform any automated backups nor does it extend the backup retention period.

Deleting an Amazon DocumentDB cluster

You can delete an Amazon DocumentDB cluster using the AWS Management Console or the AWS CLI. To delete a cluster, the cluster must be in the *available* state and must not have any instances associated with it. If the cluster is stopped, first start the cluster, wait for the cluster to become *available*, and then delete the cluster. For more information, see Stopping and starting an Amazon DocumentDB cluster.

Deletion protection

To protect your cluster from accidental deletion, you can enable *deletion protection*. Deletion protection is enabled by default when you create a cluster using the console. However, deletion protection is disabled by default if you create a cluster using the AWS CLI.

Amazon DocumentDB enforces deletion protection for a cluster whether you perform the delete operation using the console or the AWS CLI. If deletion protection is enabled, you can't delete a cluster. To delete a cluster that has deletion protection enabled, you must first modify the cluster and disable deletion protection.

When using the console with deletion protection enabled on a cluster, you can't delete the cluster's last instance because doing so also deletes the cluster. You can delete the last instance of a deletion protected cluster using the AWS CLI. However, the cluster itself still exists, and your data is preserved. You can access the data by creating new instances for the cluster. For more information about enabling and disabling deletion protection, see:

- Creating an Amazon DocumentDB cluster
- Modifying an Amazon DocumentDB cluster

Using the AWS Management Console

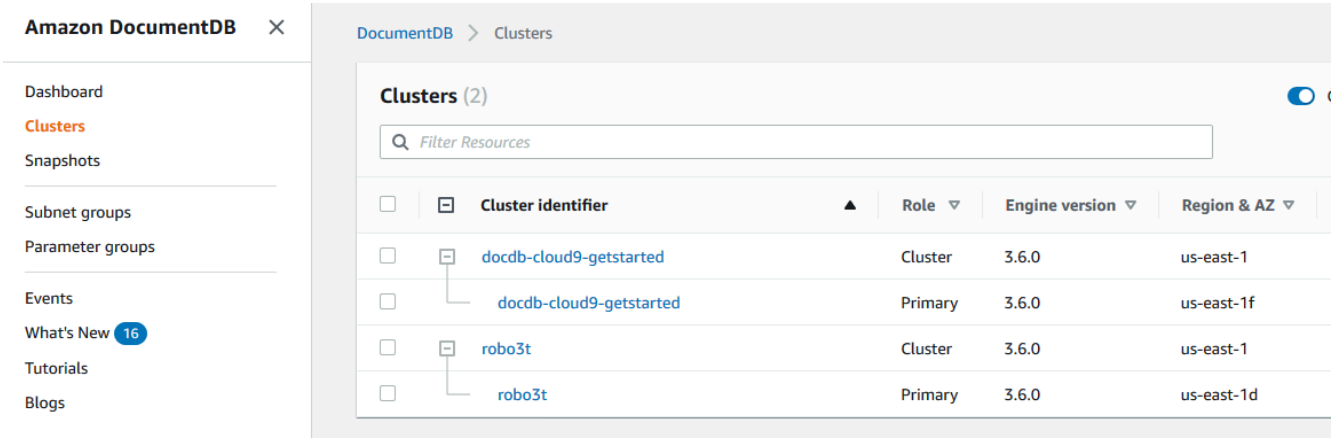
To delete a cluster using the AWS Management Console, deletion protection must be disabled.

To determine whether a cluster has deletion protection enabled:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. Note that in the Clusters navigation box, the **Cluster identifier** column shows both clusters and instances. Instances are listed underneath clusters, similar to the following screenshot.



4. Choose the cluster's name, and select the **Configuration** tab. In the **Cluster details** section, locate **Deletion protection**. If deletion protection is enabled, modify the cluster to disable deletion protection. For information about modifying a cluster, see Modifying an Amazon DocumentDB cluster.

   After **Deletion protection** is disabled, you are ready to delete the cluster.

To delete a cluster:

1. In the navigation pane, choose **Clusters**.

2. Determine whether the cluster has any instances by checking the **Cluster identifier** column for instances listed below it. Before you can delete a cluster, you must delete all of its instances. For more information, see Deleting an Amazon DocumentDB instance.
3. Depending on whether your cluster has any instances, do one of the following steps.
   - If the cluster has no instances, select the button to the left of the cluster name and choose **Actions**. From the dropdown menu, choose **Delete**. Complete the **Delete <cluster-name>** dialog box, and then choose **Delete**.
   - If the cluster has one or more instances, do the following:
     a. In the navigation pane, choose **Clusters**.
     b. Delete each of the cluster's instances by selecting the checkbox to the left of the cluster's name. Select **Actions**, and then choose **Delete**. Complete the **Delete <cluster-name>** dialog box, and then choose **Delete**.

        When you delete the last instance, the cluster will also be deleted. For more information about deleting instances, see Deleting an Amazon DocumentDB instance.

It takes several minutes for the cluster to be deleted. To monitor the status of the cluster, see Monitoring an Amazon DocumentDB cluster's status.

Using the AWS CLI

You cannot delete a cluster that has any instances associated with it. To determine which instances are associated with your cluster, run the `describe-db-clusters` command and delete all of the cluster's instances. Then, if needed, disable deletion protection on your cluster, and finally, delete the cluster.

1. **First, delete all of the cluster's instances.**

   To determine which instances you need to delete, run the following command.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].
[DBClusterIdentifier,DBClusterMembers[*].DBInstanceIdentifier]'
```

   Output from this operation looks something like the following (JSON format).

```
[
    [
        "sample-cluster",
        [
            "sample-instance-1",
            "sample-instance-2"
        ]
    ]
]
```

If the cluster you want to delete has any instances, delete them as shown below.

```
aws docdb delete-db-instance \
    --db-instance-identifier sample-instance
```

2. **Second, disable deletion protection.**

Using the AWS CLI to delete all of a cluster's instances does not delete the cluster. You must also delete the cluster, but you can do this only if deletion protection is disabled.

To determine whether the cluster has deletion protection enabled, run the following command.

> **Tip**
>
> To see the deletion protection status of all your Amazon DocumentDB clusters, omit the `--db-cluster-identifier` parameter.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].[DBClusterIdentifier,DeletionProtection]'
```

Output from this operation looks something like the following.

```
[
    [
        "sample-cluster",
```

```
        "true"
    ]
]
```

If the cluster has deletion protection enabled, modify the cluster and disable deletion protection. To disable deletion protection on the cluster, run the following command.

```
aws docdb modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --no-deletion-protection \
    --apply-immediately
```

3. **Finally, delete the cluster.**

   After deletion protection is disabled, you are ready to delete the cluster. To delete a cluster, use the `delete-db-cluster` operation with the following parameters.

   - **`--db-cluster-identifier`**—Required. The identifier of the cluster that you want to delete.
   - **`--final-db-snapshot-identifier`**—Optional. If you want a final snapshot, you must include this parameter with a name for the final snapshot. You must include either `--final-db-snapshot-identifier` or `--skip-final-snapshot`.

     Naming constraints:

     - Length is [1—63] letters, numbers, or hyphens.
     - First character must be a letter.
     - Cannot end with a hyphen or contain two consecutive hyphens.
     - Must be unique for all clusters across Amazon RDS, Amazon Neptune, and Amazon DocumentDB per AWS account, per Region.
   - **`--skip-final-snapshot`**—Optional. Use this parameter only if you don't want to take a final snapshot before deleting your cluster. The default setting is to take a final snapshot. You must include either `--final-db-snapshot-identifier` or `--skip-final-snapshot`.

The following AWS CLI code deletes the cluster `sample-cluster` with a final snapshot. The operation fails if there are any instances associated with the cluster or if deletion protection is enabled.

Example

For Linux, macOS, or Unix:

```
aws docdb delete-db-cluster \
    --db-cluster-identifier sample-cluster \
    --final-db-snapshot-identifier sample-cluster-final-snapshot
```

For Windows:

```
aws docdb delete-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --final-db-snapshot-identifier sample-cluster-final-snapshot
```

Example

The following AWS CLI code deletes the cluster `sample-cluster` without taking a final snapshot.

For Linux, macOS, or Unix:

```
aws docdb delete-db-cluster \
    --db-cluster-identifier sample-cluster \
    --skip-final-snapshot
```

For Windows:

```
aws docdb delete-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --skip-final-snapshot
```

The output of the `delete-db-cluster` operation is the cluster you are deleting.

It takes several minutes for the cluster to be deleted. To monitor the status of the cluster, see Monitoring a cluster's status.

Scaling Amazon DocumentDB clusters

Amazon DocumentDB enables you to scale the storage and compute in your clusters based on your needs. This section describes how you can use storage scaling, instance scaling, and read scaling to manage performance and scaling for your Amazon DocumentDB clusters and instances.

Topics

- Storage scaling
- Instance scaling
- Read scaling
- Write scaling

Storage scaling

Amazon DocumentDB storage automatically scales with the data in your cluster volume. As your data grows, your cluster volume storage grows in 10 GiB increments, up to 256 TiB for Engine Version 8.0 and beyond (128 TiB for earlier engine versions).

Instance scaling

You can scale your Amazon DocumentDB cluster as needed by modifying the instance class for each instance in the cluster. Amazon DocumentDB supports several instance classes that are optimized for Amazon DocumentDB.

For more information, see Modifying an Amazon DocumentDB instance.

Read scaling

You can achieve read scaling for your Amazon DocumentDB cluster by creating up to 15 Amazon DocumentDB replicas in the cluster. Each Amazon DocumentDB replica returns the same data from the cluster volume with minimal replica lag—usually less than 100 milliseconds after the primary instance has written an update. As your read traffic increases, you can create additional Amazon DocumentDB replicas and connect to them directly to distribute the read load for your cluster. Amazon DocumentDB replicas don't have to be of the same instance class as the primary instance.

For more information, see Adding an Amazon DocumentDB instance to a cluster.

To read scale with Amazon DocumentDB, connect to your cluster as a replica set and distribute reads to replica instances using the built-in read preference capabilities of your driver. For more information, see Connecting to Amazon DocumentDB as a replica set.

Write scaling

You can scale write capacity on your Amazon DocumentDB cluster by increasing the size of your cluster's primary instance. This section provides two methods for scaling your cluster's primary instance based on your needs. The first option seeks to minimize application impact but requires more steps to complete. The second option optimizes for simplicity as it has fewer steps, but it comes with the tradeoff of having more potential impact to your application.

Depending on your application, you can choose what approach below is best for you. For more information about available instance sizes and costs, see the Amazon DocumentDB Pricing page.

1. **Optimize for high availability and performance** — If you are connecting to your cluster in replica set mode (recommended), you can use the following process to minimize the impact to your application when scaling your primary instance. This method minimizes impact because it keeps your cluster at or above your high availability, and read scaling targets are added to the cluster as instances, instead of being updated in place.

    a. Add one or more replicas of the larger instance type to your cluster (see ???). Ensure all replicas are of the same or larger instance type as the primary. This avoids an unintentional reduction in write performance from failing over to a smaller instance type. In most cases, this means temporarily doubling the number of instances in your cluster, then removing the smaller replicas after scaling is complete.

    b. Set the failover tier on all new replicas to priority zero, ensuring a replica of the smaller instance type has the highest failover priority. For more information, see ???.

    c. Initiate a manual failover, which will promote one of the new replicas to be the primary instance. For more information, see ???.

    > **Note**
    >
    > This will incur ~30 seconds of downtime for your cluster. Plan accordingly.

    d. Remove all replicas of an instance type smaller than your new primary from the cluster.

e. Set the failover tier of all instances back to the same priority (usually, this means setting them back to 1).

As an example, suppose that you have a cluster that currently contains three `r5.large` instances (one primary and two replicas), and you want to scale to an `r5.xlarge` instance type. To do so, you would first add three `r5.xlarge` replica instances to your cluster and then set the failover tier of the new `r5.xlarge` replicas to zero. Next, you would initiate a manual failover (understanding that your application will experience ~30 seconds of downtime). Once the failover is complete, you would remove all three `r5.large` instances from your cluster, leaving the cluster scaled to `r5.xlarge` instances.

To help optimize costs, Amazon DocumentDB instances are billed in one second increments, with a ten minute minimum charge following a billable status change such as creating, modifying, or deleting an instance. For more information, see Cost optimization in the best practices documentation.

2. **Optimize for simplicity** — This approach optimizes for simplicity. It doesn't expand and contract the cluster, but it might temporarily reduce your read capacity.

   It is possible that changing the instance class of a replica will result in that instance not serving requests for a brief period of time, from a few seconds to less than 30 seconds. If you are connecting to your cluster in replica set mode (recommended), then this would reduce your read capacity by one replica (e.g., to 66% capacity in a 3-node cluster, or 75% capacity in a 4-node cluster, etc.) during the scaling operation.

   a. Scale one of the replica instances in your cluster. For more information, see Managing instance classes.

   b. Wait until the instance is available (see Monitoring an Amazon DocumentDB instance's status).

      > **Note**
      >
      > This will incur ~30 seconds of downtime for your cluster. Plan accordingly.

   c. Continue executing steps 1 and 2 until all replicas instances have been scaled, one by one.

   d. Initiate a manual failover. This will promote one of the replicas to be the primary instance. For more information, see Amazon DocumentDB failover.

> **Note**
>
> This will incur up to 30 seconds of downtime for your cluster, but often takes less time than that. Plan accordingly.

e. Scale the former primary (now a replica) instance.

Cloning a volume for an Amazon DocumentDB cluster

By using Amazon DocumentDB cloning, you can create a new cluster that uses the same Amazon DocumentDB cluster volume and has the same data as the original. The process is designed to be fast and cost-effective. The new cluster with its associated data volume is known as a *clone*. Creating a clone is faster and more space-efficient than physically copying the data using other techniques, such as restoring a snapshot.

Amazon DocumentDB supports creating an Amazon DocumentDB provisioned clone from a provisioned Amazon DocumentDB cluster. When you create a clone using a different deployment configuration than the source, the clone is created using the latest version of the source's Amazon DocumentDB engine.

When you create clones from your Amazon DocumentDB clusters, the clones are created in your AWS account—the same account that owns the source Amazon DocumentDB cluster.

Topics

- Overview of Amazon DocumentDB cloning
- Usage notes for Amazon DocumentDB cloning
- How Amazon DocumentDB cloning works
- Creating an Amazon DocumentDB clone

Overview of Amazon DocumentDB cloning

Amazon DocumentDB uses a *copy-on-write protocol* to create a clone. This mechanism uses minimal additional space to create an initial clone. When the clone is first created, Amazon DocumentDB keeps a single copy of the data that is used by the source DB cluster and the new (cloned) Amazon DocumentDB cluster. Additional storage is allocated only when changes are made to data (on the Amazon DocumentDB storage volume) by the source Amazon DocumentDB cluster

or the Amazon DocumentDB cluster clone. To learn more about the copy-on-write protocol, see How Amazon DocumentDB cloning works.

Amazon DocumentDB cloning is especially useful for quickly setting up test environments using your production data, without risking data corruption. You can use clones for many types of applications, such as the following:

- Experiment with potential changes (schema changes and parameter group changes, for example) to assess all impacts.
- Run workload-intensive operations, such as exporting data or running analytical queries on the clone.
- Create a copy of your production DB cluster for development, testing, or other purposes.

You can create more than one clone from the same Amazon DocumentDB cluster. You can also create multiple clones from another clone.

After creating an Amazon DocumentDB clone, you can configure the Amazon DocumentDB instances differently from the source Amazon DocumentDB cluster. For example, you might not need a clone for development purposes to meet the same high availability requirements as the source production Amazon DocumentDB cluster. In this case, you can configure the clone with a single Amazon DocumentDB instance rather than the multiple DB instances used by the Amazon DocumentDB cluster.

When you finish using the clone for your testing, development, or other purposes, you can delete it.

Usage notes for Amazon DocumentDB cloning

Amazon DocumentDB; cloning currently has the following limitations:

- You can create as many clones as you want, up to the maximum number of DB clusters allowed in the AWS Region. However, after you create 15 clones, the next clone is a full copy. The cloning operation acts like a point-in-time recovery.
- You can't create a clone in a different AWS Region from the source Amazon DocumentDB cluster.
- You can't create a clone from an Amazon DocumentDB cluster that has no DB instances. You can only clone Amazon DocumentDB clusters that have at least one DB instance.

- You can create a clone in a different virtual private cloud (VPC) than that of the Amazon DocumentDB cluster. If you do, the subnets of the VPCs must map to the same Availability Zones.

How Amazon DocumentDB cloning works

Amazon DocumentDB cloning works at the storage layer of an Amazon DocumentDB cluster. It uses a *copy-on-write* protocol that's both fast and space-efficient in terms of the underlying durable media supporting the Amazon DocumentDB storage volume. You can learn more about Amazon DocumentDB cluster volumes in Managing Amazon DocumentDB clusters.

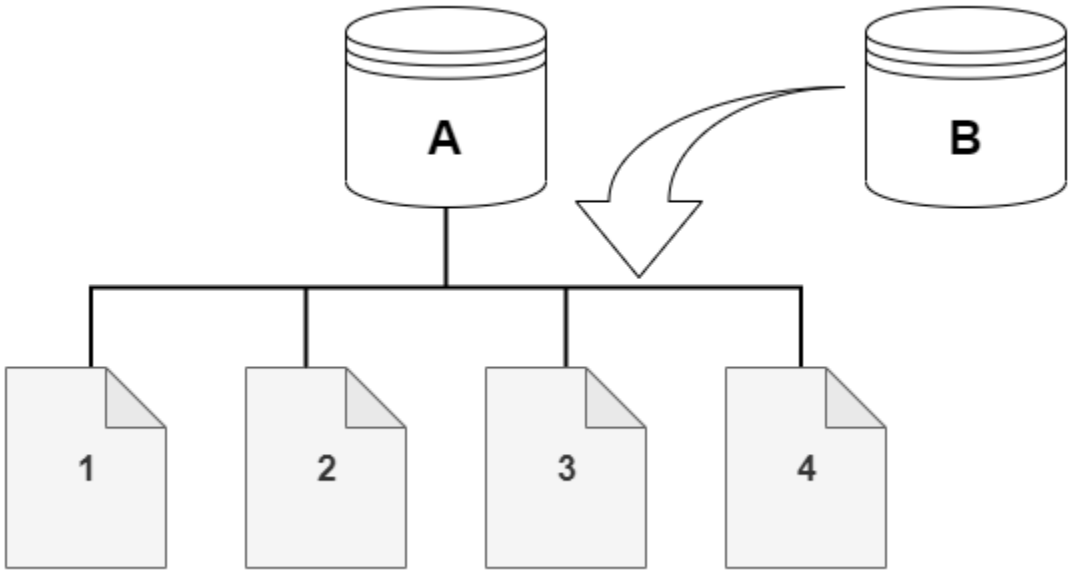
Topics

- Understanding the copy-on-write protocol
- Deleting a source cluster volume

Understanding the copy-on-write protocol

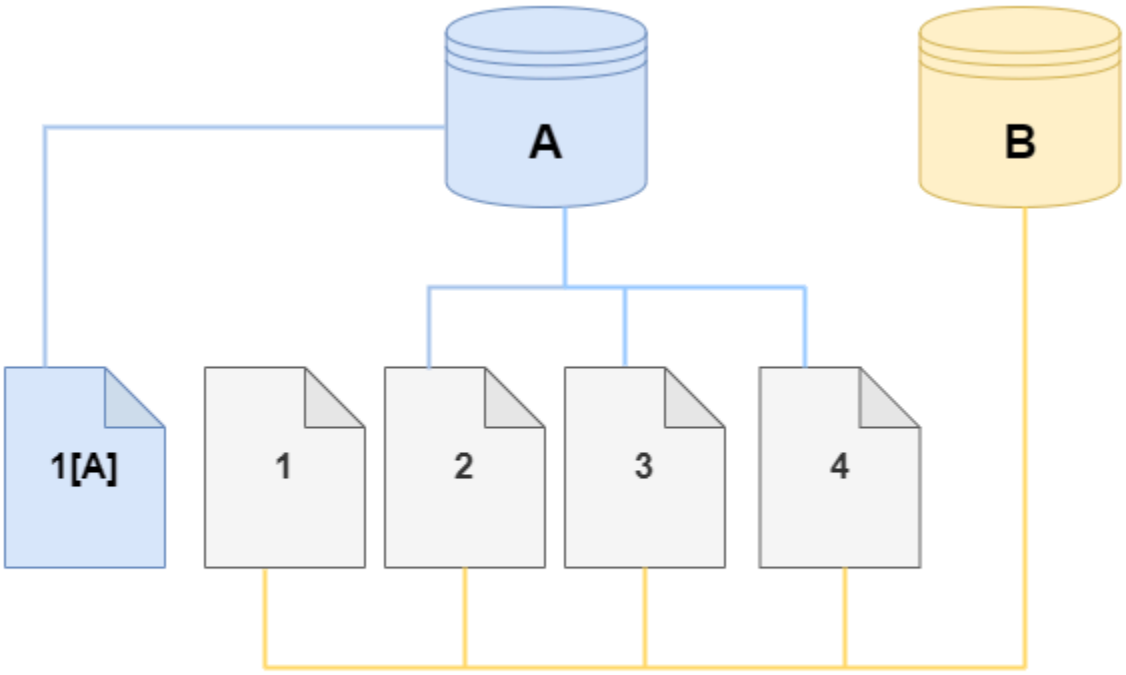
An Amazon DocumentDB cluster stores data in pages in the underlying Amazon DocumentDB storage volume.

For example, in the following diagram you can find an Amazon DocumentDB cluster (A) that has four data pages, 1, 2, 3, and 4. Imagine that a clone, B, is created from the Amazon DocumentDB cluster. When the clone is created, no data is copied. Rather, the clone points to the same set of pages as the source Amazon DocumentDB cluster.

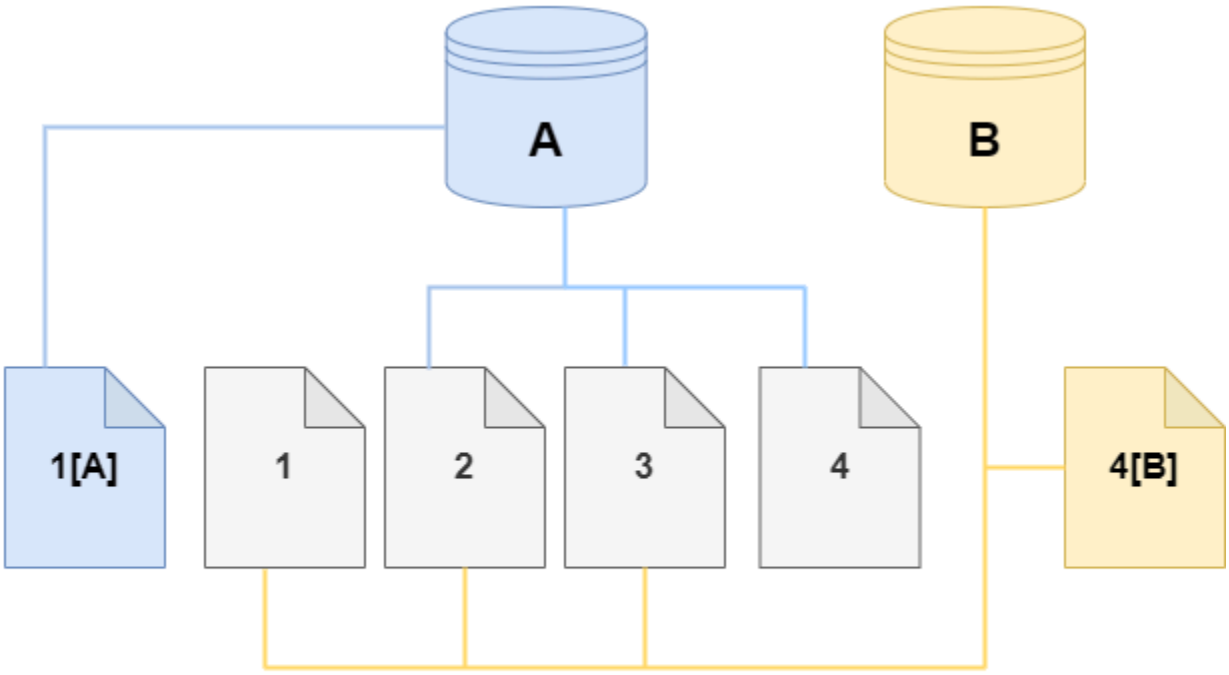


When the clone is created, no additional storage is usually needed. The copy-on-write protocol uses the same segment on the physical storage media as the source segment. Additional storage is required only if the capacity of the source segment isn't sufficient for the entire clone segment. If that's the case, the source segment is copied to another physical device.

In the following diagrams, you can find an example of the copy-on-write protocol in action using the same cluster A and its clone, B, as shown preceding. Let's say that you make a change to your Amazon DocumentDB cluster (A) that results in a change to data held on page 1. Instead of writing to the original page 1, Amazon DocumentDB creates a new page 1[A]. The Amazon DocumentDB cluster volume for cluster (A) now points to page 1[A], 2, 3, and 4, while the clone (B) still references the original pages.



On the clone, a change is made to page 4 on the storage volume. Instead of writing to the original page 4, Amazon DocumentDB creates a new page, 4[B]. The clone now points to pages 1, 2, 3, and to page 4[B], while the cluster (A) continues pointing to 1[A], 2, 3, and 4.



As more changes occur over time in both the source Amazon DocumentDB cluster volume and the clone, more storage is needed to capture and store the changes.

Deleting a source cluster volume

When you delete a source cluster volume that has one or more clones associated with it, the clones aren't affected. The clones continue to point to the pages that were previously owned by the source cluster volume.

Creating an Amazon DocumentDB clone

You can create a clone in the same AWS account as the source Amazon DocumentDB cluster. To do so, you can use the AWS Management Console or the AWS CLI and the procedures following.

By using Amazon DocumentDB cloning, you can create a provisioned Amazon DocumentDB cluster clone from a provisioned Amazon DocumentDB cluster.

Using the AWS Management Console

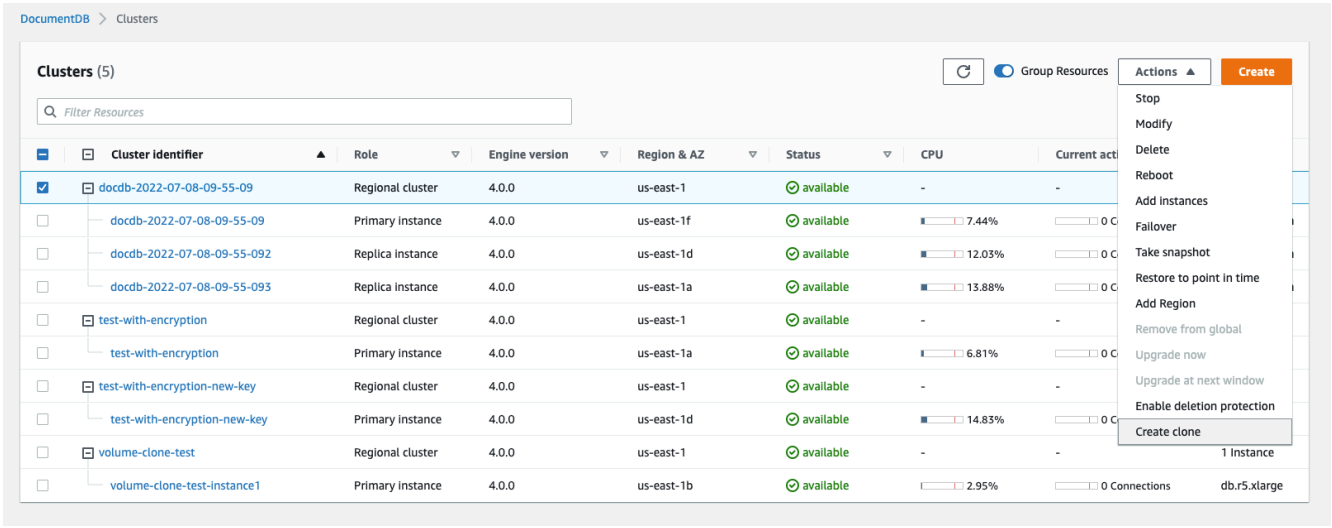
The following procedure describes how to clone an Amazon DocumentDB cluster using the AWS Management Console.

Creating a clone using the AWS Management Console results in an Amazon DocumentDB cluster with one Amazon DocumentDB instance.

These instructions apply for DB clusters owned by the same AWS account that is creating the clone. The DB cluster must be owned by the same AWS account as cross-account cloning is not supported in Amazon DocumentDB.

To create a clone of a DB cluster owned by your AWS account using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. Choose your Amazon DocumentDB cluster from the list, and for **Actions**, choose **Create clone**.

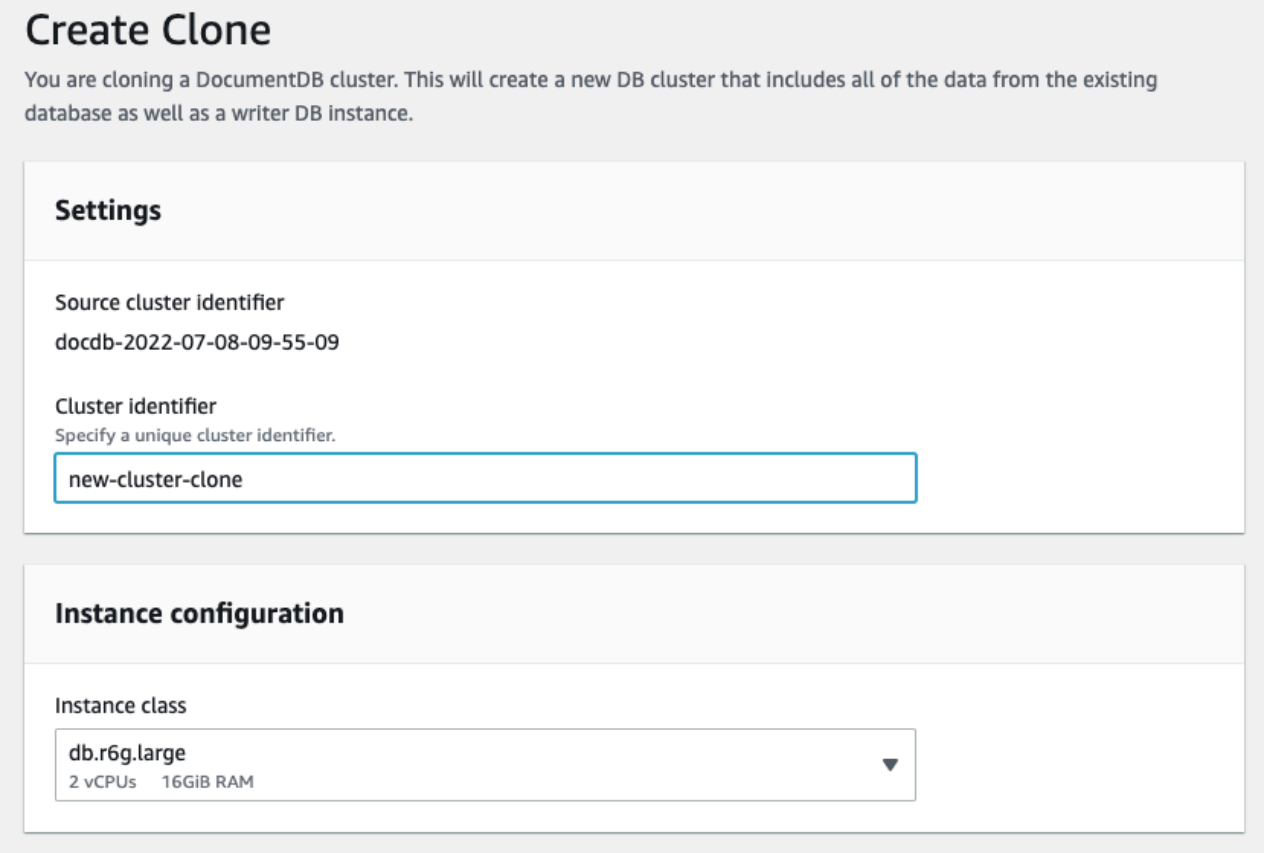


The Create clone page opens, where you can configure a **Cluster identifier** and an **Instance class**, and other options for the Amazon DocumentDB cluster clone.

4. In the **Settings** section, do the following:

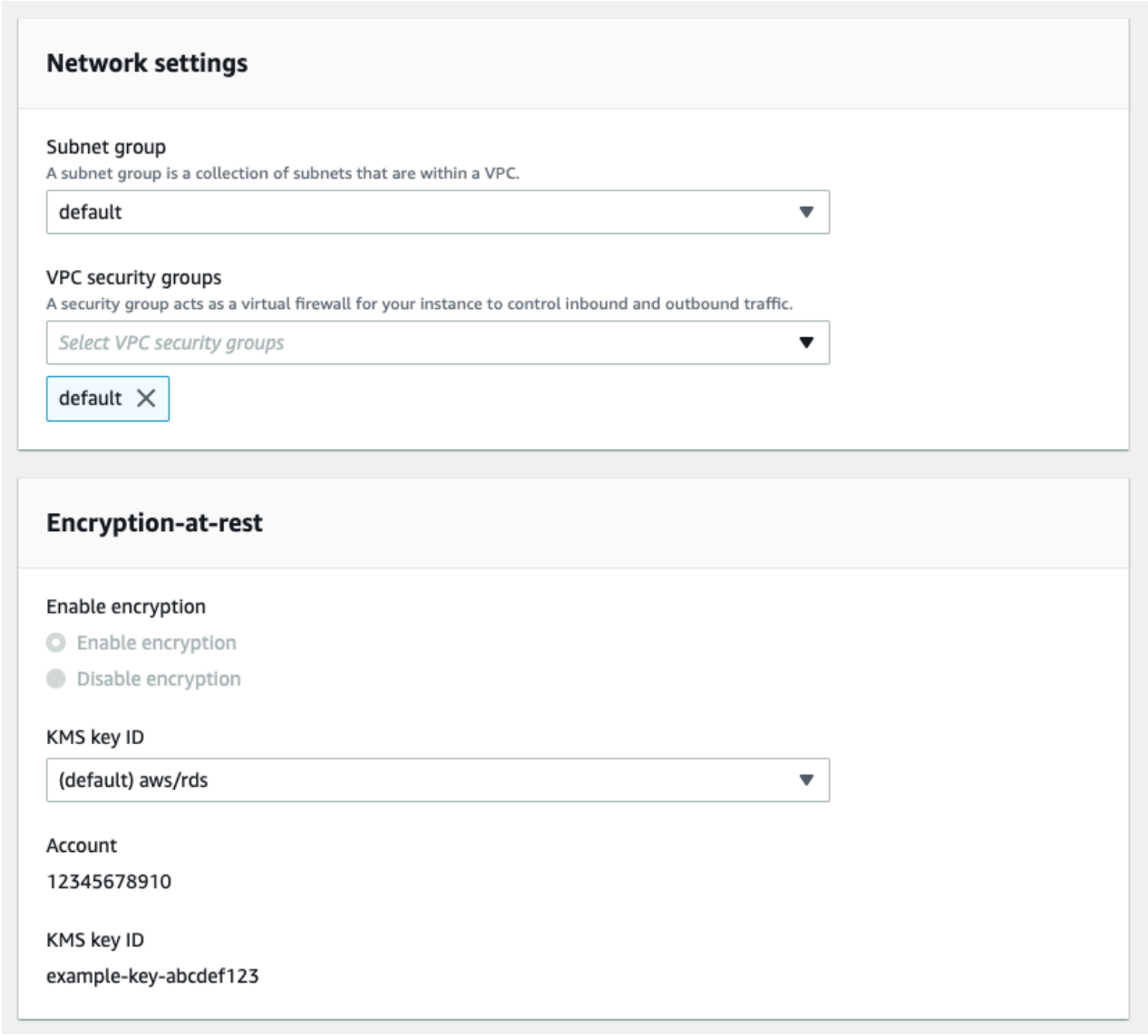
   a. For **Cluster identifier**, enter the name that you want to give to your cloned Amazon DocumentDB cluster.

b. For **Instance configuration**, select an appropriate **Instance class** for your cloned Amazon DocumentDB cluster.

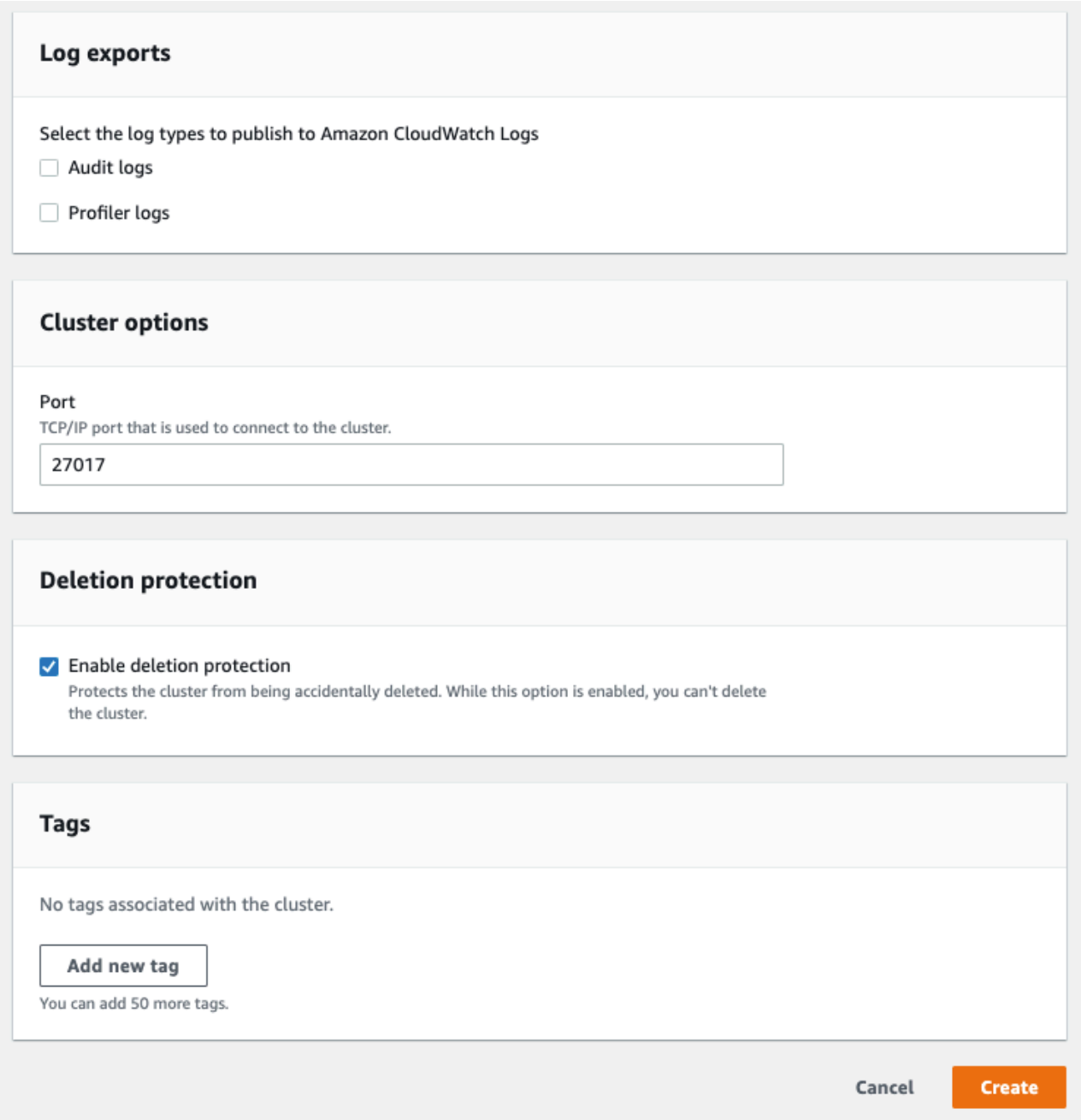


c. For **Network settings**, choose a **Subnet group** for your use case and the associated VPC security groups.

d. For **Encryption-at-rest**, if the source cluster (the cluster that is being cloned) has encryption enabled, the cloned cluster must also have encryption enabled. If this scenario is true, then the **Enable encryption** options are grayed out (disabled) but with the **Enable encryption** choice selected. Conversely, if the source cluster does not have encryption enabled, the **Enable encryption** options are available and you can choose to enable or disable encryption.



e. Complete the new cluster clone configuration by selecting the type of logs to export (optional), entering a specific port used to connect to the cluster, and enabling protection from accidentally deleting the cluster (enabled by default).



f. Finish entering all settings for your Amazon DocumentDB cluster clone. To learn more about Amazon DocumentDB cluster and instance settings, see Managing Amazon DocumentDB clusters.

5. Choose **Create clone** to launch the Amazon DocumentDB clone of your chosen Amazon DocumentDB cluster.

When the clone is created, it's listed with your other Amazon DocumentDB clusters in the console **Databases** section and displays its current state. Your clone is ready to use when its state is **Available**.

Using the AWS CLI

Using the AWS CLI for cloning your Amazon DocumentDB cluster involves a couple of steps.

The `restore-db-cluster-to-point-in-time` AWS CLI command that you use results in an empty Amazon DocumentDB cluster with 0 Amazon DocumentDB instances. That is, the command restores only the Amazon DocumentDB cluster, not the DB instances for that cluster. You do that separately after the clone is available. The two steps in the process are as follows:

1. Create the clone by using the restore-db-cluster-to-point-in-time CLI command. The parameters that you use with this command control the capacity type and other details of the empty Amazon DocumentDB cluster (clone) being created.
2. Create the Amazon DocumentDB instance for the clone by using the create-db-instance CLI command to recreate the Amazon DocumentDB instance in the restored Amazon DocumentDB cluster.

The commands following assume that the AWS CLI is set up with your AWS Region as the default. This approach saves you from passing the `--region` name in each of the commands. For more information, see Configuring the AWS CLI. You can also specify the `--region` in each of the CLI commands that follow.

Creating the clone

The specific parameters that you pass to the `restore-db-cluster-to-point-in-time` CLI command vary. What you pass depends on the type of clone that you want to create.

Use the following procedure to create a provisioned Amazon DocumentDB clone from a provisioned Amazon DocumentDB cluster.

To create a clone of the same engine mode as the source Amazon DocumentDB cluster

- Use the `restore-db-cluster-to-point-in-time` CLI command and specify values for the following parameters:
  - `--db-cluster-identifier` – Choose a meaningful name for your clone. You name the clone when you use the restore-db-cluster-to-point-in-time CLI command.

- `--restore-type` – Use `copy-on-write` to create a clone of the source DB cluster. Without this parameter, the `restore-db-cluster-to-point-in-time` restores the Amazon DocumentDB cluster rather than creating a clone. Default for `restore-type` is `full-copy`.
- `--source-db-cluster-identifier` – Use the name of the source Amazon DocumentDB cluster that you want to clone.
- `--use-latest-restorable-time` – This value points to the latest restorable volume data for the clone. This parameter is required for `restore-type copy-on-write`, however, you can not use the `restore-to-time parameter` with it.

The following example creates a clone named `my-clone` from a cluster named `my-source-cluster`.

For Linux, macOS, or Unix:

```
aws docdb restore-db-cluster-to-point-in-time \
    --source-db-cluster-identifier my-source-cluster \
    --db-cluster-identifier my-clone \
    --restore-type copy-on-write \
    --use-latest-restorable-time
```

For Windows:

```
aws docdb restore-db-cluster-to-point-in-time ^
    --source-db-cluster-identifier my-source-cluster ^
    --db-cluster-identifier my-clone ^
    --restore-type copy-on-write ^
    --use-latest-restorable-time
```

The command returns the JSON object containing details of the clone. Check to make sure that your cloned DB cluster is available before trying to create the DB instance for your clone. For more information, see Checking the status and getting clone details below:

Checking the status and getting clone details

You can use the following command to check the status of your newly created empty DB cluster.

```
$ aws docdb describe-db-clusters --db-cluster-identifier my-clone --query '*[].
[Status]' --output text
```

Or you can obtain the status and the other values that you need to create the DB instance for your clone by using the following AWS CLI query:

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters --db-cluster-identifier my-clone \
  --query '*[].{Status:Status,Engine:Engine,EngineVersion:EngineVersion}'
```

For Windows:

```
aws docdb describe-db-clusters --db-cluster-identifier my-clone ^
  --query "*[].{Status:Status,Engine:Engine,EngineVersion:EngineVersion}"
```

This query returns output similar to the following.

```
[
  {
        "Status": "available",
        "Engine": "docdb",
        "EngineVersion": "4.0.0",
    }
]
```

Creating the Amazon DocumentDB instance for your clone

Use the create-db-instance CLI command to create the DB instance for your clone.

The `--db-instance-class` parameter is used for provisioned Amazon DocumentDB clusters only.

For Linux, macOS, or Unix:

```
aws docdb create-db-instance \
    --db-instance-identifier my-new-db \
    --db-cluster-identifier my-clone \
    --db-instance-class  db.r5.4xlarge \
```

```
    --engine docdb
```

For Windows:

```
aws docdb create-db-instance ^
    --db-instance-identifier my-new-db ^
    --db-cluster-identifier my-clone ^
    --db-instance-class  db.r5.4xlarge ^
    --engine docdb
```

Parameters to use for cloning

The following table summarizes the various parameters used with `restore-db-cluster-to-point-in-time` to clone Amazon DocumentDB clusters.

Parameter	Description
--source-db-cluster-identifier	Use the name of the source Amazon DocumentDB cluster that you want to clone.
--db-cluster-identifier	Choose a meaningful name for your clone. You name your clone with the `restore-db-cluster-to-point-in-time` command. Then you pass this name to the `create-db-instance` command.
--restore-type	Specify `copy-on-write` as the `--restore-type` to create a clone of the source DB cluster rather than restoring the source Amazon DocumentDB cluster.
--use-latest-restorable-time	This value points to the latest restorable volume data for the clone.

Understanding Amazon DocumentDB cluster fault tolerance

Amazon DocumentDB clusters are fault tolerant by design. Each cluster's volume spans multiple Availability Zones in a single AWS Region, and each Availability Zone contains a copy of the cluster's volume data. This functionality means that your cluster can tolerate an Availability Zone failure without any loss of data and only a brief interruption of service.

If the primary instance in a cluster fails, Amazon DocumentDB automatically performs a failover to a new primary instance in one of two ways:

- By promoting an existing Amazon DocumentDB replica to the new primary instance chosen based on the Promotion Tier setting of each replica, and then creating a replacement for the former primary. A failover to the replica instance typically takes less than 30 seconds. Read and write operations may experience brief interruption during this period. To increase the availability of your cluster, create at least one or more Amazon DocumentDB replicas in two or more different Availability Zones.
- By creating a new primary instance. This only happens if you do not have a replica instance in your cluster and can take a few minutes to complete.

If the cluster has one or more Amazon DocumentDB replicas, an Amazon DocumentDB replica is promoted to the primary instance during a failure event. A failure event results in a brief interruption, during which read and write operations fail with an exception. However, service is typically restored in less than 120 seconds, and often less than 60 seconds. To increase the availability of your cluster, create at least one or more Amazon DocumentDB replicas in two or more different Availability Zones.

You can customize the order in which your Amazon DocumentDB replicas are promoted to the primary instance after a failure by assigning each replica a priority. Priorities range from 0 for the highest priority to 15 for the lowest priority. If the primary instance fails, the Amazon DocumentDB replica with the highest priority is promoted to the new primary instance. You can modify the priority of an Amazon DocumentDB replica at any time. Modifying the priority doesn't trigger a failover. You can use the `modify-db-instance` operation with the `--promotion-tier` parameter. For more information about customizing the failover priority of an instance, see Amazon DocumentDB failover.

More than one Amazon DocumentDB replica can share the same priority, resulting in promotion tiers. If two or more Amazon DocumentDB replicas share the same priority, then the replica that is largest in size is promoted to primary. If two or more Amazon DocumentDB replicas share the same priority and size, an arbitrary replica in the same promotion tier is promoted.

If the cluster doesn't contain any Amazon DocumentDB replicas, the primary instance is re-created during a failure event. A failure event results in an interruption, during which read and write operations fail with an exception. Service is restored when the new primary instance is created, which typically takes less than 10 minutes. Promoting an Amazon DocumentDB replica to the primary instance is much faster than creating a new primary instance.

Managing Amazon DocumentDB instances

The following topics provide information to help you manage your Amazon DocumentDB instances. They include details about instance classes and statuses, and how to create, delete, and modify an instance.

Topics

- Determining an instance's status
- Amazon DocumentDB instance lifecycle
- Managing instance classes
- NVMe-backed instances

Determining an instance's status

To see the valid instance statuses, their meaning, and how to determine the status of your instances, see Monitoring an Amazon DocumentDB instance's status.

Amazon DocumentDB instance lifecycle

The lifecycle of an Amazon DocumentDB instance includes creating, modifying, maintaining and upgrading, performing backups and restores, rebooting, and deleting the instance. This section provides information about how to complete these processes.

Topics

- Adding an Amazon DocumentDB instance to a cluster
- Describing Amazon DocumentDB instances
- Modifying an Amazon DocumentDB instance
- Rebooting an Amazon DocumentDB instance
- Deleting an Amazon DocumentDB instance

You can create a new Amazon DocumentDB instance using the AWS Management Console or the AWS CLI. To add an instance to a cluster, the cluster must be in an *available* state. You cannot add an instance to a cluster that is stopped. If the cluster is stopped, first start the cluster, wait for the cluster to become *available*, and then add an instance. For more information, see Stopping and starting an Amazon DocumentDB cluster.

> **Note**
>
> If you create an Amazon DocumentDB cluster using the console, an instance is automatically created for you at the same time. If you want to create additional instances, use one of the following procedures.

Adding an Amazon DocumentDB instance to a cluster

Using the AWS Management Console

Use the following procedure to create an instance for your cluster using the Amazon DocumentDB console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. To choose the cluster that you want to add an instance to, select the button to the left of the cluster's name.
4. Choose **Actions**, and then choose **Add instances**.
5. In the **Add instance to: <cluster-name>** page, repeat the following steps for each instance that you want to add to the cluster. You can have up to 15.

   a. **Instance identifier**— You can either enter a unique identifier for this instance or allow Amazon DocumentDB to provide the instance identifier based on the cluster identifier.

      Instance naming constraints:

      - Length is [1—63] letters, numbers, or hyphens.
      - First character must be a letter.
      - Cannot end with a hyphen or contain two consecutive hyphens.

- Must be unique for all instances across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.

b. **Instance class** — From the drop-down list, choose the instance type you want for this instance.

c. **Promotion tier** — From the drop-down list, choose the promotion tier for your instance or choose *No preference* to allow Amazon DocumentDB to set the promotion tier for your instance. Lower numbers mean higher priority. For more information, see Controlling the failover target.

d. To add more instances, choose **Add additional instances** and repeat steps a, b, and c.

6. Finish the operation.

- To add the instances to your cluster, choose **Create**.
- To cancel the operation, choose **Cancel**.

It takes several minutes to create an instance. You can use the console or AWS CLI to view the instance's status. For more information, see Monitoring an instance's status.

Using the AWS CLI

Use the `create-db-instance` AWS CLI operation with the following parameters to create the primary instance for your cluster.

- **`--db-instance-class`** — Required. The compute and memory capacity of the instance, for example, `db.m4.large`. Not all instance classes are available in all AWS Regions.
- **`--db-instance-identifier`** — Required. A string that identifies the instance.

  Instance Naming Constraints:

  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all instances across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.
- **`--engine`** — Required. Must be `docdb`.
- **`--availability-zone`** — Optional. The Availability Zone that you want this instance to be created in. Use this parameter to locate your instances in different Availability Zones to

increase fault tolerance. For more information, see Amazon DocumentDB high availability and replication.

- **`--promotion-tier`** — Optional. The failover priority tier for this instance. Must be between 0 and 15 with lower numbers being higher priority. For more information, see Controlling the failover target.

1. **First, determine what Availability Zones you can create your instance in.**

   If you want to specify the Availability Zone before you create your instance, run the following command to determine which Availability Zones are available for your Amazon DocumentDB cluster.

   For Linux, macOS, or Unix:

   ```
   aws docdb describe-db-clusters \
         --query 'DBClusters[*].[DBClusterIdentifier,AvailabilityZones[*]]'
   ```

   For Windows:

   ```
   aws docdb describe-db-clusters ^
         --query 'DBClusters[*].[DBClusterIdentifier,AvailabilityZones[*]]'
   ```

   Output from this operation looks something like the following.

   ```
   [
       [
           "sample-cluster",
           [
               "us-east-1c",
               "us-east-1b",
               "us-east-1a"
           ]
       ]
   ]
   ```

2. **Second, determine what instance classes you can create in your Region.**

To determine which instance classes are available to you in your Region, run the following command. From the output, choose an instance class for the instance you want to add to your Amazon DocumentDB cluster.

For Linux, macOS, or Unix:

```
aws docdb describe-orderable-db-instance-options \
        --engine docdb \
        --query 'OrderableDBInstanceOptions[*].DBInstanceClass'
```

For Windows:

```
aws docdb describe-orderable-db-instance-options ^
        --engine docdb ^
        --query 'OrderableDBInstanceOptions[*].DBInstanceClass'
```

Output from this operation looks something like the following.

```
[
    "db.r5.16xlarge",
    "db.r5.2xlarge",
    "db.r5.4xlarge",
    "db.r5.8xlarge",
    "db.r5.large",
    "db.r5.xlarge"
]
```

3. **Finally, add an instance to your Amazon DocumentDB cluster.**

   To add an instance to your Amazon DocumentDB cluster, run the following command..

   For Linux, macOS, or Unix:

```
aws docdb create-db-instance \
       --db-cluster-identifier sample-cluster \
       --db-instance-identifier sample-instance-2 \
       --availability-zone us-east-1b \
       --promotion-tier 2 \
       --db-instance-class db.r5.xlarge \
       --engine docdb
```

For Windows:

```
aws docdb create-db-instance ^
       --db-cluster-identifier sample-cluster ^
       --db-instance-identifier sample-instance-2 ^
       --availability-zone us-east-1b ^
       --promotion-tier 2 ^
       --db-instance-class db.r5.xlarge ^
       --engine docdb
```

Output from this operation looks something like the following.

```
{
    "DBInstance": {
        "DBInstanceIdentifier": "sample-instance-2",
        "DBInstanceClass": "db.r5.xlarge",
        "Engine": "docdb",
        "DBInstanceStatus": "creating",
        "PreferredBackupWindow": "02:00-02:30",
        "BackupRetentionPeriod": 1,
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcd0123",
                "Status": "active"
            }
        ],
        "AvailabilityZone": "us-east-1b",
        "DBSubnetGroup": {
            "DBSubnetGroupName": "default",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-6242c31a",
            "SubnetGroupStatus": "Complete",
            "Subnets": [
                {
                    "SubnetIdentifier": "subnet-abcd0123",
                    "SubnetAvailabilityZone": {
                        "Name": "us-west-2a"
                    },
                    "SubnetStatus": "Active"
                },
                {
                    "SubnetIdentifier": "subnet-wxyz0123",
```

```
                    "SubnetAvailabilityZone": {
                        "Name": "us-west-2b"
                    },
                    "SubnetStatus": "Active"
                }
            ]
        },
        "PreferredMaintenanceWindow": "sun:11:35-sun:12:05",
        "PendingModifiedValues": {},
        "EngineVersion": "3.6.0",
        "AutoMinorVersionUpgrade": true,
        "PubliclyAccessible": false,
        "DBClusterIdentifier": "sample-cluster",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DbiResourceId": "db-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "CACertificateIdentifier": "rds-ca-2019",
        "PromotionTier": 2,
        "DBInstanceArn": "arn:aws:rds:us-east-1:<accountID>:db:sample-instance-2"
    }
}
```

It takes several minutes to create the instance. You can use the console or AWS CLI to view the instance's status. For more information, see Monitoring an Amazon DocumentDB instance's status.

Describing Amazon DocumentDB instances

You can use either the Amazon DocumentDB Management Console or the AWS CLI to see details such as connection endpoints, security groups VPCs, certificate authority, and parameter groups pertaining to your Amazon DocumentDB instances.

Using the AWS Management Console

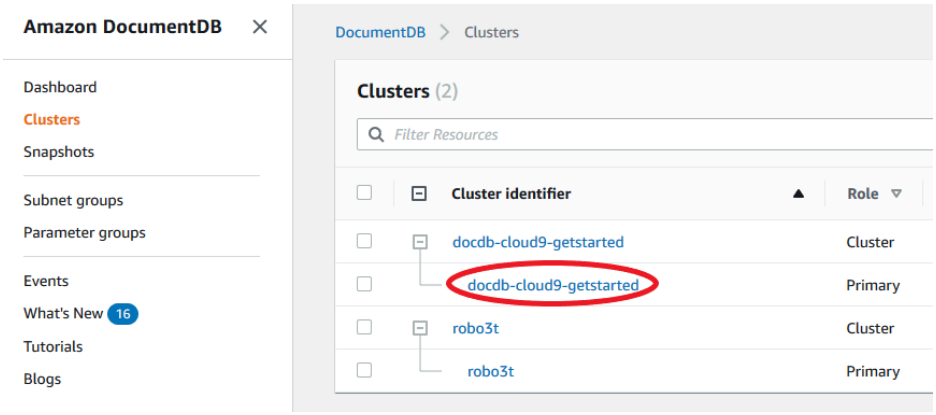
To view the details of your instances using the AWS Management Console, follow the steps below.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

2. In the navigation pane, choose **Clusters** .

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. In the list of instances, choose the name of the instance that you want to see its details. The information about the instance is organized into the following groupings:

   - **Summary**—General information about the instance, including the engine version, class, status, and any pending maintenance.
   - **Connectivity & Security** —The **Connect** section lists the connection endpoints to connect to this instance with the mongo shell or with an application. The **Security Groups** section lists the security groups associated with this instance and their VPC ID and descriptions.
   - **Configuration**—The **Details** section lists the configurations and status of the instance, including the instance's Amazon Resource Name (ARN), endpoint, role, class, and certificate authority. It also lists the instance's security and network settings, and backup information. The **Cluster details** section lists the details of the cluster that this instance belongs to. The **Cluster instances** section lists all the instances that belong to your cluster with each instance's role and cluster parameter group status.

> **Note**
>
> You can modify the cluster associated with your instance by selecting **Modify** next to the **Cluster details** header. For more information, see Modifying an Amazon DocumentDB cluster.

- **Monitoring**—The CloudWatch Logs metrics for this instance. For more information, see Monitoring Amazon DocumentDB with CloudWatch.
- **Events & tags** —The **Recent events** section lists the recent events for this instance. Amazon DocumentDB keeps a record of events that relate to your clusters, instances, snapshots, security groups, and cluster parameter groups. This information includes the date, time, and message associated with each event. The **Tags** section lists the tags attached to this cluster. For more information, see Tagging Amazon DocumentDB resources.

Using the AWS CLI

To view the details of your Amazon DocumentDB instances using the AWS CLI, use the `describe-db-clusters` command as shown in the following examples. For more information, see `DescribeDBInstances` in the *Amazon DocumentDB Resource Management API Reference*.

> **Note**
>
> For certain management features such as cluster and instance lifecycle management, Amazon DocumentDB leverages operational technology that is shared with Amazon RDS. The `filterName=engine,Values=docdb` filter parameter returns only Amazon DocumentDB clusters.

1. **List all Amazon DocumentDB instances.**

   The following AWS CLI code lists the details for all Amazon DocumentDB instances in a Region.

   For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --filter Name=engine,Values=docdb
```

For Windows:

```
aws docdb describe-db-instances \
    --filter Name=engine,Values=docdb
```

2. **List all details for a specified Amazon DocumentDB instance**

The following code lists the details for `sample-cluster-instance`. Including the `--db-instance-identifier` parameter with the name of an instance restricts the output to information on that particular instance.

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --db-instance-identifier sample-cluster-instance
```

For Windows:

```
aws docdb describe-db-instances \
    --db-instance-identifier sample-cluster-instance
```

Output from this operation looks like the following.

```
{
    "DBInstances": [
        {
            "DbiResourceId": "db-BJKKB54PIDV5QFKGVRX5T3S6GM",
            "DBInstanceArn": "arn:aws:rds:us-east-1:012345678901:db:sample-
cluster-instance-00",
            "VpcSecurityGroups": [
                {
                    "VpcSecurityGroupId": "sg-77186e0d",
                    "Status": "active"
                }
            ],
            "DBInstanceClass": "db.r5.large",
            "DBInstanceStatus": "creating",
```

```
            "AutoMinorVersionUpgrade": true,
            "PreferredMaintenanceWindow": "fri:09:32-fri:10:02",
            "BackupRetentionPeriod": 1,
            "StorageEncrypted": true,
            "DBClusterIdentifier": "sample-cluster",
            "EngineVersion": "3.6.0",
            "AvailabilityZone": "us-east-1a",
            "Engine": "docdb",
            "PromotionTier": 2,
            "DBInstanceIdentifier": "sample-cluster-instance",
            "PreferredBackupWindow": "00:00-00:30",
            "PubliclyAccessible": false,
            "DBSubnetGroup": {
                "DBSubnetGroupName": "default",
                "Subnets": [
                    {
                        "SubnetIdentifier": "subnet-4e26d263",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1a"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
                        "SubnetIdentifier": "subnet-afc329f4",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1c"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
                        "SubnetIdentifier": "subnet-b3806e8f",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1e"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
                        "SubnetIdentifier": "subnet-53ab3636",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1d"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
```

```
                        "SubnetIdentifier": "subnet-991cb8d0",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1b"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
                        "SubnetIdentifier": "subnet-29ab1025",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1f"
                        },
                        "SubnetStatus": "Active"
                    }
                ],
                "VpcId": "vpc-91280df6",
                "DBSubnetGroupDescription": "default",
                "SubnetGroupStatus": "Complete"
            },
            "PendingModifiedValues": {},
            "KmsKeyId": "arn:aws:kms:us-east-1:012345678901:key/0961325d-
a50b-44d4-b6a0-a177d5ff730b"
        }
    ]
}
```

Modifying an Amazon DocumentDB instance

You can modify your Amazon DocumentDB instance using either the AWS Management Console or the AWS CLI. To modify an instance, the instance must be in the *available* state. You cannot modify an instance that is stopped. If the cluster is stopped, first start the cluster, wait for the instance to become *available*, and then make the desired modifications. For more information, see Stopping and starting an Amazon DocumentDB cluster.

Using the AWS Management Console

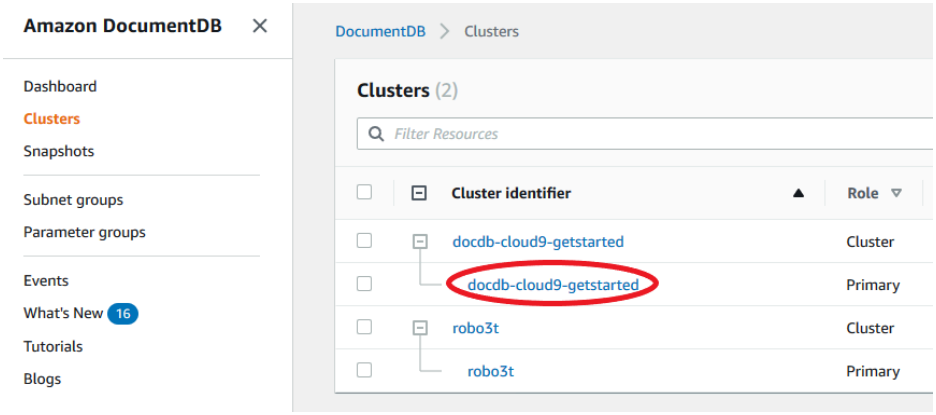
To modify a specific Amazon DocumentDB instance using the console, complete the following steps.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

2. In the navigation pane, choose **Clusters** .

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. Check the box to the left of the instance you wish to modify.
5. Choose **Actions**, and then choose **Modify**.
6. In the **Modify instance: <instance-name>** pane, make the changes that you want. You can make the following changes:

   - **Instance specifications** — The instance identifier and class. Instance identifier naming constraints:
     - **Instance identifier** — Enter a name that is unique for all instances owned by your AWS account in the current Region. The instance identifier must contain [1—63] alphanumeric characters or hyphens, have a letter as the first character, and cannot end with a hyphen or contain two consecutive hyphens.
     - **Instance class** — From the drop-down menu, select an instance class for your Amazon DocumentDB instance. For more information, see Managing instance classes.
   - **Certificate authority** — Server certificate for this instance. For more information, see Updating your Amazon DocumentDB TLS certificates.
   - **Failover** — During failover, the instance with the highest promotion tier will be promoted to primary. For more information, see Amazon DocumentDB failover.

- **Maintenance** — The maintenance window in which pending modifications or patches are applied to instances in the cluster.

7. When you have finished, choose **Continue** to see a summary of your changes.
8. After verifying your changes, you can apply them immediately or during the next maintenance window under **Scheduling of modifications**. Choose **Modify instance** to save your changes. Alternatively, you can choose **Cancel** to discard your changes.

It takes a few minutes for your changes to be applied. You can use the instance only when its status is *available*. You can monitor the instance's status using the console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB instance's status.

Using the AWS CLI

To modify a specific Amazon DocumentDB instance using the AWS CLI, use the `modify-db-instance` with the following parameters. For more information, see ModifyDBInstance. The following code modifies the instance class to `db.r5.large` for the instance `sample-instance`.

Parameters

- **`--db-instance-identifier`** — Required. The identifier for the instance to be modified.
- **`--db-instance-class`** — Optional. The new compute and memory capacity of the instance; for example, `db.r5.large`. Not all instance classes are available in all AWS Regions. If you modify the instance class, an outage occurs during the change. The change is applied during the next maintenance window, unless `ApplyImmediately` is specified as true for this request.
- **`--apply-immediately`** or **`--no-apply-immediately`** — Optional. Specifies whether this modification should be applied immediately or wait until the next maintenance window. If this parameter is omitted, the modification is performed during the next maintenance window.

Example

For Linux, macOS, or Unix:

```
aws docdb modify-db-instance \
       --db-instance-identifier sample-instance \
```

```
       --db-instance-class db.r5.large \
       --apply-immediately
```

For Windows:

```
aws docdb modify-db-instance ^
       --db-instance-identifier sample-instance ^
       --db-instance-class db.r5.large ^
       --apply-immediately
```

Output from this operation looks something like the following.

```
{
    "DBInstances": [
        {
            "DBInstanceIdentifier": "sample-instance-1",
            "DBInstanceClass": "db.r5.large",
            "Engine": "docdb",
            "DBInstanceStatus": "modifying",
            "Endpoint": {
                "Address": "sample-instance-1.node.us-east-1.docdb.amazonaws.com",
                "Port": 27017,
                "HostedZoneId": "ABCDEFGHIJKLM"
            },
            "InstanceCreateTime": "2020-01-10T22:18:55.921Z",
            "PreferredBackupWindow": "02:00-02:30",
            "BackupRetentionPeriod": 1,
            "VpcSecurityGroups": [
                {
                    "VpcSecurityGroupId": "sg-abcd0123",
                    "Status": "active"
                }
            ],
            "AvailabilityZone": "us-east-1a",
            "DBSubnetGroup": {
                "DBSubnetGroupName": "default",
                "DBSubnetGroupDescription": "default",
                "VpcId": "vpc-abcd0123",
                "SubnetGroupStatus": "Complete",
                "Subnets": [
                    {
                        "SubnetIdentifier": "subnet-abcd0123",
                        "SubnetAvailabilityZone": {
```

```
                            "Name": "us-east-1a"
                        },
                        "SubnetStatus": "Active"
                    },
                    {
                        "SubnetIdentifier": "subnet-abcd0123",
                        "SubnetAvailabilityZone": {
                            "Name": "us-east-1b"
                        },
                        "SubnetStatus": "Active"
                    }
                ]
            },
            "PreferredMaintenanceWindow": "sun:10:57-sun:11:27",
            "PendingModifiedValues": {
                "DBInstanceClass": "db.r5.large"
            },
            "EngineVersion": "3.6.0",
            "AutoMinorVersionUpgrade": true,
            "PubliclyAccessible": false,
            "DBClusterIdentifier": "sample-cluster",
            "StorageEncrypted": true,
            "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/wJalrXUtnFEMI/
K7MDENG/bPxRfiCYEXAMPLEKEY",
            "DbiResourceId": "db-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
            "CACertificateIdentifier": "rds-ca-2019",
            "PromotionTier": 1,
            "DBInstanceArn": "arn:aws:rds:us-east-1:123456789012:db:sample-
instance-1",
            "EnabledCloudwatchLogsExports": [
                "profiler"
            ]
        }
    ]
}
```

It takes a few minutes for your modifications to be applied. You can use the instance only when its status is *available*. You can monitor the instance's status using the AWS Management Console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB instance's status.

Rebooting an Amazon DocumentDB instance

Occasionally, you might need to reboot your Amazon DocumentDB instance, usually for maintenance reasons. If you make certain changes, such as changing the cluster parameter group that is associated with a cluster, you must reboot the instances in the cluster for the changes to take effect. You can reboot a specified instance using the AWS Management Console or the AWS CLI.

Rebooting an instance restarts the database engine service. Rebooting results in a momentary outage, during which the instance status is set to `rebooting`. An Amazon DocumentDB event is created when the reboot is completed.

Rebooting an instance doesn't result in a failover. To failover an Amazon DocumentDB cluster, use the AWS Management Console or the AWS CLI operation `failover-db-cluster`. For more information, see Amazon DocumentDB failover.

You can't reboot your instance if it isn't in the *available* state. Your database can be unavailable for several reasons, such as a previously requested modification, or a maintenance-window action. For more information on instance states, see Monitoring an Amazon DocumentDB instance's status.

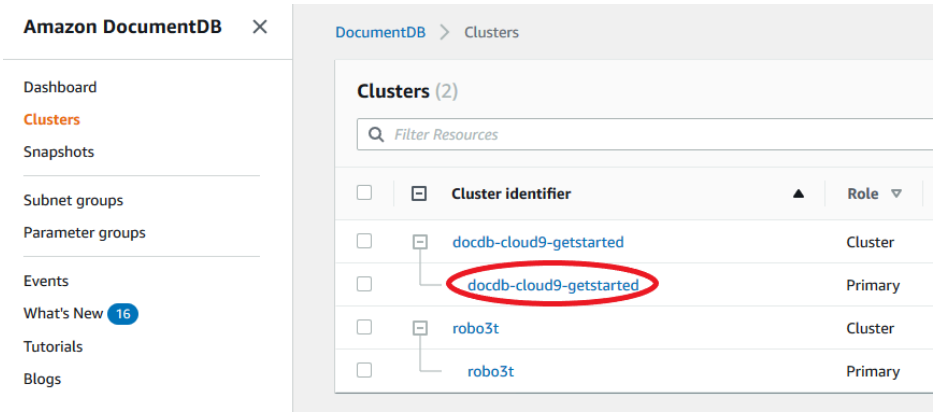
Using the AWS Management Console

The following procedure reboots an instance that you specify using the console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters** .

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. Check the box to the left of the instance you wish to reboot.
5. Choose **Actions**, choose **Reboot**, and then choose **Reboot** to confirm your reboot.

It takes a few minutes for your instance to reboot. You can use the instance only when its status is *available*. You can monitor the instance's status using the console or the AWS CLI. For more information, see Monitoring an Amazon DocumentDB instance's status.

Using the AWS CLI

To reboot an Amazon DocumentDB instance, use the `reboot-db-instance` operation with the `--db-instance-identifier` parameter. This parameter specifies the identifier for the instance to be rebooted.

The following code reboots the instance `sample-instance`.

Example

For Linux, macOS, or Unix:

```
aws docdb reboot-db-instance \
       --db-instance-identifier sample-instance
```

For Windows:

```
aws docdb reboot-db-instance ^
       --db-instance-identifier sample-instance
```

Output from this operation looks something like the following.

```
{
    "DBInstance": {
```

```
        "DBInstanceIdentifier": "sample-instance",
        "DBInstanceClass": "db.r5.large",
        "Engine": "docdb",
        "DBInstanceStatus": "rebooting",
        "Endpoint": {
            "Address": "sample-instance.node.us-east-1.docdb.amazonaws.com",
            "Port": 27017,
            "HostedZoneId": "ABCDEFGHIJKLM"
        },
        "InstanceCreateTime": "2020-03-27T08:05:56.314Z",
        "PreferredBackupWindow": "02:00-02:30",
        "BackupRetentionPeriod": 1,
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcd0123",
                "Status": "active"
            }
        ],
        "AvailabilityZone": "us-east-1c",
        "DBSubnetGroup": {
            "DBSubnetGroupName": "default",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-abcd0123",
            "SubnetGroupStatus": "Complete",
            "Subnets": [
                {
                    "SubnetIdentifier": "subnet-abcd0123",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1a"
                    },
                    "SubnetStatus": "Active"
                },
                {
                    "SubnetIdentifier": "subnet-wxyz0123",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1b"
                    },
                    "SubnetStatus": "Active"
                }
            ]
        },
        "PreferredMaintenanceWindow": "sun:06:53-sun:07:23",
        "PendingModifiedValues": {},
        "EngineVersion": "3.6.0",
```

```
        "AutoMinorVersionUpgrade": true,
        "PubliclyAccessible": false,
        "DBClusterIdentifier": "sample-cluster",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DbiResourceId": "db-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "CACertificateIdentifier": "rds-ca-2019",
        "PromotionTier": 1,
        "DBInstanceArn": "arn:aws:rds:us-east-1:<accountID>:db:sample-instance",
        "EnabledCloudwatchLogsExports": [
            "profiler"
        ]
    }
}
```

It takes a few minutes for your instance to reboot. You can use the instance only when its status is *available*. You can monitor the instance's status using the console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB instance's status.

Deleting an Amazon DocumentDB instance

You can delete your Amazon DocumentDB instance using either the AWS Management Console or the AWS CLI. To delete an instance, the instance must be in the *available* state. You cannot delete an instance that is stopped. If the Amazon DocumentDB cluster that contains your instance is stopped, first start the cluster, wait for the instance to become *available*, and then delete the instance. For more information, see Stopping and starting an Amazon DocumentDB cluster.

> **Note**
>
> Amazon DocumentDB stores all of your data in the cluster volume. The data persists in that cluster volume, even if you remove all the instances from your cluster. If you need to access the data again, you can add an instance to the cluster at any time and pick up where you left off.

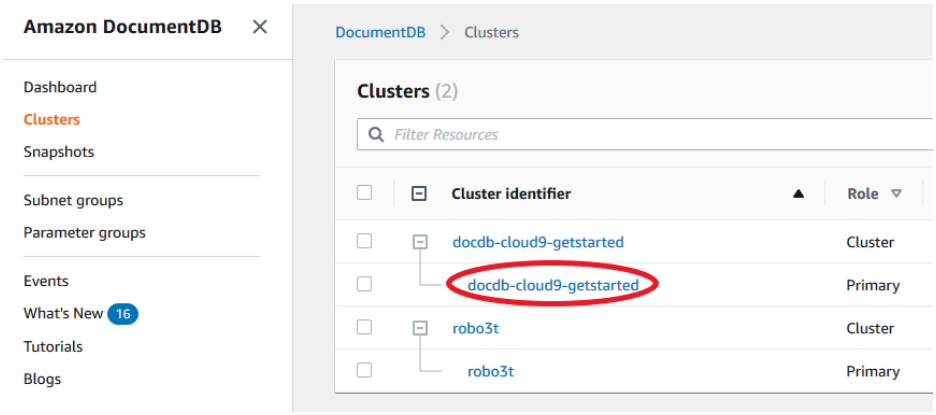
Using the AWS Management Console

The following procedure deletes a specified Amazon DocumentDB instance using the console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters** .

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. Check the box to the left of the instance you wish to delete.
5. Select **Actions**, and then choose **Delete**.

   1. If you are deleting the last instance in your cluster:

      - **Create final cluster snapshot?** — Choose **Yes** if you want to create a final snapshot before the cluster is deleted. Otherwise, choose **No**.
      - **Final snapshot name** — If you choose to create a final snapshot, enter the cluster snapshot identifier of the new cluster snapshot created.
      - **Delete <instance-name> instance?** — Enter the phrase **delete entire cluster** into the field to confirm the deletion.

   2. If you are not deleting the last instance in your cluster:

      - **Delete <instance-name> instance?** — Enter the phrase **delete me** into the field to confirm the deletion.

6. Select **Delete** to delete the instance.

It takes several minutes for an instance to be deleted. To monitor the status of an instance, see Monitoring an Amazon DocumentDB instance's status.

Using the AWS CLI

The following procedure deletes an Amazon DocumentDB instance using the AWS CLI.

1. **First, determine how many instances are in your Amazon DocumentDB cluster:**

   To determine how many instances are in your cluster, run the `describe-db-clusters` command, as follows.

   ```
   aws docdb describe-db-clusters \
       --db-cluster-identifier sample-cluster \
       --query 'DBClusters[*].
   [DBClusterIdentifier,DBClusterMembers[*].DBInstanceIdentifier]'
   ```

   Output from this operation looks something like the following.

   ```
   [
       [
           "sample-cluster",
           [
               "sample-instance-1",
               "sample-instance-2"
           ]
       ]
   ]
   ```

2. **If there are more than one instances in your Amazon DocumentDB cluster:**

   To delete a specified Amazon DocumentDB instance, use the `delete-db-instance` command with the `--db-instance-identifier` parameter, as shown below. It takes several minutes for an instance to be deleted. To monitor the status of an instance, see Monitoring an Amazon DocumentDB instance's status.

   ```
   aws docdb delete-db-instance \
          --db-instance-identifier sample-instance-2
   ```

   Output from this operation looks something like the following.

```
{
    "DBInstance": {
        "DBInstanceIdentifier": "sample-instance-2",
        "DBInstanceClass": "db.r5.large",
        "Engine": "docdb",
        "DBInstanceStatus": "deleting",
        "Endpoint": {
            "Address": "sample-instance-2.node.us-east-1.docdb.amazonaws.com",
            "Port": 27017,
            "HostedZoneId": "ABCDEFGHIJKLM"
        },
        "InstanceCreateTime": "2020-03-27T08:05:56.314Z",
        "PreferredBackupWindow": "02:00-02:30",
        "BackupRetentionPeriod": 1,
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcd0123",
                "Status": "active"
            }
        ],
        "AvailabilityZone": "us-east-1c",
        "DBSubnetGroup": {
            "DBSubnetGroupName": "default",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-6242c31a",
            "SubnetGroupStatus": "Complete",
            "Subnets": [
                {
                    "SubnetIdentifier": "subnet-abcd0123",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1a"
                    },
                    "SubnetStatus": "Active"
                },
                {
                    "SubnetIdentifier": "subnet-wxyz0123",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1b"
                    },
                    "SubnetStatus": "Active"
                }
            ]
        },
```

```
        "PreferredMaintenanceWindow": "sun:06:53-sun:07:23",
        "PendingModifiedValues": {},
        "EngineVersion": "3.6.0",
        "AutoMinorVersionUpgrade": true,
        "PubliclyAccessible": false,
        "DBClusterIdentifier": "sample-cluster",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DbiResourceId": "db-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "CACertificateIdentifier": "rds-ca-2019",
        "PromotionTier": 1,
        "DBInstanceArn": "arn:aws:rds:us-east-1:<accountID>:db:sample-instance-2",
        "EnabledCloudwatchLogsExports": [
            "profiler"
        ]
    }
}
```

3. **If the instance your want to delete is the last instance in your Amazon DocumentDB cluster:**

   If you delete the last instance in an Amazon DocumentDB cluster, you also delete that cluster and the automatic snapshots and continuous backups associated with that cluster.

   To delete the last instance in your cluster, you can delete the cluster and optionally create a final snapshot. For more information, see Deleting an Amazon DocumentDB cluster.

Deletion protection

Deleting the last instance of an Amazon DocumentDB cluster will also delete the cluster, as well as the automatic snapshots and continuous backups associated with that cluster. Amazon DocumentDB enforces deletion protection for a cluster whether you perform the delete operation using the AWS Management Console or the AWS CLI. If deletion protection is enabled, you can't delete a cluster.

To delete a cluster that has deletion protection enabled, you must first modify the cluster and disable deletion protection. For more information, see Deleting an Amazon DocumentDB cluster.

Managing instance classes

The instance class determines the computation and memory capacity of an Amazon DocumentDB (with MongoDB compatibility) instance. The instance class you need depends on your processing power and memory requirements.

Amazon DocumentDB supports the R4, R5, R6G, R8G, T3, and T4G families of instance classes. These classes are current-generation instance classes that are optimized for memory-intensive applications. For the specifications on these classes, see Instance class specifications.

Topics

- Determining an instance class
- Changing an instance's class
- Supported instance classes by Region
- Instance class specifications

Determining an instance class

To determine the class of an instance, you can use the AWS Management Console or the `describe-db-instances` AWS CLI operation.

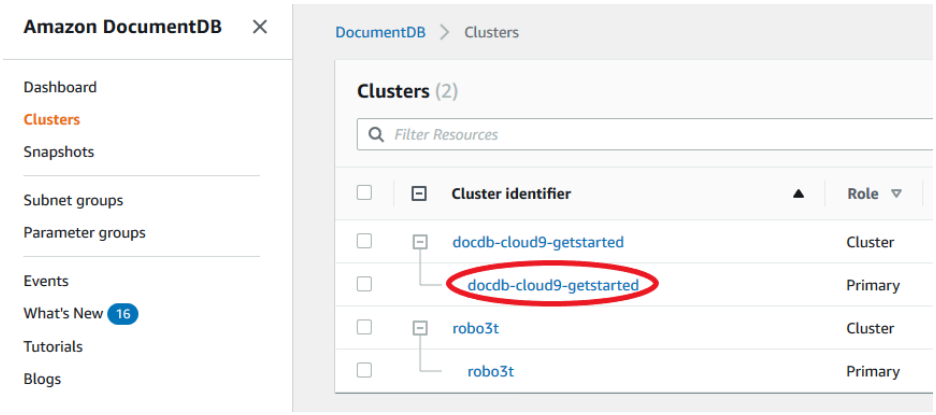
Using the AWS Management Console

To determine the instance class for your cluster's instances, complete the following steps in the console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters** to find the instance that you're interested in.

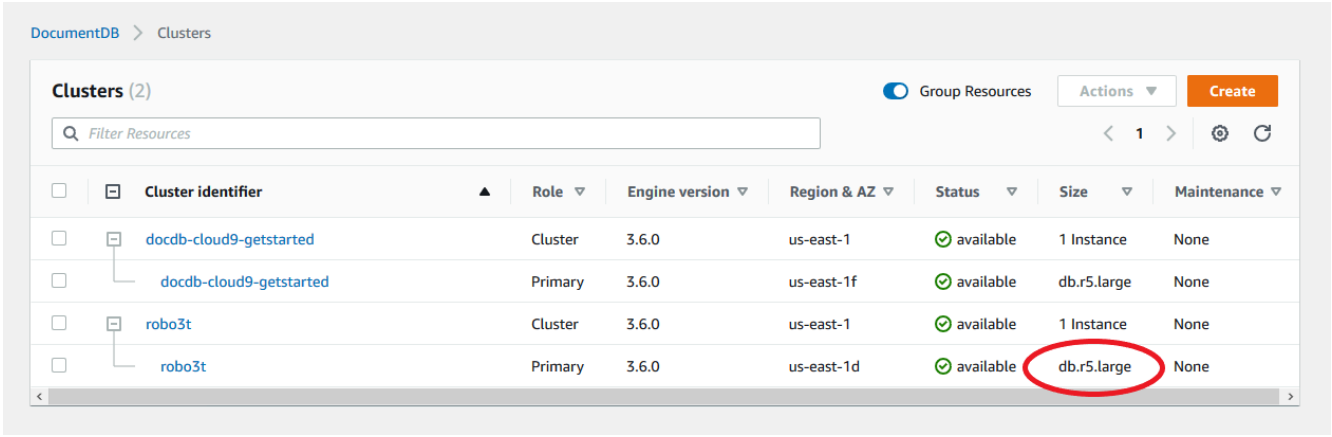
   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. In the list of instances, expand the cluster to find the instances you are interested in. Find the instance that you want. Then, look at the **Size** column of the instance's row to see its instance class.

   In the following image, the instance class for instance `robo3t` is `db.r5.4xlarge`.



Using the AWS CLI

To determine the class of an instance using the AWS CLI, use the `describe-db-instances` operation with the following parameters.

- **`--db-instance-identifier`** — Optional. Specifies the instance that you want to find the instance class for. If this parameter is omitted, `describe-db-instances` returns a description for up to 100 of your instances.
- **`--query`** — Optional. Specifies the members of the instance to include in the results. If this parameter is omitted, all instance members are returned.

Example

The following example finds the instance name and class for the instance `sample-instance-1`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceClass]' \
    --db-instance-identifier sample-instance-1
```

For Windows:

```
aws docdb describe-db-instances ^
    --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceClass]' ^
    --db-instance-identifier sample-instance-1
```

Output from this operation looks something like the following.

```
[
    [
        "sample-instance-1",
        "db.r5.large"
    ]
```

Example

The following example finds the instance name and class for up to 100 Amazon DocumentDB instances.

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceClass]' \
    --filter Name=engine,Values=docdb
```

For Windows:

```
aws docdb describe-db-instances ^
    --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceClass]' ^
    --filter Name=engine,Values=docdb
```

Output from this operation looks something like the following.

```
[
    [
        "sample-instance-1",
        "db.r5.large"
    ],
    [
        "sample-instance-2",
        "db.r5.large"
    ],
    [
        "sample-instance-3",
        "db.r5.4xlarge"
    ],
    [
        "sample-instance-4",
        "db.r5.4xlarge"
    ]
]
```

For more information, see Describing Amazon DocumentDB instances.

Changing an instance's class

You can change the instance class of your instance using the AWS Management Console or the AWS CLI. For more information, see Modifying an Amazon DocumentDB instance.

Supported instance classes by Region

Amazon DocumentDB supports the following instance classes:

- R8G—Latest generation of memory-optimized instances powered by Arm-based AWS Graviton4 processors that provide up to 30% better performance over R6G instances.
- R6G—Memory-optimized instances powered by Arm-based AWS Graviton2 processors that provide up to 30% better performance over R5 instances at 5% less cost.
- R6GD—Memory-optimized R6G instances with local non-volatile memory express (NVMe)-based Solid-State Drive (SSD) storage for ephemeral data.
- R5—Memory-optimized instances that provide up to 100% better performance over R4 instances for the same instance cost.

- R4—Previous generation of memory-optimized instances.
- T4G—Latest-generation low cost burstable general-purpose instance type powered by Arm-based AWS Graviton2 processors that provides a baseline level of CPU performance, delivering up to 35% better price performance over T3 instances and ideal for running applications with moderate CPU usage that experience temporary spikes in usage.
- T3—Low cost burstable general-purpose instance type that provides a baseline level of CPU performance with the ability to burst CPU usage at any time for as long as required.

For detailed specifications on the instance classes, see Instance class specifications.

> **Note**
>
> I/O-Optimized storage is only available on engine versions 5.0 and 8.0 (instance-based clusters).

A particular instance class may or may not be supported in a given Region. The following table specifies which instance classes are supported by Amazon DocumentDB in each Region.

Supported instance classes by Region

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
US East (Ohio)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
US East (N. Virginia)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
US West (Oregon)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
Africa (Cape Town)			Supported	Supported		Supported	Supported	Supported
South America (São Paulo)	Supported	Supported	Supported	Supported		Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Asia Pacific (Hong Kong)	Supported		Supported	Supported		Supported	Supported	Supported
Asia Pacific (Taipei)			Supported			Supported	Supported	
Asia Pacific (Hyderabad)			Supported	Supported		Supported	Supported	Supported
Asia Pacific (Malaysia)			Supported			Supported	Supported	Supported
Asia Pacific (Mumbai)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Osaka)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Seoul)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Sydney)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Jakarta)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Asia Pacific (Melbourne)			Supported	Supported		Supported	Supported	Supported
Asia Pacific (New Zealand)			Supported			Supported	Supported	
Asia Pacific (Singapore)	Supported	Supported	Supported	Supported		Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Asia Pacific (Thailand)			Supported			Supported	Supported	Supported
Asia Pacific (Tokyo)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Canada (Central)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Canada West (Calgary)			Supported	Supported		Supported	Supported	Supported
Europe (Frankfurt)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Zurich)		Supported	Supported	Supported		Supported	Supported	Supported
Europe (Ireland)	Supported	Supported	Supported	Supported	Supported	Supported	Supported	Supported
Europe (London)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Milan)			Supported	Supported		Supported	Supported	Supported
Europe (Paris)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Spain)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Europe (Stockholm)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
Mexico (Central)			Supported			Supported	Supported	Supported
Middle East (UAE)			Supported	Supported		Supported	Supported	Supported
China (Beijing)	Supported	Supported	Supported	Supported		Supported	Supported	Supported
China (Ningxia)	Supported		Supported	Supported		Supported	Supported	Supported

Region	R8G	R6GD	R6G	R5	R4	T4G	T3	Serverless
Israel (Tel Aviv)			Supported	Supported		Supported	Supported	Supported
AWS GovCloud (US-West)	Supported	Supported	Supported	Supported			Supported	Supported
AWS GovCloud (US-East)		Supported	Supported	Supported		Supported	Supported	Supported

Instance class specifications

The following table provides details of the Amazon DocumentDB instance classes, including which instance types are supported in each class. You can find explanations for each table column below the table.

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
R8G – Current Generation Memory-Optimized Instance Class based on Graviton4 R8G is the newest instance family (Graviton4), available on engine versions 5.0 and 8.0 only.						
`db.r8g.large`	2	16	-	30	0.937 / 12.5	5.0.0 and 8.0.0
`db.r8g.xlarge`	4	32	-	60	1.875 / 12.5	5.0.0 and 8.0.0
`db.r8g.2xlarge`	8	64	-	121	3.75 / 15.0	5.0.0 and 8.0.0
`db.r8g.4xlarge`	16	128	-	243	7.5 / 15.0	5.0.0 and 8.0.0

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
db.r8g.8xlarge	32	256	-	488	15	5.0.0 and 8.0.0
db.r8g.12xlarge	48	384	-	732	22	5.0.0 and 8.0.0
db.r8g.16xlarge	64	512	-	987	30	5.0.0 and 8.0.0
db.r8g.24xlarge	96	768	-	1484	40	5.0.0 and 8.0.0
db.r8g.48xlarge	192	1536	-	2967	50	5.0.0 and 8.0.0
R6G – Current Generation Memory-Optimized Instance Class based on Graviton2						
db.r6g.large	2	16	-	32	0.75 / 10	4.0.0, 5.0.0, and 8.0.0
db.r6g.xlarge	4	32	-	63	1.25 / 10	4.0.0, 5.0.0, and 8.0.0
db.r6g.2xlarge	8	64	-	126	2.5 / 10	4.0.0, 5.0.0, and 8.0.0
db.r6g.4xlarge	16	128	-	252	5.0 / 10	4.0.0, 5.0.0, and 8.0.0

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
db.r6g.8xlarge	32	256	-	504	12	4.0.0, 5.0.0, and 8.0.0
db.r6g.12xlarge	48	384	-	756	20	4.0.0, 5.0.0, and 8.0.0
db.r6g.16xlarge	64	512	-	1008	25	4.0.0, 5.0.0, and 8.0.0
R6GD – Current Generation NVMe-backed Instance Class based on Graviton2						
db.r6gd.xlarge	4	32	173	64	1.25 / 10	5.0.0 and 8.0.0
db.r6gd.2xlarge	8	64	346	128	2.5 / 10	5.0.0 and 8.0.0
db.r6gd.4xlarge	16	128	694	256	5.0 / 10	5.0.0 and 8.0.0
db.r6gd.8xlarge	32	256	1388	512	12	5.0.0 and 8.0.0
db.r6gd.12xlarge	48	384	2082	768	20	5.0.0 and 8.0.0
db.r6gd.16xlarge	64	512	2776	1024	25	5.0.0 and 8.0.0
R5 – Previous Generation Memory-Optimized Instance Class						

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
db.r5.large	2	16	-	31	0.75 / 10	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.xlarge	4	32	-	62	1.25 / 10	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.2xlarge	8	64	-	124	2.5 / 10	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.4xlarge	16	128	-	249	5.0 / 10	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.8xlarge	32	256	-	504	10	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.12xlarge	48	384	-	748	12	3.6.0, 4.0.0, 5.0.0, and 8.0.0

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
db.r5.16xlarge	64	512	-	1008	20	3.6.0, 4.0.0, 5.0.0, and 8.0.0
db.r5.24xlarge	96	768	-	1500	25	3.6.0, 4.0.0, 5.0.0, and 8.0.0
R4 – Previous Generation Memory-Optimized Instance Class R4 instances are only supported on engine version 3.6. Note that Amazon DocumentDB 3.6 reaches end of standard support on March 30, 2026. Extended Support (paid) is available until March 2029.						
db.r4.large	2	15.25	-	30	0.75 / 10	3.6.0 only
db.r4.xlarge	4	30.5	-	60	1.25 / 10	3.6.0 only
db.r4.2xlarge	8	61	-	120	2.5 / 10	3.6.0 only
db.r4.4xlarge	16	122	-	240	5.0 /10	3.6.0 only
db.r4.8xlarge	32	244	-	480	10	3.6.0 only
db.r4.16xlarge	64	488	-	960	25	3.6.0 only

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]
T4G – Latest Generation Burstable Performance Instance Classes based on Graviton2 T-series instances run in Unlimited CPU burst mode. Burst usage beyond the baseline is billed extra. Not supported for Global Clusters.						
`db.t4g.medium`	2	4	-	8.13	0.256 / 5	4.0.0, 5.0.0, and 8.0.0
T3 – Previous Generation Burstable Performance Instance Classes						
`db.t3.medium`	2	4	-	7.5	0.256 / 5	3.6.0, 4.0.0, 5.0.0, and 8.0.0

Instance class	vCPU[1]	Memory (GiB)[2]	NVMe SSD tiered cache (GiB)[3]	Max. temp. storage (GiB)[4]	Baseline / burst bandwidth (Gbps)[5]	Supporting Engines[6]

1. **vCPU** — The number of virtual central processing units (CPUs). A virtual CPU is a unit of capacity that you can use to compare instance classes. Instead of purchasing or leasing a particular processor to use for several months or years, you are renting capacity by the hour. Our goal is to provide a consistent amount of CPU capacity no matter what the actual underlying hardware.
2. **Memory (GiB)** — The RAM, in gigabytes, that is allocated to the instance. There is often a consistent ratio between memory and vCPU.
3. **NVMe SSD tiered cache** — The space on the SSD volume, measured in gigabytes, allocated as extended cache for storing ephemeral data. This cache is only available in NVMe-backed instances.
4. **Max. temp. storage (GiB)** — The space, measured in gigabytes, allocated to the instance for non-persistent temporary file storage. For NVMe-backed instances, this storage is hosted on an NVMe-based SSD volume. In all other instances, it is hosted on Amazon Elastic Block Store (EBS).
5. **Baseline / burst bandwidth (Gbps)** — Burst bandwidth represents the maximum bandwidth in gigabits per second. Divide by 8 to get the expected throughput in gigabytes per second. Instances of size 4xlarge and smaller have a baseline bandwidth. To meet additional demand, they can use a network I/O credit mechanism to burst beyond their baseline bandwidth . Instances can use burst bandwidth for a limited time, typically from 5 to 60 minutes, depending on the instance size.
6. **Supporting Engines** — The Amazon DocumentDB engines that support the instance class.

NVMe-backed instances

NVMe-backed instances offer up to 7x faster query performance for applications with large datasets that exceed the memory of a regular instance. These instances leverage the local non-volatile memory express (NVMe)-based Solid-State Drive (SSD) storage available on r6gd instances to store ephemeral data, reducing network-based storage access, and improving read latency and throughput.

The local SSD space is divided into two sections:

- **Tiered cache** — Approximately 73% of the local SSD is allocated as a database cache, allowing the system to store up to five times more database pages than the main memory alone. The local SSD acts as a second-tier cache, while the existing in-memory buffer cache remains the first-tier cache. The query only accesses external storage if both the buffer cache and the SSD cache experience a miss.
- **Temporary storage** — The remaining 27% is reserved for non-persistent temporary file storage, used for complex queries involving sorts or resource-intensive operations like index builds. On regular instances, the temporary space resides on an Amazon Elastic Block Store (EBS) volume. The locally hosted temporary storage on the SSD reduces query latency involving sorts by up to two times and accelerates resource-intensive operations like index builds.

The details regarding the type of NVMe-backed instances supported and its specification can be found in Instance class specifications.

Topics

- Recommended use cases for NVMe-backed instances
- Using NVMe-backed instances with Amazon DocumentDB
- Monitoring NVMe-backed instances

Recommended use cases for NVMe-backed instances

Use NVMe-backed instances in the following scenarios:

- **Read-heavy workloads** — If your workload is read-intensive, and your dataset is larger than the buffer cache, indicated by low `BufferCacheHitRatio` and high `ReadIOPS` metrics, NVMe-backed instances can provide performance benefits.
- **Update-heavy workloads** — If your workload is update-intensive, and garbage collection is unable to keep up due to read latency on network storage, NVMe-backed instances could help mitigate the issue.

NVMe-backed instances can benefit various use cases, including:

- **Internet-scale applications** — Applications such as payment processing, billing, and e-commerce with strict performance Service Level Agreements (SLAs) can leverage the performance advantages of NVMe-backed instances.
- **Real-time reporting dashboards** — Dashboards that run hundreds of queries for metrics/data collection can benefit from the low latency and high throughput of NVMe-backed instances.
- **Generative AI applications** — Applications using vector search to find exact or nearest neighbors across millions of vector embeddings can leverage the high performance of NVMe-backed instances.

Using NVMe-backed instances with Amazon DocumentDB

To use NVMe-backed instances of Amazon DocumentDB:

- Create an Amazon DocumentDB cluster and add one of the NVMe-backed instance classes. For more information, see Creating an Amazon DocumentDB cluster.
- Alternatively, modify an existing Amazon DocumentDB cluster to use one of the NVMe-backed instance classes. For more information, see Modifying an Amazon DocumentDB cluster.

To check the availability of NVMe-backed instances across different AWS Regions, see Supported instance classes by Region.

If you want to switch back from an NVMe-backed instance to a regular instance, modify the database instance class of your Amazon DocumentDB instance to a similar instance class without the NVMe storage. For example, if your current instance class is 'db.r6gd.4xlarge', choose 'db.r6g.4xlarge' to switch back. For more information, see Modifying an Amazon DocumentDB cluster.

Monitoring NVMe-backed instances

In addition to the regular instance metrics available in Amazon CloudWatch, NVMe-backed instances emit additional metrics specific to NVMe-based SSD storage, IOPS, and throughput.

```
NVMeStorageCacheHitRatio
FreeNVMeStorage
ReadIOPSNVMeStorage
ReadLatencyNVMeStorage
ReadThroughputNVMeStorage
WriteIOPSNVMeStorage
```

```
WriteLatencyNVMeStorage
WriteThroughputNVMeStorage
```

For more information about these metrics, see NVMe-backed instance metrics

Managing Amazon DocumentDB subnet groups

A virtual private cloud (VPC) is a virtual network dedicated to your AWS account. It is logically isolated from other virtual networks in the AWS Cloud. You can launch your AWS resources, such as Amazon DocumentDB clusters, into your Amazon VPC. You can specify an IP address range for the VPC, add subnets, associate security groups, and configure route tables.

A subnet is a range of IP addresses in your Amazon VPC. You can launch AWS resources into a specified subnet. Use a *public* subnet for resources that must be connected to the internet. Use a *private* subnet for resources that won't be connected to the internet. For more information about public and private subnets, see VPC and Subnet Basics in the *Amazon Virtual Private Cloud User Guide*.

A DB subnet group is a collection of subnets that you create in a VPC that you then designate for your clusters. A subnet group allows you to specify a particular VPC when creating clusters. If you use the `default` subnet group, it spans all subnets in the VPC.

Each DB subnet group should have subnets in at least two Availability Zones in a given Region. When creating a DB cluster in a VPC, you must select a DB subnet group. Amazon DocumentDB uses that DB subnet group and your preferred Availability Zone to select a subnet and an IP address within that subnet to associate with your cluster. If the primary instance fails, Amazon DocumentDB can promote a corresponding replica instance to be the new primary. It can then create a new replica instance using an IP address of the subnet in which the previous primary was located.

When Amazon DocumentDB creates an instance in a VPC, it assigns a network interface to your cluster by using an IP address selected from your DB subnet group. Always use the DNS name because the underlying IP address can change during failover. For more information, see Amazon DocumentDB endpoints.

For information about creating your own VPC and subnets, see Working with VPCs and Subnets in the *Amazon Virtual Private Cloud User Guide*.

Topics

- Creating an Amazon DocumentDB subnet group
- Describing an Amazon DocumentDB subnet group
- Modifying an Amazon DocumentDB subnet group
- Deleting an Amazon DocumentDB subnet group

Creating an Amazon DocumentDB subnet group

When creating an Amazon DocumentDB cluster, you must choose a Amazon VPC and corresponding subnet group within that Amazon VPC to launch your cluster. Subnets determine the availability zone and IP range within the availability zone that you want to use to launch an instance.

A subnet group is a named set of subnets (or AZs) that allows you to specify the availability zones that you want to use to for launching Amazon DocumentDB instances. For example, in a cluster with three instances, it is recommended that each of those instances are provisioned in separate AZs—doing so optimizes for high availability. Thus, if a single AZ fails, it will only affect a single instance.

Amazon DocumentDB instances can be provisioned in up to three Availability Zones. Even if a subnet group has more than three subnets, you can only use three of those subnets to create an Amazon DocumentDB cluster. When you create a subnet group, choose only the three subnets where you want to deploy your instances.

For example: A cluster is created and Amazon DocumentDB choose AZs {1A, 1B, and 1C}. If you attempt to create an instance in AZ {1D} the API call will fail. However, if you choose to create an instance, without specifying the particular AZ, then Amazon DocumentDB will choose an AZ on your behalf. Amazon DocumentDB uses an algorithm to load balance the instances across AZs to help you achieve high availability. If three instances are provisioned, by default, they will be provisioned across three AZs and will not be provisioned all in a single AZ.

Best practices

- Unless you have a specific reason, always create a subnet group with three subnets. This ensures that clusters with three or more instances will be able to achieve higher availability as instances will be provisioned across three AZs.
- Always spread instances across multiple AZs to achieve high availability. Never place all instances for a cluster in a single AZ.

- Because failover events can happen at any time, you should not assume that a primary instance or replica instances will always be in a particular AZ.

How to create a subnet group

You can use the AWS Management Console or AWS CLI to create an Amazon DocumentDB subnet group:

Using the AWS Management Console

Use the following steps to create an Amazon DocumentDB subnet group.

To create an Amazon DocumentDB subnet group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Subnet groups**, then choose **Create**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. On the **Create subnet group** page:

   a. In the **Subnet group details** section:

      i. **Name**—Enter a meaningful name for the subnet group.

      ii. **Description**—Enter a description for the subnet group.

   b. In the **Add subnets** section:

      i. **VPC**—In the list, choose a VPC for this subnet group.

      ii. Do one of the following:

      - To include all subnets in the chosen VPC, choose **Add all the subnets related to this VPC**.

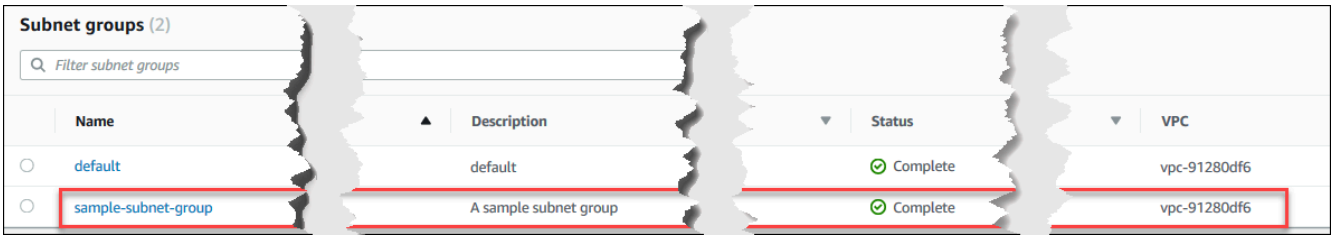
- To specify subnets for this subnet group, do the following for each Availability Zone for which you want to include subnets. You must include at least two Availability Zones.

  A. **Availability zone**—In the list, choose an Availability Zone.

  B. **Subnet**—In the list, choose a subnet from the chosen Availability Zone for this subnet group.

  C. Choose **Add subnet**.

4. Choose **Create**. When the subnet group is created, it is listed with your other subnet groups.



Using the AWS CLI

Before you can create a subnet group using the AWS CLI, you must first determine which subnets are available. Run the following AWS CLI operation to list the Availability Zones and their subnets.

Parameters:

- **`--db-subnet-group`**—Optional. Specifying a particular subnet group lists the Availability Zones and subnets for that group. Omitting this parameter lists Availability Zones and subnets for all your subnet groups. Specifying the `default` subnet group lists all the VPC's subnets.

Example

For Linux, macOS, or Unix:

```
aws docdb describe-db-subnet-groups \
    --db-subnet-group-name default \
    --query 'DBSubnetGroups[*].[DBSubnetGroupName,Subnets[*].
[SubnetAvailabilityZone.Name,SubnetIdentifier]]'
```

For Windows:

```
aws docdb describe-db-subnet-groups ^
    --db-subnet-group-name default ^
    --query 'DBSubnetGroups[*].[DBSubnetGroupName,Subnets[*].
[SubnetAvailabilityZone.Name,SubnetIdentifier]]'
```

Output from this operation looks something like the following (JSON format).

```
[
    [
        "default",
        [
            [
                "us-east-1a",
                "subnet-4e26d263"
            ],
            [
                "us-east-1c",
                "subnet-afc329f4"
            ],
            [
                "us-east-1e",
                "subnet-b3806e8f"
            ],
            [
                "us-east-1d",
                "subnet-53ab3636"
            ],
            [
                "us-east-1b",
                "subnet-991cb8d0"
            ],
            [
                "us-east-1f",
                "subnet-29ab1025"
            ]
        ]
    ]
]
```

Using the output from the previous operation, you can create a new subnet group. The new subnet group must include subnets from at least two Availability Zones.

Parameters:

- **`--db-subnet-group-name`**—Required. The name for this subnet group.
- **`--db-subnet-group-description`**—Required. The description of this subnet group.
- **`--subnet-ids`**—Required. A list of subnets to include in this subnet group. Example: `subnet-53ab3636`.
- --Tags—Optional. A list of tags (key-value pairs) to attach to this subnet group.

The following code creates the subnet group `sample-subnet-group` with three subnets, `subnet-4e26d263`, `subnet-afc329f4`, and `subnet-b3806e8f`.

For Linux, macOS, or Unix:

```
aws docdb create-db-subnet-group \
    --db-subnet-group-name sample-subnet-group \
    --db-subnet-group-description "A sample subnet group" \
    --subnet-ids subnet-4e26d263 subnet-afc329f4 subnet-b3806e8f \
    --tags Key=tag1,Value=One Key=tag2,Value=2
```

For Windows:

```
aws docdb create-db-subnet-group ^
    --db-subnet-group-name sample-subnet-group ^
    --db-subnet-group-description "A sample subnet group" ^
    --subnet-ids subnet-4e26d263 subnet-afc329f4 subnet-b3806e8f ^
    --tags Key=tag1,Value=One Key=tag2,Value=2
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBSubnetGroup": {
        "DBSubnetGroupDescription": "A sample subnet group",
        "DBSubnetGroupName": "sample-subnet-group",
        "Subnets": [
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1a"
                },
                "SubnetIdentifier": "subnet-4e26d263",
```

```
                "SubnetStatus": "Active"
            },
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1c"
                },
                "SubnetIdentifier": "subnet-afc329f4",
                "SubnetStatus": "Active"
            },
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1e"
                },
                "SubnetIdentifier": "subnet-b3806e8f",
                "SubnetStatus": "Active"
            }
        ],
        "VpcId": "vpc-91280df6",
        "DBSubnetGroupArn": "arn:aws:rds:us-east-1:123SAMPLE012:subgrp:sample-
subnet-group",
        "SubnetGroupStatus": "Complete"
    }
}
```

Describing an Amazon DocumentDB subnet group

You can use the AWS Management Console or the AWS CLI to get the details of an Amazon DocumentDB subnet group.

Using the AWS Management Console

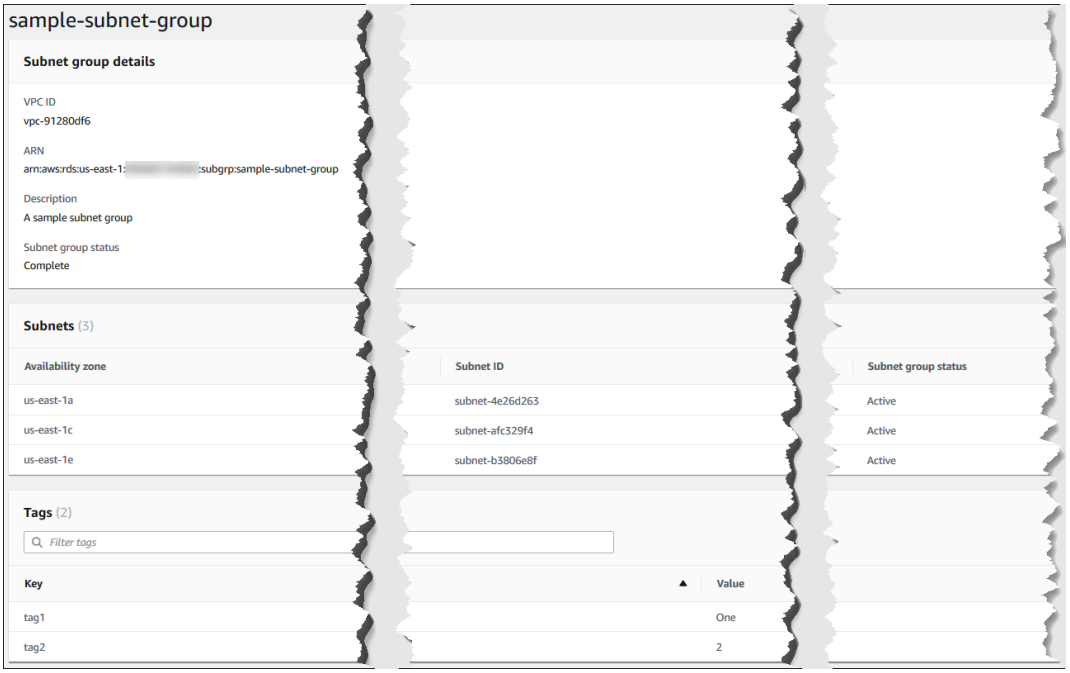
The following procedure shows you how to get the details of an Amazon DocumentDB subnet group.

To find the details of a subnet group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Subnet groups**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. To see the details of a subnet group, choose the name of that subnet group.



Using the AWS CLI

To find the details of an Amazon DocumentDB subnet group, use the `describe-db-subnet-groups` operation with the following parameter.

Parameter

- `--db-subnet=group-name`—Optional. If included, details for the named subnet group are listed. If omitted, details for up to 100 subnet groups are listed.

Example

The following code lists details for the `sample-subnet-group` subnet group that we created in the Creating an Amazon DocumentDB subnet group section.

For Linux, macOS, or Unix:

```
aws docdb describe-db-subnet-groups \
    --db-subnet-group-name sample-subnet-group
```

For Windows:

```
aws docdb describe-db-subnet-groups ^
    --db-subnet-group-name sample-subnet-group
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBSubnetGroup": {
        "DBSubnetGroupArn": "arn:aws:rds:us-east-1:123SAMPLE012:subgrp:sample-
subnet-group",
        "VpcId": "vpc-91280df6",
        "SubnetGroupStatus": "Complete",
        "DBSubnetGroupName": "sample-subnet-group",
        "Subnets": [
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1a"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-4e26d263"
            },
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1c"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-afc329f4"
            },
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1e"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-b3806e8f"
            }
        ],
        "DBSubnetGroupDescription": "A sample subnet group"
```

```
    }
}
```

Modifying an Amazon DocumentDB subnet group

You can use the AWS Management Console or AWS CLI to modify a subnet group's description or to add or remove subnets from an Amazon DocumentDB subnet group. However, you cannot modify the `default` subnet group.

Using the AWS Management Console

You can use the AWS Management Console to change a subnet group's description or to add and remove subnets. Remember that when you're finished, you must have at least two Availability Zones associated with your subnet group.

To modify your subnet group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Subnet groups**. Then choose the button to the left of the subnet group's name. Remember that you can't modify the `default` subnet group.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. Choose **Actions**, and then choose **Modify**.
4. **Description**—To change the description of your subnet group, enter a new description.
5. To change the subnets associated with your subnet group, in the **Add subnets** section, do any one or more of the following:
   - To remove all subnets from this subnet group, choose **Remove all**.
   - To remove specific subnets from this subnet group, choose **Remove** for each subnet you want to remove.
   - To add all the subnets associated with this VPC, choose **Add all the subnets related to this VPC**.

- To add specific subnets to this subnet group, do the following for each Availability Zone for which you want to add a subnet.

  a. **Availability zone**—In the list, choose a new Availability Zone.

  b. **Subnet**—In the list, choose a subnet from the chosen Availability Zone for this subnet group.

  c. Choose **Add subnet**.

6. In the confirmation dialog box:
   - To make these changes to the subnet group, choose **Modify**.
   - To keep the subnet group unchanged, choose **Cancel**.

Using the AWS CLI

You can use the AWS CLI to change a subnet group's description or to add and remove subnets. Remember that when you're finished, you must have at least two Availability Zones associated with your subnet group. You can't modify the `default` subnet group.

Parameters:

- `--db-subnet-group-name`—Required. The name of the Amazon DocumentDB subnet group you are modifying.
- `--subnet-ids`—Required. A list of all the subnets that you want in the subnet group after this change is done.

> **Important**
>
> Any subnets currently in the subnet group that are not included in this list are removed from the subnet group. If you want to keep any of the subnets currently in the subnet group, you must include them in this list.

- `--db-subnet-group-description`—Optional. The description of the subnet group.

Example

The following code modifies the description and replaces the existing subnets with the subnets `subnet-991cb8d0`, `subnet-53ab3636`, and `subnet-29ab1025`.

For Linux, macOS, or Unix:

```
aws docdb modify-db-subnet-group \
    --db-subnet-group-name sample-subnet-group \
    --subnet-ids subnet-991cb8d0 subnet-53ab3636 subnet-29ab1025 \
    --db-subnet-group-description "Modified subnet group"
```

For Windows:

```
aws docdb modify-db-subnet-group ^
    --db-subnet-group-name sample-subnet-group ^
    --subnet-ids subnet-991cb8d0 subnet-53ab3636 subnet-29ab1025 ^
    --db-subnet-group-description "Modified subnet group"
```

Output from this operation looks something like the following (JSON format). Notice that this is the same subnet group that was created in the Creating an Amazon DocumentDB subnet group section. However, the subnets in the subnet group are replaced with those listed in the `modify-db-subnet-group` operation.

```
{
    "DBSubnetGroup": {
        "DBSubnetGroupArn": "arn:aws:rds:us-east-1:123SAMPLE012:subgrp:sample-
subnet-group",
        "DBSubnetGroupDescription": "Modified subnet group",
        "SubnetGroupStatus": "Complete",
        "Subnets": [
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1d"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-53ab3636"
            },
            {
                "SubnetAvailabilityZone": {
                    "Name": "us-east-1b"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-991cb8d0"
            },
            {
                "SubnetAvailabilityZone": {
```

```
                    "Name": "us-east-1f"
                },
                "SubnetStatus": "Active",
                "SubnetIdentifier": "subnet-29ab1025"
            }
        ],
        "VpcId": "vpc-91280df6",
        "DBSubnetGroupName": "sample-subnet-group"
    }
}
```

Deleting an Amazon DocumentDB subnet group

You can use the AWS Management Console or AWS CLI to delete an Amazon DocumentDB subnet group. However, you cannot delete the `default` subnet group.

Using the AWS Management Console

You can use the AWS Management Console to delete a subnet group. But you can't delete the `default` subnet group.

To delete a subnet group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Subnet groups**. Then choose the button to the left of the subnet group's name. Remember that you can't delete the `default` subnet group.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. Choose **Actions**, and then choose **Delete**.
4. In the confirmation dialog box:
   - To delete the subnet group, choose **Delete**.
   - To keep the subnet group, choose **Cancel**.

Using the AWS CLI

To delete an Amazon DocumentDB subnet group using the AWS CLI, use the `delete-db-subnet-group` operation with the following parameter.

Parameter

- `--db-subnet-group-name`—Required. The name of the Amazon DocumentDB subnet group to delete. Remember that you can't delete the `default` subnet group.

Example

The following code deletes `sample-subnet-group`.

For Linux, macOS, or Unix:

```
aws docdb delete-db-subnet-group \
    --db-subnet-group-name sample-subnet-group
```

For Windows:

```
aws docdb delete-db-subnet-group ^
    --db-subnet-group-name sample-subnet-group
```

This operation produces no output.

Amazon DocumentDB high availability and replication

You can achieve high availability and read scaling in Amazon DocumentDB (with MongoDB compatibility) by using replica instances. A single Amazon DocumentDB cluster supports a single primary instance and up to 15 replica instances. These instances can be distributed across Availability Zones within the cluster's Region. The primary instance accepts read and write traffic, and replica instances accept only read requests.

The cluster volume is made up of multiple copies of the data for the cluster. However, the data in the cluster volume is represented as a single, logical volume to the primary instance and to Amazon DocumentDB replicas in the cluster. Replica instances are eventually consistent. They return query results with minimal replica lag—usually much less than 100 milliseconds after the

primary instance has written an update. Replica lag varies depending on the rate of database change. That is, during periods in which a large number of write operations occur for the database, you might see an increase in the replica lag.

Read scaling

Amazon DocumentDB replicas work well for read scaling because they are fully dedicated to read operations on your cluster volume. Write operations are managed by the primary instance. The cluster volume is shared among all instances in your cluster. Therefore, you don't have to replicate and maintain a copy of the data for each Amazon DocumentDB replica.

High availability

When you create an Amazon DocumentDB cluster, depending upon the number of Availability Zones in the subnet group (there must be at least two), Amazon DocumentDB provisions instances across the Availability Zones. When you create instances in the cluster, Amazon DocumentDB automatically distributes the instances across the Availability Zones in a subnet group to balance the cluster. This action also prevents all instances from being located in the same Availability Zone.

Example

To illustrate the point, consider an example where you create a cluster that has a subnet group with three Availability Zones: *AZ1*, *AZ2*, and *AZ3*.

When the first instance in the cluster is created, it is the primary instance and is located in one of the Availability Zones. In this example, it's in *AZ1*. The second instance created is a replica instance and is located in one of the other two Availability Zones, say *AZ2*. The third instance created is a replica instance and is located in the remaining Availability Zone, *AZ3*. If you create more instances, they are distributed across the Availability Zones so that you achieve balance in the cluster.

If a failure occurs in the primary instance (AZ1), a failover is triggered, and one of the existing replicas is promoted to primary. When the old primary recovers, it becomes a replica in the same Availability Zone in which it was provisioned (AZ1). When you provision a three-instance cluster, Amazon DocumentDB continues to preserve that three-instance cluster. Amazon DocumentDB automatically handles detection, failover, and recovery of instance failures without any manual intervention.

When Amazon DocumentDB performs a failover and recovers an instance, the recovered instance remains in the Availability Zone in which it was originally provisioned. However, the role of the

instance might change from primary to replica. Doing this prevents the scenario in which a series of failovers could result in all instances being in the same Availability Zone.

You can specify Amazon DocumentDB replicas as failover targets. That is, if the primary instance fails, the specified Amazon DocumentDB replica or replica from a tier is promoted to the primary instance. There is a brief interruption during which read and write requests made to the primary instance fail with an exception. If your Amazon DocumentDB cluster doesn't include any Amazon DocumentDB replicas, when the primary instance fails, it is re-created. Promoting an Amazon DocumentDB replica is much faster than re-creating the primary instance.

For high availability, create one or more Amazon DocumentDB replicas of the same instance class as the primary instance in different Availability Zones.

For more information, see the following:

- Understanding Amazon DocumentDB cluster fault tolerance
- Amazon DocumentDB failover
  - Controlling the failover target

In Amazon DocumentDB versions 5.0 and 8.0, replica instances don't restart when the primary instance restarts. They continue to serve read requests during primary instance restarts, maintaining improved availability.

High availability with global clusters

For high availability across multiple AWS Regions, you can set up Amazon DocumentDB global clusters. Each global cluster spans multiple Regions, enabling low latency global reads and disaster recovery from outages across Regions. Amazon DocumentDB automatically handles replicating all data and updates from the primary Region to each of the secondary Regions.

Adding replicas

The first instance added to the cluster is the primary instance. Every instance that is added after the first instance is a replica instance. A cluster can have up to 15 replica instances in addition to the primary.

When you create a cluster using the AWS Management Console, a primary instance is automatically created at the same time. To create a replica at the same time as you create the cluster and the primary instance, choose **Create replica in different zone**. For more information, see step 4.d in

Creating an Amazon DocumentDB cluster. To add more replicas to an Amazon DocumentDB cluster, see Adding an Amazon DocumentDB instance to a cluster.

When using the AWS CLI to create your cluster, you must explicitly create your primary and replica instances. For more information, see the "Using the AWS CLI" section in the following topics:

- Creating an Amazon DocumentDB cluster
- Adding an Amazon DocumentDB instance to a cluster

Amazon DocumentDB failover

In certain cases, such as certain types of planned maintenance, or in the unlikely event of a primary node or Availability Zone failure, Amazon DocumentDB (with MongoDB compatibility) detects the failure and replaces the primary node. During a failover, write down time is minimized. This is because the role of primary node fails over to one of the read replicas instead of having to create and provision a new primary node. This failure detection and replica promotion ensure that you can resume writing to the new primary as soon as promotion is complete.

For failover to function, your cluster must have at least two instances — a primary and at least one replica instance.

> **Note**
>
> This topic only applies to original Amazon DocumentDB instance-based clusters. It does not apply to elastic or global clusters.

Controlling the failover target

Amazon DocumentDB provides you with failover tiers as a means to control which replica instance is promoted to primary when a failover occurs.

Failover tiers

Each replica instance is associated with a failover tier (0–15). When a failover occurs due to maintenance or an unlikely hardware failure, the primary instance fails over to a replica with the highest priority (the lowest numbered tier). If multiple replicas have the same priority tier, the primary fails over to that tier's replica that is the closest in size to the previous primary.

By setting the failover tier for a group of select replicas to `0` (the highest priority), you can ensure that a failover will promote one of the replicas in that group. You can effectively prevent specific replicas from being promoted to primary in case of a failover by assigning a low-priority tier (high number) to these replicas. This is useful in cases where specific replicas are receiving heavy use by an application and failing over to one of them would negatively impact a critical application.

We recommend that replicas that are potential failover targets use the same instance class as the primary instance. This avoids write-performance degradation if a failover promotes a replica during OS updates or other maintenance.

You can set the failover tier of an instance when you create it or later by modifying it. Setting an instance failover tier by modifying the instance does not trigger a failover. For more information see the following topics:

- Adding an Amazon DocumentDB instance to a cluster
- Modifying an Amazon DocumentDB instance

When manually initiating a failover, you have two means to control which replica instance is promoted to primary: the failover tiers as previously described, and the `--target-db-instance-identifier` parameter.

`--target-db-instance-identifier`

For testing, you can force a failover event using the `failover-db-cluster` operation. You can use the `--target-db-instance-identifier` parameter to specify which replica to promote to primary. Using the `--target-db-instance-identifier` parameter supersedes the failover priority tier. If you do not specify the `--target-db-instance-identifier` parameter, the primary failover is in accordance with the failover priority tier.

What happens during a failover

Failover is automatically handled by Amazon DocumentDB so that your applications can resume database operations as quickly as possible without administrative intervention.

- If you have an Amazon DocumentDB replica instance in the same or different Availability Zone when failing over: Amazon DocumentDB flips the canonical name record (CNAME) for your instance to point at the healthy replica, which is, in turn, promoted to become the new primary. Failover typically completes within 30 seconds from start to finish.

- If you don't have an Amazon DocumentDB replica instance (for example, a single instance cluster): Amazon DocumentDB will attempt to create a new instance in the same Availability Zone as the original instance. This replacement of the original instance is done on a best-effort basis and may not succeed if, for example, there is an issue that is broadly affecting the Availability Zone.

Your application should retry database connections in the event of a connection loss.

Testing failover

A failover for a cluster promotes one of the Amazon DocumentDB replicas (read-only instances) in the cluster to be the primary instance (the cluster writer).

When the primary instance fails, Amazon DocumentDB automatically fails over to an Amazon DocumentDB replica, if one exists. You can force a failover when you want to simulate a failure of a primary instance for testing. Each instance in a cluster has its own endpoint address. Therefore, you need to clean up and re-establish any existing connections that use those endpoint addresses when the failover is complete.

To force a failover, use the `failover-db-cluster` operation with these parameters.

- `--db-cluster-identifier`—Required. The name of the cluster to fail over.
- `--target-db-instance-identifier`—Optional. The name of the instance to be promoted to the primary instance.

Example

The following operation forces a failover of the `sample-cluster` cluster. It does not specify which instance to make the new primary instance, so Amazon DocumentDB chooses the instance according to failover tier priority.

For Linux, macOS, or Unix:

```
aws docdb failover-db-cluster \
   --db-cluster-identifier sample-cluster
```

For Windows:

```
aws docdb failover-db-cluster ^
```

```
   --db-cluster-identifier sample-cluster
```

The following operation forces a failover of the `sample-cluster` cluster, specifying that `sample-cluster-instance` is to be promoted to the primary role. (Notice `"IsClusterWriter": true` in the output.)

For Linux, macOS, or Unix:

```
aws docdb failover-db-cluster \
   --db-cluster-identifier sample-cluster \
   --target-db-instance-identifier sample-cluster-instance
```

For Windows:

```
aws docdb failover-db-cluster ^
   --db-cluster-identifier sample-cluster ^
   --target-db-instance-identifier sample-cluster-instance
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBCluster": {
        "HostedZoneId": "Z2SUY0A1719RZT",
        "Port": 27017,
        "EngineVersion": "3.6.0",
        "PreferredMaintenanceWindow": "thu:04:05-thu:04:35",
        "BackupRetentionPeriod": 1,
        "ClusterCreateTime": "2018-06-28T18:53:29.455Z",
        "AssociatedRoles": [],
        "DBSubnetGroup": "default",
        "MasterUsername": "master-user",
        "Engine": "docdb",
        "ReadReplicaIdentifiers": [],
        "EarliestRestorableTime": "2018-08-21T00:04:10.546Z",
        "DBClusterIdentifier": "sample-cluster",
        "ReaderEndpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "DBClusterMembers": [
            {
                "DBInstanceIdentifier": "sample-cluster-instance",
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1,
```

```
                "IsClusterWriter": true
            },
            {
                "DBInstanceIdentifier": "sample-cluster-instance-00",
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1,
                "IsClusterWriter": false
            },
            {
                "DBInstanceIdentifier": "sample-cluster-instance-01",
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1,
                "IsClusterWriter": false
            }
        ],
        "AvailabilityZones": [
            "us-east-1b",
            "us-east-1c",
            "us-east-1a"
        ],
        "DBClusterParameterGroup": "default.docdb3.6",
        "Endpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "IAMDatabaseAuthenticationEnabled": false,
        "AllocatedStorage": 1,
        "LatestRestorableTime": "2018-08-22T21:57:33.904Z",
        "PreferredBackupWindow": "00:00-00:30",
        "StorageEncrypted": false,
        "MultiAZ": true,
        "Status": "available",
        "DBClusterArn": "arn:aws:rds:us-east-1:123456789012:cluster:sample-cluster",
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-12345678"
            }
        ],
        "DbClusterResourceId": "cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    }
}
```

Replication lag

Replication lag is typically 50ms or less. The most common reasons for increased replica lag are:

- A high write rate on the primary that causes the read replicas to fall behind the primary.
- Contention on the read replicas between long running queries (e.g., large sequential scans, aggregation queries) and incoming write replication.
- Very large number of concurrent queries on the read replicas.

To minimize replication lag, try these troubleshooting techniques:

- If you have a high write rate or high CPU utilization, scale up the instances in your cluster.
- If there are long running queries on your read replicas, and very frequent updates to the documents being queried, consider altering your long running queries, or running them against the primary/write replica to avoid contention on the read replicas.
- If there is a very large number of concurrent queries or high CPU utilization only on the read replicas, another option is to scale out the number of read replicas to spread out the workload.
- Because replication lag results from high write throughput and long-running queries, troubleshoot replication lag using the `DBClusterReplicaLagMaximum` CloudWatch metric in combination with the slow query logger and `WriteThroughput`/`WriteIOPS` metrics.

Ensure all your replicas are of the same instance type so that a cluster failover doesn't cause a degradation in performance.

If you are choosing between scaling up and scaling out (eg. six smaller instances vs three larger instances), we generally recommend trying to scale up first (larger instances) before scaling out, as you will get a larger buffer cache per DB instance.

Proactively, you should set a replication lag alarm and set its threshold to a value that you feel is the upper bound for how far behind (or “stale”) your data on replica instances can be before it starts affecting the functionality of your application. In general, we would advise that the replication lag threshold be exceeded for several data points before alarming, due to transient workloads.

> **Note**
>
> Set another alarm for replication lags that exceed 10 seconds. If you surpass this threshold for multiple data points, scale up your instances or reduce your write throughput on the primary instance.

Managing Amazon DocumentDB events

Amazon DocumentDB (with MongoDB compatibility) keeps a record of events that relate to your clusters, instances, snapshots, security groups, and cluster parameter groups. This information includes the date and time of the event, the source name and source type of the event, and a message that is associated with the event.

> **Important**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon RDS and Amazon Neptune. Region limits, limits that are governed at the Region level, are shared between Amazon DocumentDB, Amazon RDS, and Amazon Neptune. For more information, see Regional quotas.

Topics

- Viewing Amazon DocumentDB event categories
- Viewing Amazon DocumentDB events

Viewing Amazon DocumentDB event categories

Each Amazon DocumentDB resource type has specific types of events that can be associated with it. You can use the AWS CLI `describe-event-categories` operation to view the mapping between event types and Amazon DocumentDB resource types.

Parameters

- **`--source-type`**—Optional. Use the `--source-type` parameter to see the event categories for a particular source type. The following are permitted values:
  - `db-cluster`
  - `db-instance`
  - `db-parameter-group`
  - `db-security-group`
  - `db-cluster-snapshot`
- **`--filters`**—Optional. To view the event categories for just Amazon DocumentDB, use the filter `--filter Name=engine,Values=docdb`.

Example

The following code lists the event categories associated with clusters.

For Linux, macOS, or Unix:

```
aws docdb describe-event-categories \
    --filter Name=engine,Values=docdb \
    --source-type db-cluster
```

For Windows:

```
aws docdb describe-event-categories ^
    --filter Name=engine,Values=docdb ^
    --source-type db-cluster
```

Output from this operation looks something like the following (JSON format).

```
{
    "EventCategoriesMapList": [
        {
            "EventCategories": [
                "notification",
                "failure",
                "maintenance",
                "failover"
            ],
            "SourceType": "db-cluster"
        }
    ]
}
```

The following code lists the event categories that are associated with each Amazon DocumentDB source type.

```
aws docdb describe-event-categories
```

Output from this operation looks something like the following (JSON format).

```
{
    "EventCategoriesMapList": [
        {
```

```
            "SourceType": "db-instance",
            "EventCategories": [
                "notification",
                "failure",
                "creation",
                "maintenance",
                "deletion",
                "recovery",
                "restoration",
                "configuration change",
                "read replica",
                "backtrack",
                "low storage",
                "backup",
                "availability",
                "failover"
            ]
        },
        {
            "SourceType": "db-security-group",
            "EventCategories": [
                "configuration change",
                "failure"
            ]
        },
        {
            "SourceType": "db-parameter-group",
            "EventCategories": [
                "configuration change"
            ]
        },
        {
            "SourceType": "db-cluster",
            "EventCategories": [
                "notification",
                "failure",
                "maintenance",
                "failover"
            ]
        },
        {
            "SourceType": "db-cluster-snapshot",
            "EventCategories": [
                "backup"
```

```
            ]
        }
    ]
}
```

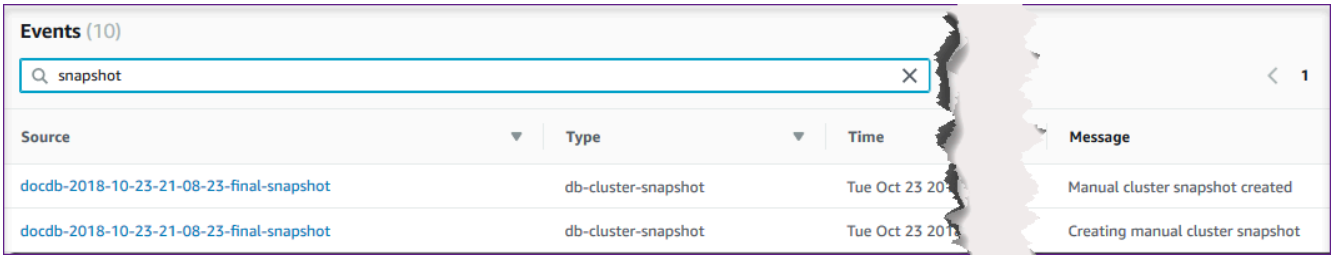
Viewing Amazon DocumentDB events

You can retrieve events for your Amazon DocumentDB resources through the Amazon DocumentDB console, which shows events from the past 24 hours. You can also retrieve events for your Amazon DocumentDB resources by using the describe-events AWS CLI command, or the DescribeEvents Amazon DocumentDB API operation. If you use the AWS CLI or the Amazon DocumentDB API to view events, you can retrieve events for up to the past 14 days.

Using the AWS Management Console

To view all Amazon DocumentDB instance events for the past 24 hours

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Events**. The available events appear in a list.
3. Use the **Filter** list to filter the events by type. Enter a term in the text box to further filter your results. For example, the following screenshot shows filtering all Amazon DocumentDB events for *snapshot* events.



Using the AWS CLI

To view all Amazon DocumentDB instance events for the past 7 days

You can view all Amazon DocumentDB instance events for the past 7 days by running the describe-events AWS CLI operation with the `--duration` parameter set to `10080` (10,080 minutes).

```
aws docdb describe-events --duration 10080
```

Filtering for Amazon DocumentDB Events

To see specific Amazon DocumentDB events, use the `describe-events` operation with the following parameters.

Parameters

- **`--filter`**—Required to limit returned values to Amazon DocumentDB events. Use **`Name=engine,Values=docdb`** to filter all events for Amazon DocumentDB only.
- **`--source-identifier`**—Optional. The identifier of the event source for which events are returned. If omitted, events from all sources are included in the results.
- **`--source-type`**—Optional, unless `--source-identifier` is provided, then required. If `--source-identifier` is provided, `--source-type` must agree with the type of the `--source-identifier`. The following are permitted values:
  - `db-cluster`
  - `db-instance`
  - `db-parameter-group`
  - `db-security-group`
  - `db-cluster-snapshot`

The following example lists all your Amazon DocumentDB events.

```
aws docdb describe-events --filters Name=engine,Values=docdb
```

Output from this operation looks something like the following (JSON format).

```
{
    "Events": [
        {
            "SourceArn": "arn:aws:rds:us-east-1:123SAMPLE012:db:sample-cluster-
instance3",
            "Message": "instance created",
            "SourceType": "db-instance",
            "Date": "2018-12-11T21:17:40.023Z",
            "SourceIdentifier": "sample-cluster-instance3",
            "EventCategories": [
                "creation"
            ]
```

```
        },
        {
            "SourceArn": "arn:aws:rds:us-
east-1:123SAMPLE012:db:docdb-2018-12-11-21-08-23",
            "Message": "instance shutdown",
            "SourceType": "db-instance",
            "Date": "2018-12-11T21:25:01.245Z",
            "SourceIdentifier": "docdb-2018-12-11-21-08-23",
            "EventCategories": [
                "availability"
            ]
        },
        {
            "SourceArn": "arn:aws:rds:us-
east-1:123SAMPLE012:db:docdb-2018-12-11-21-08-23",
            "Message": "instance restarted",
            "SourceType": "db-instance",
            "Date": "2018-12-11T21:25:11.441Z",
            "SourceIdentifier": "docdb-2018-12-11-21-08-23",
            "EventCategories": [
                "availability"
            ]
        }
    ]
}
```

For more information, see Auditing Amazon DocumentDB events.

Managing Amazon DocumentDB cluster parameter groups

You can manage Amazon DocumentDB engine configuration by using parameters in a cluster parameter group. A *cluster parameter group* is a collection of Amazon DocumentDB configuration values that make it easier to manage the parameters of your Amazon DocumentDB clusters. Cluster parameter groups act as a container for engine configuration values that are applied to all instances in the cluster.

This section describes how to create, view, and modify cluster parameter groups. It also shows how you can determine which cluster parameter group is associated with a given cluster.

Topics

- Describing Amazon DocumentDB cluster parameter groups

- Creating Amazon DocumentDB cluster parameter groups
- Modifying Amazon DocumentDB cluster parameter groups
- Modifying Amazon DocumentDB clusters to use custom cluster parameter groups
- Copying Amazon DocumentDB cluster parameter groups
- Resetting Amazon DocumentDB cluster parameter groups
- Deleting Amazon DocumentDB cluster parameter groups
- Amazon DocumentDB cluster parameters reference

Describing Amazon DocumentDB cluster parameter groups

A `default` cluster parameter group is created automatically when you create the first Amazon DocumentDB cluster in a new AWS Region or are using a new engine. Subsequent clusters, which are created in the same Region and have the same engine version, are created with the `default` cluster parameter group.

Topics

- Describing the details of an Amazon DocumentDB cluster parameter group
- Determining an Amazon DocumentDB cluster's parameter group
- Determining clusters and instances associated with a Amazon DocumentDB cluster parameter group

Describing the details of an Amazon DocumentDB cluster parameter group

To describe the details of a given cluster parameter group, complete the following steps using the AWS Management Console or the AWS Command Line Interface (AWS CLI).

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Cluster parameter groups** pane, select the name of the parameter group that you want to see the details of.
4. The resulting page shows the parameter group's parameters, recent activity, and tags.

   - Under **Cluster parameters**, you can see the parameter's name, current value, allowed values, whether the parameter is modifiable, its apply type, data type, and description. You can modify individual parameters by selecting the parameter and then choosing **Edit** in the **Cluster parameters** section. For more information, see Modifying Amazon DocumentDB cluster parameters.
   - Under **Recent events**, you can see the most recent events for this parameter group. You can filter through these events using the search bar in this section. For more information, see Managing Amazon DocumentDB events.
   - Under **Tags**, you can see the tags that are on this cluster parameter group. You can add or remove tags by choosing **Edit** in the **Tags** section. For more information, see Tagging Amazon DocumentDB resources.

Using the AWS CLI

You can use the `describe-db-cluster-parameter-groups` AWS CLI command to view the Amazon Resource Name (ARN), family, description, and name of a single cluster parameter group or all cluster parameter groups that you have for Amazon DocumentDB. You can also use the `describe-db-cluster-parameters` AWS CLI command to view the parameters and their details inside a single cluster parameter group.

- **`--describe-db-cluster-parameter-groups`** — To see a listing of all your cluster parameter groups and their details.
  - **`--db-cluster-parameter-group-name`** — Optional. The name of the cluster parameter group that you want described. If this parameter is omitted, all cluster parameter groups are described.

- **`--describe-db-cluster-parameters`** — To list all the parameters inside a parameter group and their values.
  - **`--db-cluster-parameter-group name`** — Required. The name of the cluster parameter group that you want described.

Example

The following code lists up to 100 cluster parameter groups and their ARN, family, description, and name.

```
aws docdb describe-db-cluster-parameter-groups
```

Output from this operation looks something like the following (JSON format).

```
{
          "DBClusterParameterGroups": [
              {
                  "DBClusterParameterGroupArn": "arn:aws:rds:us-
east-1:012345678912:cluster-pg:default.docdb4.0",
                  "DBParameterGroupFamily": "docdb4.0",
                  "Description": "Default cluster parameter group for docdb4.0",
                  "DBClusterParameterGroupName": "default.docdb4.0"
              },
              {
                  "DBClusterParameterGroupArn": "arn:aws:rds:us-
east-1:012345678912:cluster-pg:sample-parameter-group",
                  "DBParameterGroupFamily": "docdb4.0",
                  "Description": "Custom docdb4.0 parameter group",
                  "DBClusterParameterGroupName": "sample-parameter-group"
              }
          ]
}
```

Example

The following code lists the ARN, family, description, and name for `sample-parameter-group`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameter-groups \
```

```
          --db-cluster-parameter-group-name sample-parameter-group
```

For Windows:

```
aws docdb describe-db-cluster-parameter-groups ^
          --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following (JSON format).

```
{
          "DBClusterParameterGroups": [
              {
                  "DBClusterParameterGroupArn": "arn:aws:rds:us-
east-1:123456789012:cluster-pg:sample-parameter-group",
                  "Description": "Custom docdb4.0 parameter group",
                  "DBParameterGroupFamily": "docdb4.0",
                  "DBClusterParameterGroupName": "sample-parameter-group"
              }
          ]
}
```

Example

The following code lists the values of the parameters in `sample-parameter-group`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
    --db-cluster-parameter-group-name sample-parameter-group
```

For Windows:

```
aws docdb describe-db-cluster-parameters ^
    --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following (JSON format).

```
{
   "Parameters": [
         {
            "ParameterName": "audit_logs",
            "ParameterValue": "disabled",
            "Description": "Enables auditing on cluster.",
            "Source": "system",
            "ApplyType": "dynamic",
            "DataType": "string",
            "AllowedValues": "enabled,disabled",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot"
         },
         {
            "ParameterName": "change_stream_log_retention_duration",
            "ParameterValue": "17777",
            "Description": "Duration of time in seconds that the change stream log
 is retained and can be consumed.",
            "Source": "user",
            "ApplyType": "dynamic",
            "DataType": "integer",
            "AllowedValues": "3600-86400",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot"
         }
   ]
}
```

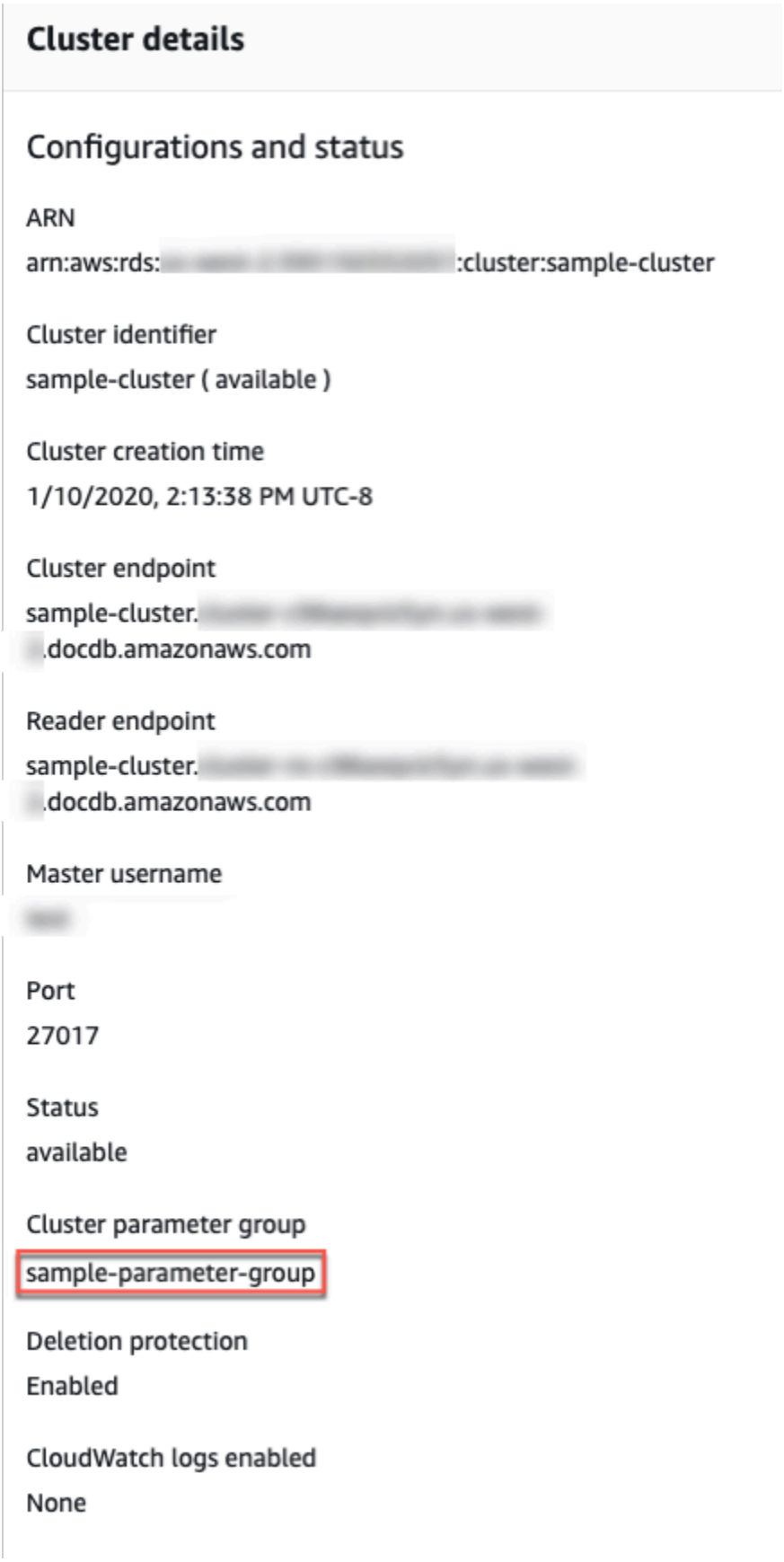
Determining an Amazon DocumentDB cluster's parameter group

To determine which parameter group is associated with a particular cluster, complete the following steps using the AWS Management Console or the AWS CLI.

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the left navigation pane, choose **Clusters**.
3. In the list of clusters, select the name of the cluster you are interested in.

4. The resulting page shows the details of the cluster that you selected. Scroll down to **Cluster details**. At the bottom of that section, locate the parameter group's name below **Cluster parameter group**.


Cluster details
Configurations and status
ARN
arn:aws:rds: :cluster:sample-cluster
Cluster identifier
sample-cluster (available)
Cluster creation time
1/10/2020, 2:13:38 PM UTC-8
Cluster endpoint
sample-cluster.
.docdb.amazonaws.com
Reader endpoint
sample-cluster.
.docdb.amazonaws.com
Master username
Port
27017
Status
available
Cluster parameter group
sample-parameter-group
Deletion protection
Enabled
CloudWatch logs enabled
None

Using the AWS CLI

The following AWS CLI code determines which parameter group is governing the cluster `sample-cluster`.

```
aws docdb describe-db-clusters \
      --db-cluster-identifier sample-cluster \
      --query 'DBClusters[*].[DBClusterIdentifier,DBClusterParameterGroup]'
```

Output from this operation looks something like the following (JSON format).

```
[
       [
           "sample-cluster",
           "sample-parameter-group"
       ]
]
```

Determining clusters and instances associated with a Amazon DocumentDB cluster parameter group

To determine how many clusters and instances are associated with each parameter group, complete the following steps using the AWS CLI.

Using the AWS CLI

The following AWS CLI code determines how many clusters and instances are associated with `sample-parameter-group`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
      --query 'DBClusters[*].
[DBClusterParameterGroup,DBClusterIdentifier,DBClusterMembers[*].DBInstanceIdentifier]'
 \
      --output json | \
      jq -r 'group_by(.[0]) | map({
         parameter_group_name: .[0][0],
         total_clusters: length,
         total_instances: map(.[2] | length) | add // 0
      }) | .[]'
```

For Windows:

```
aws docdb describe-db-clusters ^
      --query 'DBClusters[*].
[DBClusterParameterGroup,DBClusterIdentifier,DBClusterMembers[*].DBInstanceIdentifier]'
 ^
      --output json | ^
      jq -r 'group_by(.[0]) | map({
         parameter_group_name: .[0][0],
         total_clusters: length,
         total_instances: map(.[2] | length) | add // 0
      }) | .[]'
```

Output from this operation looks something like the following (JSON format).

```
{
   "parameter_group_name": "sample-parameter-group",
   "total_clusters": 50,
   "total_instances": 150
}
```

Creating Amazon DocumentDB cluster parameter groups

Default cluster parameter groups such as `default.docdb5.0`, `default.docdb4.0`, or `default.docdb3.6`, are created when you create a cluster with a new engine version and in a new AWS Region. Subsequent clusters created in the same Region and with the same engine version inherit the `default` cluster parameter group. Once created, the `default` parameter groups cannot be deleted or renamed. You can modify the engine behavior of cluster instances by creating a custom parameter group with preferred parameter values and attaching it to your Amazon DocumentDB cluster.

The following procedure guides you through creating a custom cluster parameter group. You can then modify the parameters within that parameter group.

> **Note**
>
> After you create a cluster parameter group, you should wait at least 5 minutes before using that particular parameter group. This allows Amazon DocumentDB to fully complete the `create` action before the cluster parameter group is used for a new cluster. You can use

the AWS Management Console or the `describe-db-cluster-parameter-groups` AWS CLI operation to verify that your cluster parameter group has been created. For more information, see Describing Amazon DocumentDB cluster parameter groups.

Using the AWS Management Console

To create a cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.

   Tip

   If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Cluster parameter groups** pane, choose **Create**.
4. In the **Create cluster parameter group** pane, enter the following:

   a. **New cluster parameter group name** — Enter a name for the cluster parameter group. For example, `sample-parameter-group`. Cluster parameter groups have the following naming constraints:

   - Length is [1–255] alphanumeric characters.
   - First character must be a letter.
   - Cannot end with a hyphen or contain two consecutive hyphens.

   b. **Family** — Choose a DocumentDB version you intend to use for your cluster.

   c. **Description** — Provide a description for this cluster parameter group.

5. To create the cluster parameter group, choose **Create**. To cancel the operation, choose **Cancel**.
6. After you choose **Create**, the following text appears at the top of the page to verify that your cluster parameter group has been successfully created:

```
Successfully created cluster parameter group 'sample-parameter-group'.
```

Using the AWS CLI

To create a new cluster parameter group for Amazon DocumentDB 4.0 clusters, use the AWS CLI `create-db-cluster-parameter-group` operation with the following parameters:

- **`--db-cluster-parameter-group-name`** — The name of the custom cluster parameter group. For example, `sample-parameter-group`.
- **`--db-cluster-parameter-group-family`** — The cluster parameter group family that is used as the template for the custom cluster parameter group.
- **`--description`** — The user-provided description for this cluster parameter group. The following example uses "`Custom docdb4.0 parameter group`".

For Linux, macOS, or Unix:

Example

```
aws docdb create-db-cluster-parameter-group \
 --db-cluster-parameter-group-name sample-parameter-group \
 --db-parameter-group-family docdb5.0 \
 --description "Custom docdb5.0 parameter group"
```

For Windows:

```
aws docdb create-db-cluster-parameter-group ^
 --db-cluster-parameter-group-name sample-parameter-group ^
 --db-parameter-group-family docdb5.0 ^
 --description "Custom docdb5.0 parameter group"
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupName": "sample-parameter-group",
        "DBParameterGroupFamily": "docdb5.0",
        "Description": "Custom docdb4.0 parameter group",
```

```
        "DBClusterParameterGroupArn": "sample-parameter-group-arn"
    }
}
```

Modifying Amazon DocumentDB cluster parameter groups

This section explains how to modify a *custom* Amazon DocumentDB parameter group. In Amazon DocumentDB, you cannot modify a `default` cluster parameter group which is created when you first create a cluster with new engine version in a new Region. If your Amazon DocumentDB cluster is using the default cluster parameter group and you want to modify a value in it, you must first create a new parameter group or copy an existing parameter group, modify it, and then apply the modified parameter group to your cluster.

Complete the following steps to modify a custom cluster parameter group. Modify actions could take a while to propagate. Wait for the modified cluster parameter group to be available before attaching it to your cluster. You can use the AWS Management Console or the AWS CLI `describe-db-cluster-parameters` operation to verify that your cluster parameter group has been modified. For more information, see Describing cluster parameter groups.

Using the AWS Management Console

Follow these steps to modify a custom Amazon DocumentDB parameter group. You can't modify a `default` parameter group. If you want to modify a value in the `default` parameter group, you can copy the default cluster parameter group, modify it, and then apply the modified parameter group to your cluster. For more information about applying parameter groups to your cluster, see Modifying an Amazon DocumentDB cluster.

To modify a custom cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane on the left side of the console, choose **Parameter groups**. In the list of parameter groups, choose the name of the parameter group that you want to modify.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. For each parameter in the parameter group that you want to modify, do the following:

   a. Locate the parameter that you want to modify, and verify that it is modifiable by checking if it is listed as `true` under the **Modifiable** column.

   b. If it is modifiable, select the parameter and choose **Edit** from the top right of the console page.

   c. In the **Modify <parameter-name>** dialog box, make the changes that you want. Then choose **Modify cluster parameter**, or choose **Cancel** to discard the changes.

Using the AWS CLI

You can modify the `ParameterValue`, `Description`, or `ApplyMethod` of any modifiable parameter in a custom Amazon DocumentDB cluster parameter group using the AWS CLI. You can't make modifications directly to a default cluster parameter group.

To modify a custom cluster parameter group's parameters, use the `modify-db-cluster-parameter-group` operation with the following parameters.

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group that you are modifying.
- **`--parameters`** — Required. The parameters that you are modifying. For a list of the parameters that apply to all instances in an Amazon DocumentDB cluster, see the Amazon DocumentDB cluster parameters reference. Each parameter entry must include the following:
  - **`ParameterName`** — The name of the parameter that you are modifying.
  - **`ParameterValue`** — The new value for this parameter.
  - **`ApplyMethod`** — How you want changes to this parameter applied. Permitted values are `immediate` and `pending-reboot`.

> **Note**
>
> Parameters with the `ApplyType` of `static` must have an `ApplyMethod` of `pending-reboot`.

Example- Modifying a parameter's value

In this example, you list the parameter values of `sample-parameter-group` and modify the `tls` parameter. Then, after waiting 5 minutes, you again list the parameter values of `sample-parameter-group` to see the changed parameter values.

1. List the parameters and their values of `sample-parameter-group`.

   For Linux, macOS, or Unix:

   ```
   aws docdb describe-db-cluster-parameters \
          --db-cluster-parameter-group-name sample-parameter-group
   ```

   For Windows:

   ```
   aws docdb describe-db-cluster-parameters ^
          --db-cluster-parameter-group-name sample-parameter-group
   ```

   Output from this operation looks something like the following (JSON format).

   ```
   {
          "Parameters": [
              {
                  "Source": "system",
                  "ApplyType": "static",
                  "AllowedValues": "disabled,enabled",
                  "ParameterValue": "enabled",
                  "ApplyMethod": "pending-reboot",
                  "DataType": "string",
                  "ParameterName": "tls",
                  "IsModifiable": true,
                  "Description": "Config to enable/disable TLS"
              },
              {
   ```

```
                "Source": "user",
                "ApplyType": "dynamic",
                "AllowedValues": "disabled,enabled",
                "ParameterValue": "enabled",
                "ApplyMethod": "pending-reboot",
                "DataType": "string",
                "ParameterName": "ttl_monitor",
                "IsModifiable": true,
                "Description": "Enables TTL Monitoring"
            }
        ]
}
```

2. Modify the `tls` parameter so that its value is `disabled` .

   You can't modify the `ApplyMethod` because the `ApplyType` is `static`.

   For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster-parameter-group \
       --db-cluster-parameter-group-name sample-parameter-group \
       --parameters
 "ParameterName"=tls,"ParameterValue"=disabled,"ApplyMethod"=pending-reboot
```

   For Windows:

```
aws docdb modify-db-cluster-parameter-group ^
       --db-cluster-parameter-group-name sample-parameter-group ^
       --parameters
 "ParameterName"=tls,"ParameterValue"=disabled,"ApplyMethod"=pending-reboot
```

   Output from this operation looks something like the following (JSON format).

```
{
       "DBClusterParameterGroupName": "sample-parameter-group"
   }
```

3. Wait at least 5 minutes.
4. List the parameter values of `sample-parameter-group` to verify that the `tls` parameter was modified.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
       --db-cluster-parameter-group-name sample-parameter-group
```

For Windows:

```
aws docdb describe-db-cluster-parameters ^
       --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following (JSON format).

```
{
       "Parameters": [
           {
               "ParameterValue": "false",
               "ParameterName": "enable_audit_logs",
               "ApplyType": "dynamic",
               "DataType": "string",
               "Description": "Enables auditing on cluster.",
               "AllowedValues": "true,false",
               "Source": "system",
               "IsModifiable": true,
               "ApplyMethod": "pending-reboot"
           },
           {
               "ParameterValue": "disabled",
               "ParameterName": "tls",
               "ApplyType": "static",
               "DataType": "string",
               "Description": "Config to enable/disable TLS",
               "AllowedValues": "disabled,enabled",
               "Source": "system",
               "IsModifiable": true,
               "ApplyMethod": "pending-reboot"
           }
       ]
}
```

Modifying Amazon DocumentDB clusters to use custom cluster parameter groups

When you create an Amazon DocumentDB cluster, a `default.docdb4.0` parameter group is automatically created for that cluster. You can't modify the `default` cluster parameter group. Instead, you can modify your Amazon DocumentDB cluster to associate a new customized parameter group with it.

This section explains how to modify an existing Amazon DocumentDB cluster to use a custom cluster parameter group using the AWS Management Console and the AWS Command Line Interface (AWS CLI).

Using the AWS Management Console

To modify an Amazon DocumentDB cluster to use a new, non-default cluster parameter group

1. Before you begin, make sure you have created an Amazon DocumentDB cluster and a cluster parameter group. See Creating an Amazon DocumentDB cluster and Creating Amazon DocumentDB cluster parameter groups for further instructions.
2. After creating your cluster parameter group, open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb. In the navigation pane, choose **Clusters** to add your new parameter group to a cluster.
3. Choose the cluster that you want to associate your parameter group with. Choose **Actions**, and then choose **Modify** to modify your cluster.
4. Under **Cluster options**, choose the new parameter group that you want to associate your cluster with.
5. Choose **Continue** to view a summary of your modifications.
6. After verifying your changes, you can apply them immediately or during the next maintenance window under **Scheduling of modifications**.
7. Choose **Modify cluster** to update your cluster with your new parameter group.

Using the AWS CLI

Before you begin, make sure that you have created an Amazon DocumentDB cluster and a cluster parameter group. You can create an Amazon DocumentDB cluster using the AWS CLI

`create-db-cluster` operation. You can create a cluster parameter group using the AWS CLI `create-db-cluster-parameter-group` operation.

To add your new cluster parameter group to your cluster, use the AWS CLI `modify-db-cluster` operation with the following parameters.

- **--db-cluster-identifier** — The name of your cluster (for example, `sample-cluster`).
- **--db-cluster-parameter-group-name** — The name of the parameter group that you want to associate your cluster with (for example, `sample-parameter-group`).

Example

```
aws docdb modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following (JSON format).

```
"DBCluster": {
    "AvailabilityZones": [
            "us-west-2c",
            "us-west-2b",
            "us-west-2a"
    ],
    "BackupRetentionPeriod": 1,
    "DBClusterIdentifier": "sample-cluster",
    "DBClusterParameterGroup": "sample-parameter-group",
    "DBSubnetGroup": "default",
   ...
}
```

Copying Amazon DocumentDB cluster parameter groups

You can make a copy of a cluster parameter group in Amazon DocumentDB using the AWS Management Console or the AWS Command Line Interface (AWS CLI).

Using the AWS Management Console

The following procedure guides you through making a new cluster parameter group by making a copy of an existing cluster parameter group.

To copy a cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.
3. In the **Cluster parameter groups** pane, choose the name of the cluster parameter group that you want to copy.
4. Choose **Actions**, and then choose **Copy** to copy that parameter group.
5. Under **Copy options**, enter a name and description for the new cluster parameter group. Then choose **Copy** to save your changes.

Using the AWS CLI

To make a copy of a cluster parameter group, use the `copy-db-cluster-parameter-group` operation with the following parameters.

- **`--source-db-cluster-parameter-group-identifier`** — Required. The name or Amazon Resource Name (ARN) of the cluster parameter group that you want to make a copy of.

  If the source and target cluster parameter groups are in the same AWS Region, the identifier can be either a name or an ARN.

  If the source and target cluster parameter groups are in different AWS Regions, the identifier must be an ARN.
- **`--target-db-cluster-parameter-group-identifier`** — Required. The name or ARN of the cluster parameter group copy.

  Constraints:

  - Cannot be null, empty, or blank.
  - Must contain 1–255 letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
- **`--target-db-cluster-parameter-group-description`** — Required. A user-supplied description for the cluster parameter group copy.

Example

The following code makes a copy of `sample-parameter-group`, naming the copy `sample-parameter-group-copy`.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-parameter-group \
    --source-db-cluster-parameter-group-identifier sample-parameter-group \
    --target-db-cluster-parameter-group-identifier sample-parameter-group-copy \
    --target-db-cluster-parameter-group-description "Copy of sample-parameter-group"
```

For Windows:

```
aws docdb copy-db-cluster-parameter-group ^
    --source-db-cluster-parameter-group-identifier sample-parameter-group ^
    --target-db-cluster-parameter-group-identifier sample-parameter-group-copy ^
    --target-db-cluster-parameter-group-description "Copy of sample-parameter-group"
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBClusterParameterGroup": {
        "DBClusterParameterGroupArn": "arn:aws:rds:us-east-1:123456789012:cluster-
pg:sample-parameter-group-copy",
        "DBClusterParameterGroupName": "sample-parameter-group-copy",
        "DBParameterGroupFamily": "docdb4.0",
        "Description": "Copy of sample-parameter-group"
    }
}
```

Resetting Amazon DocumentDB cluster parameter groups

You can reset some or all of an Amazon DocumentDB cluster parameter group's parameter values to their default values by using the AWS Management Console or the AWS Command Line Interface (AWS CLI) to reset the cluster parameter group.

Using the AWS Management Console

Follow these steps to reset some or all of a cluster parameter group's parameter values to their default values.

To reset a cluster parameter group's parameter values

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane on the left side of the console, choose **Parameter groups**.
3. In the **Cluster parameter groups** pane, choose the name of the cluster parameter group that you want to reset.
4. Choose **Actions**, and then choose **Reset** to reset that parameter group.
5. On the resulting **Cluster parameter group reset confirmation** page, confirm that you want to reset all cluster parameters for that parameter group to their defaults. Then choose **Reset** to reset your parameter group. You can also choose **Cancel** to discard your changes.

Using the AWS CLI

To reset some or all of a cluster parameter group's parameter values to their default values, use the `reset-db-cluster-parameter-group` operation with the following parameters.

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group to reset.
- **`--parameters`** — Optional. A list of `ParameterName` and `ApplyMethod` in the cluster parameter group to reset to their default values. Static parameters must be set to `pending-reboot` to take effect on the next instance restart or `reboot-db-instance` request. You must call `reboot-db-instance` for every instance in your cluster that you want the updated static parameter to apply to.

  This parameter and `--reset-all-parameters` are mutually exclusive: you can use either one but not both.
- **`--reset-all-parameters`** or **`--no-reset-all-parameters`** — Optional. Specifies whether to reset all parameters (`--reset-all-parameters` or only some of the parameters (`--no-reset-all-parameters`) to their default values. The `--reset-all-parameters` parameter and `--parameters` are mutually exclusive: you can use either one but not both.

  When you reset the entire group, dynamic parameters are updated immediately. Static parameters are set to `pending-reboot` to take effect on the next instance restart or `reboot-db-instance` request. You must call `reboot-db-instance` for every instance in your cluster that you want the updated static parameter applied to.

Example

Example 1: Resetting all parameters to their default values

The following code resets all parameters in the cluster parameter group `sample-parameter-group` their default values.

For Linux, macOS, or Unix:

```
aws docdb reset-db-cluster-parameter-group \
       --db-cluster-parameter-group-name sample-parameter-group \
       --reset-all-parameters
```

For Windows:

```
aws docdb reset-db-cluster-parameter-group ^
       --db-cluster-parameter-group-name sample-parameter-group ^
       --reset-all-parameters
```

Example 2: Resetting specified parameters to their default values

The following code resets the `tls` parameter in the cluster parameter group `sample-parameter-group` to its default value.

For Linux, macOS, or Unix:

```
aws docdb reset-db-cluster-parameter-group \
       --db-cluster-parameter-group-name sample-parameter-group \
       --no-reset-all-parameters \
       --parameters ParameterName=tls,ApplyMethod=pending-reboot
```

For Windows:

```
aws docdb reset-db-cluster-parameter-group ^
       --db-cluster-parameter-group-name sample-parameter-group ^
       --no-reset-all-parameters ^
       --parameters ParameterName=tls,ApplyMethod=pending-reboot
```

Output from this operation looks something like the following (JSON format).

```
{
       "DBClusterParameterGroupName": "sample-parameter-group"
```

```
    }
```

Rebooting a cluster instance

Before a static parameter's value is changed, the cluster instance must be rebooted. Reboot each instance in your cluster that you want the updated static parameter to apply to.

For Linux, macOS, or Unix:

```
aws docdb reboot-db-instance \
       --db-instance-identifier sample-cluster-instance
```

For Windows:

```
aws docdb reboot-db-instance ^
       --db-instance-identifier sample-cluster-instance
```

Deleting Amazon DocumentDB cluster parameter groups

You can delete a custom Amazon DocumentDB cluster parameter group using the AWS Management Console or the AWS Command Line Interface (AWS CLI). You can't delete the `default.docdb4.0` cluster parameter group.

Using the AWS Management Console

To delete a cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Parameter groups** pane, choose the radio button to the left of the cluster parameter group that you want to delete.

4. Choose **Actions**, and then choose **Delete**.
5. In the **Delete** confirmation pane, choose **Delete** to delete the cluster parameter group. To keep the cluster parameter group, choose **Cancel**.

Using the AWS CLI

To delete a cluster parameter group, use the `delete-db-cluster-parameter-group` operation with the following parameter.

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group to delete. This must be an existing cluster parameter group. *You cannot delete the `default.docdb4.0` cluster parameter group.*

Example- Deleting a cluster parameter group

The following example walks you through the three steps for deleting a cluster parameter group:

1. Finding the name of the cluster parameter group that you want to delete.
2. Deleting the specified cluster parameter group.
3. Verifying that the cluster parameter group was deleted.

1. Find the name of the cluster parameter group that you want to delete.

The following code lists the names of all cluster parameter groups.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameter-groups \
       --query 'DBClusterParameterGroups[*].[DBClusterParameterGroupName]'
```

For Windows:

```
aws docdb describe-db-cluster-parameter-groups ^
       --query 'DBClusterParameterGroups[*].[DBClusterParameterGroupName]'
```

The output of the preceding operation is a list the names of cluster parameter groups similar to the following (JSON format).

```
[
       [
           "default.docdb4.0"
       ],
       [
           "sample-parameter-group"
       ],
       [
           "sample-parameter-group-copy"
       ]
   ]
```

2. Delete a specific cluster parameter group.

The following code deletes the cluster parameter group `sample-parameter-group-copy`.

For Linux, macOS, or Unix:

```
aws docdb delete-db-cluster-parameter-group \
       --db-cluster-parameter-group-name sample-parameter-group-copy
```

For Windows:

```
aws docdb delete-db-cluster-parameter-group ^
       --db-cluster-parameter-group-name sample-parameter-group-copy
```

There is no output from this operation.

3. Verify that the specified cluster parameter group was deleted.

The following code lists the names of all remaining cluster parameter groups.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameter-groups \
       --query 'DBClusterParameterGroups[*].[DBClusterParameterGroupName]'
```

For Windows:

```
aws docdb describe-db-cluster-parameter-groups ^
```

```
        --query 'DBClusterParameterGroups[*].[DBClusterParameterGroupName]'
```

The output of the preceding operation is a list of cluster parameter groups similar to the following (JSON format). The cluster parameter group that you just deleted should not be in the list.

Output from this operation looks something like the following (JSON format).

```
[
        [
            "default.docdb4.0"
        ],
        [
            "sample-parameter-group"
        ]
   ]
```

Amazon DocumentDB cluster parameters reference

When you change a dynamic parameter and save the cluster parameter group, the change is applied immediately regardless of the *Apply immediately* setting. When you change a static parameter and save the cluster parameter group, the parameter change takes effect after you manually reboot the instance. You can reboot an instance using the Amazon DocumentDB console or by explicitly calling `reboot-db-instance`.

The following table shows the parameters that apply to all instances in an Amazon DocumentDB cluster.

Amazon DocumentDB cluster-level parameters

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
audit_log s	disabled	enabled, disabled, ddl, dml_read, dml_write, all, none	Yes	Dynamic	String	Defines whether Amazon CloudWatc h audit

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						logs are enabled. • **`enabled`** — CloudWatch audit logs are enabled. • **`disabled`** — CloudWatch audit logs are disabled. • **`ddl`** — auditing for DDL events is enabled. • **`dml_read`** — auditing for DML read events is enabled. • **`dml_write`** — auditing for DML write

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Description
						events is enabled. • **all** — auditing for all database events is enabled. • **none** — auditing is disabled.
change_stream_log_retention_duration	10800	3600-604800	Yes	Dynamic	Integer	Defines the duration of time (in seconds) that the change stream log is retained and can be consumed.

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
`default_c ollection _compress ion`	disabled (Amazon DocumentD B 5.0) / zstd (Amazon DocumentD B 8.0)	enabled, disabled (Amazon DocumentD B 5.0) / zstd, lz4, disabled (Amazon DocumentD B 8.0)	Yes	Dynamic	String	Defines the default compressi on setting for new collections in a cluster. • **`enabled`** (Amazon DocumentD B 5.0) — New collectio ns use LZ4 compressi on. • **`zstd`** (Amazon DocumentD B 8.0) — New collectio ns use Zstandard (Zstd) compressi on. This is the default value in Amazon

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Description
						DocumentDB 8.0. • **`lz4`** (Amazon DocumentDB 8.0) — New collections use LZ4 compression. • **`disabled`** — New collections aren't compressed. This is the default value in Amazon DocumentDB 5.0.

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
`profiler`	disabled	enabled, disabled	Yes	Dynamic	String	Enables profiling for slow operations. • **`enabled`** — operation s that take longer than a customer-defined threshold value (e.g., 100ms) are logged toAmazon CloudWatc h Logs. • **`disabled`** — slow operation s are not logged to CloudWatc h Logs.

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Description
profiler_sampling_rate	1.0	0.0-1.0	Yes	Dynamic	Float	Defines the sampling rate for logged operations.
profiler_threshold_ms	100	50-2147483646	Yes	Dynamic	Integer	Defines the threshold for profiler. • All operations greater than profiler_threshold_ms are logged to CloudWatch Logs.

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
`planner_v ersion`	3.0	1.0, 2.0, 3.0	Yes	Dynamic	Float	Defines the query planner version to use for queries. • **`1.0`** — Supported in Amazon DocumentD B 5.0. • **`2.0`** — Improveme nts to find and update operators . Supported in Amazon DocumentD B 5.0. • **`3.0`** — Improveme nts to find, update and aggregati on

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						operators . Default query planner in Amazon DocumentD B 8.0.

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
`tls`	enabled	enabled, disabled, fips-140-3, tls1.2+, tls1.3+	Yes	Static	String	Defines whether Transport Layer Security (TLS) connectio ns are required. • **`enabled`** — TLS connectio ns are required to connect. For Amazon DocumentD B 5.0 and earlier, this enables TLS 1.0 through 1.3, but use TLS 1.2 or higher. Only TLS 1.2 and higher is

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						supported on Amazon DocumentD B 5.0 starting with minor version 5.0.1, Amazon DocumentD B 8.0 and newer. • **`disabled`** — TLS connectio ns cannot be used to connect. • **`fips-140-3`** — TLS connectio ns with Federal Informati on Processin g

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						Standards (FIPS) attribute s are required to connect. The cluster only accepts secure connectio ns per FIPS Publicati on 140-3. This is only supported starting with Amazon DocumentD B 5.0 (engine version 3.0.3727) clusters in these Regions: ca- centra

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						l-1, us-west-2 , us-east-1 , us-east-2, us-gov-east-1, us-gov-west-1. • **`tls1.2+`** — TLS connectio ns using TLS version 1.2 and above are required to connect. This is only supported starting with Amazon DocumentD B 4.0 (engine version 2.0.10980) and

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						Amazon DocumentD B 5.0 (engine version 3.0.11051). • **`tls1.3+`** — TLS connectio ns using TLS version 1.3 and above are required to connect. This is only supported starting with Amazon DocumentD B 4.0 (engine version 2.0.10980) and Amazon DocumentD B 5.0

Parameter	Default Valu	Valid Values	Modifiable	Apply Type	Data Type	Descripti on
						(engine version 3.0.11051).
`ttl_monit or`	enabled	enabled, disabled	Yes	Dynamic	String	Defines whether Time to Live (TTL) monitoring is enabled for the cluster. • **`enabled`** — TTL monitorin g is enabled. • **`disabled`** — TTL monitorin g is disabled.

Modifying Amazon DocumentDB cluster parameters

In Amazon DocumentDB, *cluster parameter groups* consist of *parameters* that apply to all of the instances that you create in the cluster. For custom cluster parameter groups, you can modify a parameter value at any time or reset all the parameter values to their defaults for parameter groups that you create. This section describes how to view the parameters that make up an Amazon DocumentDB cluster parameter group and their values, and how you can change or update these values.

Parameters can be *dynamic* or *static*. When you change a dynamic parameter and save the cluster parameter group, the change is applied immediately regardless of the `Apply Immediately` setting. When you change a static parameter and save the cluster parameter group, the parameter change takes effect only after you manually reboot the instances.

Viewing an Amazon DocumentDB cluster parameter group's parameters

You can see an Amazon DocumentDB cluster's parameters and their values using the AWS Management Console or AWS CLI.

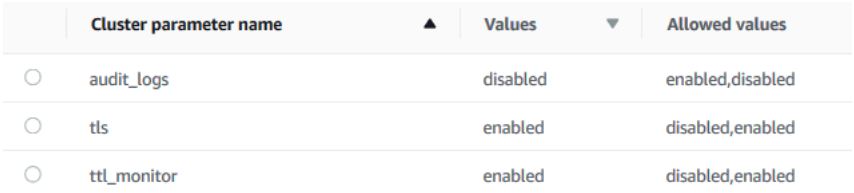
Using the AWS Management Console

To view the details of a cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Parameter groups** pane, choose the name of the cluster parameter group that you want to see the details of.
4. The resulting page shows the following values for each parameter: the parameter's name, current value, allowed values, whether the parameter is modifiable, apply type, data type, and description.



Cluster parameter name	Values	Allowed values
audit_logs	disabled	enabled,disabled
tls	enabled	disabled,enabled
ttl_monitor	enabled	disabled,enabled

Using the AWS CLI

To see a cluster's parameter group's parameters and their values, use the `describe-db-cluster-parameters` operation with the following parameters.

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group for which you want a detailed parameter list.
- **`--source`** — Optional. If supplied, returns only parameters for a specific source. Parameter sources can be `engine-default`, `system`, or `user`.

Example

The following code lists the parameters and their values for the `custom3-6-param-grp` parameter group. For more information about the parameter group, omit the `--query` line. For information about all parameter groups, omit the `--db-cluster-parameter-group-name` line.

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
   --db-cluster-parameter-group-name custom3-6-param-grp \
   --query 'Parameters[*].[ParameterName,ParameterValue]'
```

For Windows:

```
aws docdb describe-db-cluster-parameters ^
   --db-cluster-parameter-group-name custom3-6-param-grp ^
   --query 'Parameters[*].[ParameterName,ParameterValue]'
```

Output from this operation looks something like the following (JSON format).

```
[
    [
        "audit_logs",
        "disabled"
    ],
    [
        "tls",
        "enabled"
    ],
    [
        "ttl_monitor",
        "enabled"
    ]
]
```

Modifying an Amazon DocumentDB cluster parameter group's parameters

You can modify a parameter group's parameters using the AWS Management Console or AWS CLI.

Using the AWS Management Console

To update the parameters of a cluster parameter group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the **Parameter groups** pane, choose the cluster parameter group that you want to update the parameters of.
4. The resulting page shows the parameters and their corresponding details for this cluster parameter group. Select a parameter to update.
5. On the top right of the page, choose **Edit** to change the value of the parameter. For more information about the types of cluster parameters, see Amazon DocumentDB cluster parameters reference.
6. Make your change, and then choose **Modify cluster parameter** to save the changes. To discard your changes, choose **Cancel**.

Using the AWS CLI

To modify a cluster parameter group's parameters, use the `modify-db-cluster-parameter-group` operation with the following parameters:

- **`--db-cluster-parameter-group-name`** — Required. The name of the cluster parameter group that you are modifying.
- **`--parameters`** — Required. The parameter or parameters that you are modifying. Each parameter entry must include the following:
  - **`ParameterName`** — The name of the parameter that you are modifying.

- **`ParameterValue`** — The new value for this parameter.
- **`ApplyMethod`** — How you want changes to this parameter applied. Permitted values are `immediate` and `pending-reboot`.

> **Note**
>
> Parameters with the `ApplyType` of `static` must have an `ApplyMethod` of `pending-reboot`.

To change the values of a cluster parameter group's parameters (AWS CLI)

The following example changes the `tls` parameter.

1. **List the parameters and their values of `sample-parameter-group`**

   For Linux, macOS, or Unix:

   ```
   aws docdb describe-db-cluster-parameters \
       --db-cluster-parameter-group-name sample-parameter-group
   ```

   For Windows:

   ```
   aws docdb describe-db-cluster-parameters ^
       --db-cluster-parameter-group-name sample-parameter-group
   ```

   Output from this operation looks something like the following (JSON format).

   ```
   {
       "Parameters": [
           {
               "Source": "system",
               "ApplyType": "static",
               "AllowedValues": "disabled,enabled",
               "ParameterValue": "enabled",
               "ApplyMethod": "pending-reboot",
               "DataType": "string",
               "ParameterName": "tls",
               "IsModifiable": true,
               "Description": "Config to enable/disable TLS"
   ```

```
        },
        {
            "Source": "user",
            "ApplyType": "dynamic",
            "AllowedValues": "disabled,enabled",
            "ParameterValue": "enabled",
            "ApplyMethod": "pending-reboot",
            "DataType": "string",
            "ParameterName": "ttl_monitor",
            "IsModifiable": true,
            "Description": "Enables TTL Monitoring"
        }
    ]
}
```

2. **Modify the `tls` parameter so that its value is `disabled`**. You can't modify the `ApplyMethod` because the `ApplyType` is `static`.

   For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name sample-parameter-group \
    --parameters
 "ParameterName"=tls,ParameterValue=disabled,ApplyMethod=pending-reboot"
```

   For Windows:

```
aws docdb modify-db-cluster-parameter-group ^
    --db-cluster-parameter-group-name sample-parameter-group ^
    --parameters "ParameterName=tls,ParameterValue=disabled,ApplyMethod=pending-
reboot"
```

   Output from this operation looks something like the following (JSON format).

```
{
    "DBClusterParameterGroupName": "sample-parameter-group"
}
```

3. **Wait at least 5 minutes.**

4. **List the parameter values of `sample-parameter-group`.**

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameters \
    --db-cluster-parameter-group-name sample-parameter-group
```

For Windows:

```
aws docdb describe-db-cluster-parameters ^
    --db-cluster-parameter-group-name sample-parameter-group
```

Output from this operation looks something like the following (JSON format).

```
{
    "Parameters": [
        {
            "ParameterName": "audit_logs",
            "ParameterValue": "disabled",
            "Description": "Enables auditing on cluster.",
            "Source": "system",
            "ApplyType": "dynamic",
            "DataType": "string",
            "AllowedValues": "enabled,disabled",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot"
        },
        {
            "ParameterName": "tls",
            "ParameterValue": "disabled",
            "Description": "Config to enable/disable TLS",
            "Source": "user",
            "ApplyType": "static",
            "DataType": "string",
            "AllowedValues": "disabled,enabled",
            "IsModifiable": true,
            "ApplyMethod": "pending-reboot"
        }
    ]
}
```

Understanding Amazon DocumentDB endpoints

You can use Amazon DocumentDB (with MongoDB compatibility) endpoints to connect to a cluster or instance. Amazon DocumentDB has three different types of endpoints, each with its own purpose.

Topics

- Finding a cluster's endpoints
- Finding an instance's endpoint
- Connecting to endpoints

Cluster endpoint

A cluster endpoint is an endpoint for an Amazon DocumentDB cluster that connects to the current primary instance for the cluster. Each Amazon DocumentDB cluster has a single cluster endpoint and one primary instance. In case of a failover, the cluster endpoint is remapped to the new primary instance.

Reader endpoint

A reader endpoint is an endpoint for an Amazon DocumentDB cluster that connects to one of the available replicas for that cluster. Each Amazon DocumentDB cluster has a reader endpoint. If there is more than one replica, the reader endpoint directs each connection request to one of the Amazon DocumentDB replicas.

Instance endpoint

An instance endpoint is an endpoint that connects to a specific instance. Each instance in a cluster, regardless of whether it is a primary or replica instance, has its own unique instance endpoint. It is best to not use instance endpoints in your application. This is because they can change roles in case of a failover, thus requiring code changes in your application.

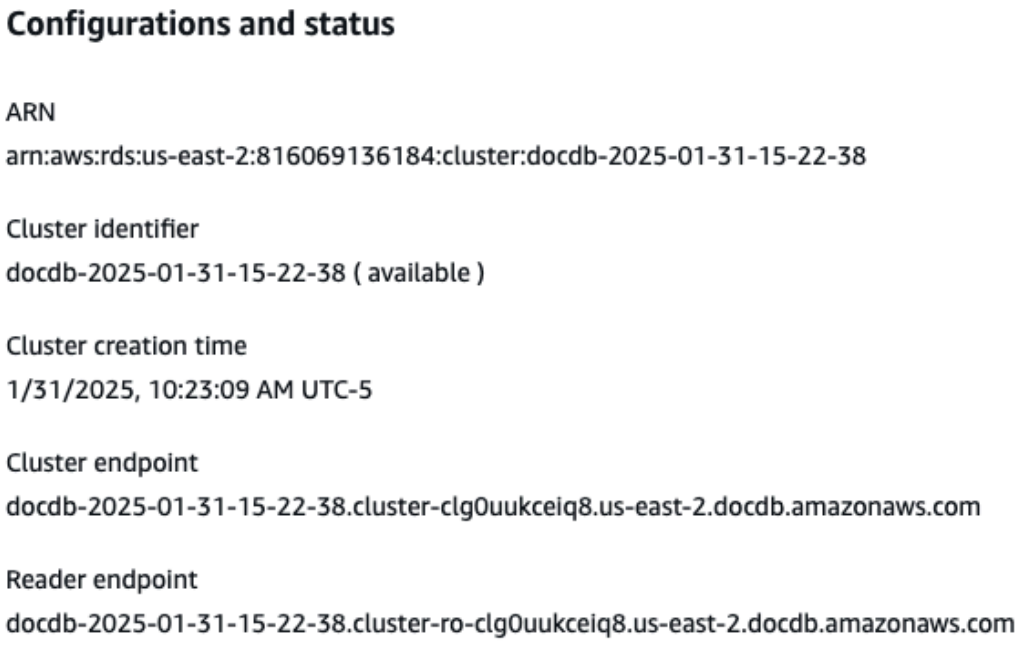
Finding a cluster's endpoints

You can find a cluster's cluster endpoint and reader endpoint using the Amazon DocumentDB console or AWS CLI.

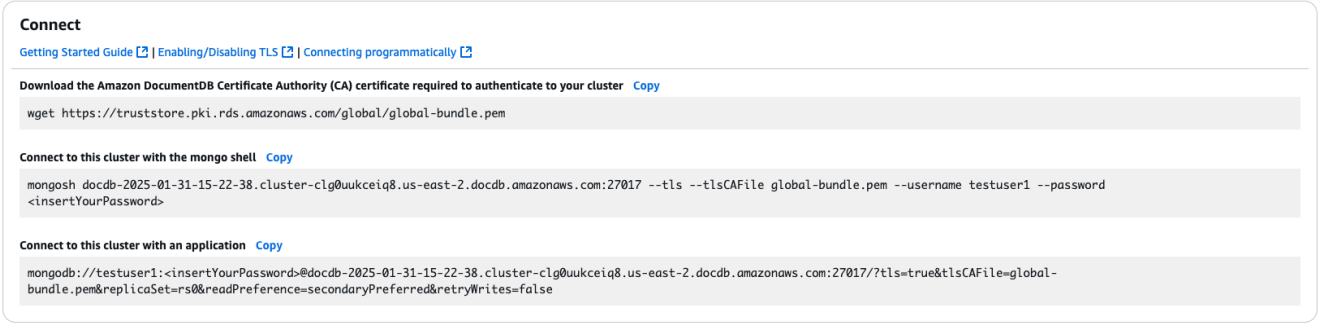
Using the AWS Management Console

To find a cluster's endpoints using the console:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. From the list of clusters, choose the name of the cluster you are interested in.
4. On the cluster details page, select the **Configuration** tab. In the **Configurations and status** section, you will find the **Cluster endpoint** and **Reader endpoint**.



5. To connect to this cluster, select the **Connectivity & security** tab. Locate the connection string for the mongo shell and the connection string that can be used in the application code to connect to your cluster.



Using the AWS CLI

To find the cluster and reader endpoints for your cluster using the AWS CLI, run the `describe-db-clusters` command with these parameters.

Parameters

- **`--db-cluster-identifier`**—Optional. Specifies the cluster to return endpoints for. If omitted, returns endpoints for up to 100 of your clusters.
- **`--query`**—Optional. Specifies the fields to display. Helpful by reducing the amount of data that you need to view to find the endpoints. If omitted, all information about a cluster is returned.
- **`--region`**—Optional. Use the `--region` parameter to specify the Region that you want to apply the command to. If omitted, your default Region is used.

Example

The following example returns the `DBClusterIdentifier`, endpoint (cluster endpoint), and `ReaderEndpoint` for `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
   --region us-east-1 \
   --db-cluster-identifier sample-cluster \
   --query 'DBClusters[*].[DBClusterIdentifier,Port,Endpoint,ReaderEndpoint]'
```

For Windows:

```
aws docdb describe-db-clusters ^
   --region us-east-1 ^
   --db-cluster-identifier sample-cluster ^
   --query 'DBClusters[*].[DBClusterIdentifier,Port,Endpoint,ReaderEndpoint]'
```

Output from this operation looks something like the following (JSON format).

```
[
  [
     "sample-cluster",
     27017,
```

```
    "sample-cluster.cluster-corlsfccjozr.us-east-1.docdb.amazonaws.com",
    "sample-cluster.cluster-ro-corlsfccjozr.us-east-1.docdb.amazonaws.com"
  ]
]
```

Now that you have the cluster endpoint, you can connect to the cluster using either `mongo` or `mongodb`. For more information, see Connecting to endpoints.

Finding an instance's endpoint

You can find the endpoint for an instance using the Amazon DocumentDB console or the AWS CLI.

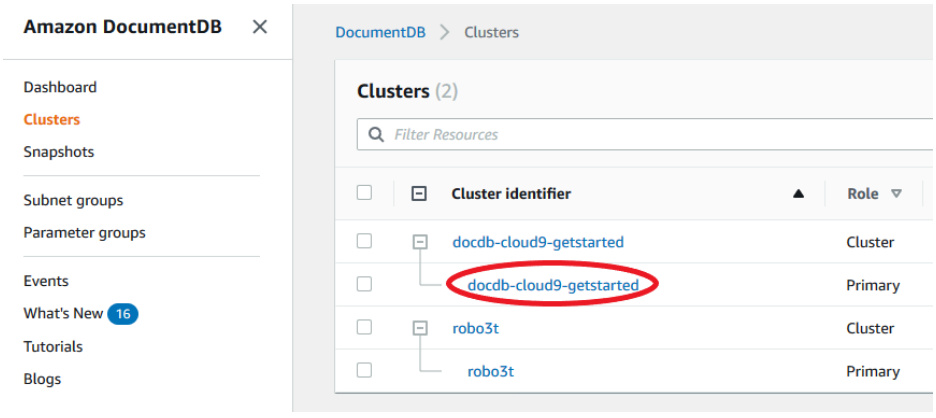
Using the AWS Management Console

To find an instance's endpoint using the console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

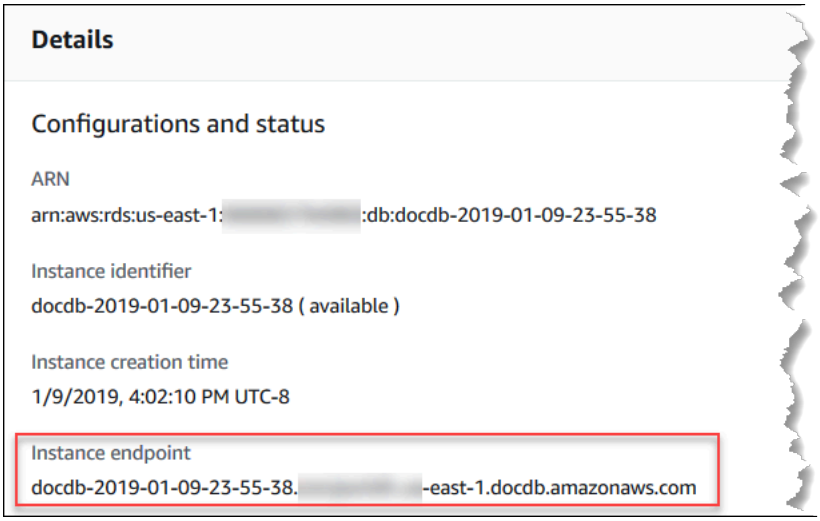
   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. Check the box to the left of the instance you are interested in.

5. Scroll down to the **Details** section then locate the instance endpoint.



6. To connect to this instance, scroll up to the **Connect** section. Locate the connection string for the `mongo` shell and a connection string that can be used in your application code to connect to your instance.



Using the AWS CLI

To find the instance endpoint using the AWS CLI, run the following command with these arguments.

Arguments

- **`--db-instance-identifier`**—Optional. Specifies the instance to return the endpoint for. If omitted, returns the endpoint for up to 100 of your instances.
- **`--query`**—Optional. Specifies the fields to display. Helpful by reducing the amount of data that you need to view to find the endpoints. If omitted, all information on an instance is returned. The `Endpoint` field has three members, so listing it in the query as in the following example returns all three members. If you're only interested in some of the `Endpoint` members, replace `Endpoint` in the query with the members you're interested in, as in the second example.

- **`--region`**—Optional. Use the `--region` parameter to specify the Region that you want to apply the command to. If omitted, your default Region is used.

Example

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --region us-east-1 \
    --db-instance-identifier sample-cluster-instance \
    --query 'DBInstances[*].[DBInstanceIdentifier,Endpoint]'
```

For Windows:

```
aws docdb describe-db-instances ^
    --region us-east-1 ^
    --db-instance-identifier sample-cluster-instance ^
    --query 'DBInstances[*].[DBInstanceIdentifier,Endpoint]'
```

Output from this operation looks something like the following (JSON format).

```
[
    [
        "sample-cluster-instance",
        {
            "Port": 27017,
            "Address": "sample-cluster-instance.corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
            "HostedZoneId": "Z2R2ITUGPM61AM"
        }
    ]
]
```

Reducing the output to eliminate the endpoint's `HostedZoneId`, you can modify your query by specifying `Endpoint.Port` and `Endpoint.Address`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --region us-east-1 \
    --db-instance-identifier sample-cluster-instance \
```

```
    --query 'DBInstances[*].[DBInstanceIdentifier,Endpoint.Port,Endpoint.Address]'
```

For Windows:

```
aws docdb describe-db-instances ^
    --region us-east-1 ^
    --db-instance-identifier sample-cluster-instance ^
    --query 'DBInstances[*].[DBInstanceIdentifier,Endpoint.Port,Endpoint.Address]'
```

Output from this operation looks something like the following (JSON format).

```
[
    [
        "sample-cluster-instance",
        27017,
        "sample-cluster-instance.corcjozrlsfc.us-east-1.docdb.amazonaws.com"
    ]
]
```

Now that you have the instance endpoint, you can connect to the instance using either mongo or mongodb. For more information, see Connecting to endpoints.

Connecting to endpoints

When you have your endpoint, either cluster or instance, you can connect to it using the mongo shell or a connection string.

Connecting using the mongo shell

Use the following structure to construct the string that you need to connect to your cluster or instance using the mongo shell:

```
mongo \
    --ssl \
    --host Endpoint:Port \
    --sslCAFile global-bundle.pem \
    --username UserName \
    --password Password
```

mongo shell examples

Connect to a cluster:

```
mongo \
    --ssl \
    --host sample-cluster.corcjozrlsfc.us-east-1.docdb.amazonaws.com:27017 \
    --sslCAFile global-bundle.pem \
    --username UserName \
    --password Password
```

Connect to an instance:

```
mongo \
    --ssl \
    --host sample-cluster-instance.corcjozrlsfc.us-east-1.docdb.amazonaws.com:27017 \
    --sslCAFile global-bundle.pem \
    --username UserName \
    --password Password
```

Connecting using a connection string

Use the following structure to construct the connection string that you need to connect to your cluster or instance.

```
mongodb://UserName:Password@endpoint:port?replicaSet=rs0&ssl_ca_certs=global-
bundle.pem
```

Connection string examples

Connect to a cluster:

```
mongodb://UserName:Password@sample-cluster.cluster-corlsfccjozr.us-
east-1.docdb.amazonaws.com:27017?replicaSet=rs0&ssl_ca_certs=global-bundle.pem
```

Connect to an instance:

```
mongodb://UserName:Password@sample-cluster-instance.cluster-corlsfccjozr.us-
east-1.docdb.amazonaws.com:27017?replicaSet=rs0&ssl_ca_certs=global-bundle.pem
```

Understanding Amazon DocumentDB Amazon Resource Names (ARNs)

Resources that you create in AWS are each uniquely identified with an Amazon Resource Name (ARN). For certain Amazon DocumentDB (with MongoDB compatibility) operations, you must uniquely identify an Amazon DocumentDB resource by specifying its ARN. For example, when you add a tag to a resource, you must provide the resource's ARN.

Topics

- Constructing an ARN for an Amazon DocumentDB resource
- Finding an Amazon DocumentDB resource ARN

Constructing an ARN for an Amazon DocumentDB resource

You can construct an ARN for an Amazon DocumentDB resource using the following syntax. Amazon DocumentDB shares the format of Amazon Relational Database Service (Amazon RDS) ARNs. Amazon DocumentDB ARNs contain `rds` and not `docdb`.

`arn:aws:rds:`*`region`*`:`*`account_number`*`:`*`resource_type`*`:`*`resource_id`*

Region Name	Region	Availability Zones (compute)
US East (Ohio)	`us-east-2`	3
US East (N. Virginia)	`us-east-1`	6
US West (Oregon)	`us-west-2`	4
Africa (Cape Town)	`af-south-1`	3
South America (São Paulo)	`sa-east-1`	3
Asia Pacific (Hong Kong)	`ap-east-1`	3
Asia Pacific (Taipei)	`ap-east-2`	3

Region Name	Region	Availability Zones (compute)
Asia Pacific (Hyderabad)	ap-south-2	3
Asia Pacific (Malaysia)	ap-southeast-5	3
Asia Pacific (Mumbai)	ap-south-1	3
Asia Pacific (Osaka)	ap-northeast-3	3
Asia Pacific (Seoul)	ap-northeast-2	4
Asia Pacific (Singapore)	ap-southeast-1	3
Asia Pacific (Sydney)	ap-southeast-2	3
Asia Pacific (Jakarta)	ap-southeast-3	3
Asia Pacific (Melbourne)	ap-southeast-4	3
Asia Pacific (New Zealand)	ap-southeast-6	3
Asia Pacific (Thailand)	ap-southeast-7	3
Asia Pacific (Tokyo)	ap-northeast-1	3
Canada (Central)	ca-central-1	3
Canada West (Calgary)	ca-west-1	3
China (Beijing) Region	cn-north-1	3
China (Ningxia)	cn-northwest-1	3
Europe (Frankfurt)	eu-central-1	3

Region Name	Region	Availability Zones (compute)
Europe (Zurich)	`eu-central-2`	3
Europe (Ireland)	`eu-west-1`	3
Europe (London)	`eu-west-2`	3
Europe (Milan)	`eu-south-1`	3
Europe (Paris)	`eu-west-3`	3
Europe (Spain)	`eu-south-2`	3
Europe (Stockholm)	`eu-north-1`	3
Mexico (Central)	`mx-central-1`	3
Middle East (UAE)	`me-central-1`	3
Israel (Tel Aviv)	`il-central-1`	3
AWS GovCloud (US-West)	`us-gov-west-1`	3
AWS GovCloud (US-East)	`us-gov-east-1`	3

Note

The Amazon DocumentDB architecture separates storage and compute. For the storage layer, Amazon DocumentDB replicates six copies of your data across three AWS Availability Zones (AZs). The AZs listed in the table above are the number of AZs that you can use in a given Region to provision compute instances. As an example, if you are launching an Amazon DocumentDB cluster in ap-northeast-1, your storage will be replicated six ways across three AZs but your compute instances will only be available in two AZs.

The following table shows the format that you should use when constructing an ARN for a particular Amazon DocumentDB resource. Amazon DocumentDB shares the format of Amazon RDS ARNs. Amazon DocumentDB ARNs contain `rds` and not `docdb`.

Resource Type	ARN Format / Example
Instance (db)	`arn:aws:rds:` *`region`*`:`*`account_number`* `:db:`*`resource_id`* `arn:aws:rds:us-east-1:` *`123456789012`* `:db:`*`sample-db-instance`*
Cluster (`cluster`)	`arn:aws:rds:` *`region`*`:`*`account_number`* `:cluster:` *`resource_id`* `arn:aws:rds:us-east-1:` *`123456789012`* `:cluster:` *`sample-db-cluster`*
Cluster parameter group (`cluster-pg`)	`arn:aws:rds:` *`region`*`:`*`account_number`* `:cluster-pg:` *`resource_id`* `arn:aws:rds:us-east-1:` *`123456789012`* `:cluster-pg:` *`sample-db-cluster-parameter-group`*
Security group (`secgrp`)	`arn:aws:rds:` *`region`*`:`*`account_number`* `:secgrp:`*`resource_id`* `arn:aws:rds:us-east-1:` *`123456789012`* `:secgrp:`*`sample-public-secgrp`*
Cluster snapshot (`cluster-snapshot`)	`arn:aws:rds:` *`region`*`:`*`account_number`* `:cluster-snapshot:` *`resource_id`* `arn:aws:rds:us-east-1:` *`123456789012`* `:cluster-snapshot:` *`sample-db-cluster-snapshot`*

Resource Type	ARN Format / Example
Subnet group (subgrp)	arn:aws:rds: *region*:*account_number* :subgrp:*resource_id* arn:aws:rds:us-east-1: *123456789012* :subgrp:*sample-subnet-10*

Note

Each Amazon DocumentDB cluster also has a cluster resource ID, which is the value of the `DbClusterResourceId` field (for example, `cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ`). Unlike the cluster identifier, the cluster resource ID doesn't change if you rename the cluster. The cluster ARN that Amazon DocumentDB API operations return, such as `DBClusterArn` in the output of `describe-db-clusters`, uses the cluster identifier. Use that ARN when you call Amazon DocumentDB API operations.
In billing data, such as AWS Cost Explorer and the AWS Cost and Usage Report, charges for a cluster can appear under an ARN that uses the cluster resource ID in lowercase, for example `arn:aws:rds:us-east-1:123456789012:cluster:cluster-abcdefghijklmnopqrstuvwxyz`. To find the cluster that the ARN refers to, list the identifier and resource ID of each of your clusters, and then match the resource ID without regard to case:

```
aws docdb describe-db-clusters \
    --filter Name=engine,Values=docdb \
    --query "DBClusters[*].[DBClusterIdentifier,DbClusterResourceId]" \
    --output table
```

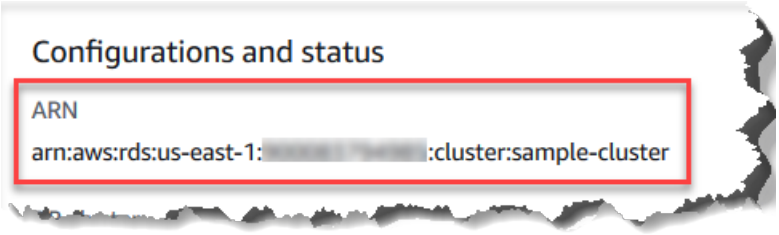
Finding an Amazon DocumentDB resource ARN

You can find the ARN of an Amazon DocumentDB resource using the AWS Management Console or the AWS CLI.

Using the AWS Management Console

To find an ARN using the console, navigate to the resource that you want an ARN for, and view the details for that resource.

For example, you can get the ARN for a cluster by selecting the **Configuration** tab on the cluster details page. The ARN can be found in the **Configurations and status** section, as shown in the following screenshot.



Using the AWS CLI

To get an ARN using the AWS CLI for a particular Amazon DocumentDB resource, use the `describe` operation for that resource. The following table shows each AWS CLI operation and the ARN property that is used with the operation to get an ARN.

AWS CLI Command	ARN Property
`describe-db-instances`	`DBInstanceArn`
`describe-db-clusters`	`DBClusterArn`
`describe-db-parameter-groups`	`DBParameterGroupArn`
`describe-db-cluster-parameter-groups`	`DBClusterParameterGroupArn`
`describe-db-security-groups`	`DBSecurityGroupArn`
`describe-db-snapshots`	`DBSnapshotArn`
`describe-db-cluster-snapshots`	`DBClusterSnapshotArn`
`describe-db-subnet-groups`	`DBSubnetGroupArn`

Example- Finding the ARN for a cluster

The following AWS CLI operation finds the ARN for the cluster `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].DBClusterArn'
```

For Windows:

```
aws docdb describe-db-clusters ^
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].DBClusterArn'
```

Output from this operation looks something like the following (JSON format).

```
[
    "arn:aws:rds:us-east-1:123456789012:cluster:sample-cluster"
]
```

Example- Finding ARNs for multiple parameter groups

For Linux, macOS, or Unix:

```
aws docdb describe-db-cluster-parameter-groups \
   --query 'DBClusterParameterGroups[*].DBClusterParameterGroupArn'
```

For Windows:

```
aws docdb describe-db-cluster-parameter-groups ^
   --query 'DBClusterParameterGroups[*].DBClusterParameterGroupArn'
```

Output from this operation looks something like the following (JSON format).

```
[
    "arn:aws:rds:us-east-1:123456789012:cluster-pg:custom3-6-param-grp",
    "arn:aws:rds:us-east-1:123456789012:cluster-pg:default.aurora5.6",
    "arn:aws:rds:us-east-1:123456789012:cluster-pg:default.docdb3.6"
]
```

Tagging Amazon DocumentDB resources

You can use Amazon DocumentDB (with MongoDB compatibility) tags to add metadata to your Amazon DocumentDB resources. These tags can be used with AWS Identity and Access Management (IAM) policies to manage access to Amazon DocumentDB resources and to control what actions can be applied to the resources. You can also use tags to track costs by grouping expenses for similarly tagged resources.

You can tag the following Amazon DocumentDB resources:

- Clusters
- Instances
- Snapshots
- Cluster snapshots
- Parameter groups
- Cluster parameter groups
- Security groups
- Subnet groups

Overview of Amazon DocumentDB resource tags

An Amazon DocumentDB tag is a name-value pair that you define and associate with an Amazon DocumentDB resource. The name is referred to as the *key*. Supplying a value for the key is optional. You can use tags to assign arbitrary information to an Amazon DocumentDB resource. You can use a tag key, for example, to define a category, and the tag value might be an item in that category. For example, you might define a tag key of `project` and a tag value of `Salix`, indicating that the Amazon DocumentDB resource is assigned to the Salix project. You can also use tags to designate Amazon DocumentDB resources as being used for test or production by using a key such as `environment=test` or `environment=production`. We recommend that you use a consistent set of tag keys to make it easier to track metadata that is associated with Amazon DocumentDB resources.

You can use tags to organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill with tag key values included. Then, to see the cost of combined resources, organize your billing information according to resources with the same tag key values.

For example, you can tag several resources with a specific application name, and then organize your billing information to see the total cost of that application across several services. For more information, see Using Cost Allocation Tags in the *AWS Billing and Cost Management User Guide*.

Each Amazon DocumentDB resource has a tag set, which contains all the tags that are assigned to that resource. A tag set can contain as many as 10 tags, or it can be empty. If you add a tag to an Amazon DocumentDB resource that has the same key as an existing tag on resource, the new value overwrites the old value.

AWS does not apply any semantic meaning to your tags; tags are interpreted strictly as character strings. Amazon DocumentDB can set tags on an instance or other Amazon DocumentDB resources, depending on the settings that you use when you create the resource. For example, Amazon DocumentDB might add a tag indicating that an instance is for production or for testing.

You can add a tag to a snapshot, but your bill will not reflect this grouping.

You can use the AWS Management Console or the AWS CLI to add, list, and delete tags on Amazon DocumentDB resources. When using the AWS CLI, you must provide the Amazon Resource Name (ARN) for the resource that you want to work with. For more information about Amazon DocumentDB ARNs, see Understanding Amazon DocumentDB Amazon Resource Names (ARNs).

Tag constraints

The following constraints apply to Amazon DocumentDB tags:

- Maximum number of tags per resource - 10
- Maximum **Key** length - 128 Unicode characters
- Maximum **Value** length - 256 Unicode characters
- Valid characters for **Key** and **Value** - uppercase and lowercase letters in the UTF-8 character set, digits, space, and the following characters: `_ . : / = + -` and `@` (Java regex: `"^([\\p{L}\\p{Z}\\p{N}_.:/=+\\-]*)$"`)
- Tag keys and values are case sensitive.
- The prefix `aws:` cannot be used for tag keys or values; it is reserved for AWS.

Adding and updating tags on an Amazon DocumentDB resource

You can add up to 10 tags to a resource using the AWS Management Console or the AWS CLI.

Using the AWS Management Console

The process for adding a tag to a resource is similar regardless of what resource you're adding the tag to. In this example, you add a tag to a cluster.

To add or update tags to a cluster using the console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. From the navigation pane, choose **clusters**.
3. Choose the name of the cluster that you want to add tags to.
4. Scroll down to the **Tags** section, and then choose **Edit**.
5. For each tag you that want to add to this resource, do the following:
   a. To add a new tag, enter in the name of the tag in the **Key** box. To change a tag's value, find the tag's name in the **Key** column.
   b. To give the tag a new or updated value, enter a value for the tag in the **Value** box.
   c. If you have more tags to add, choose **Add**. Otherwise, when finished, choose **Save**.

Using the AWS CLI

The process for adding a tag to a resource is similar regardless of what resource you're adding the tags to. In this example, you add three tags to a cluster. The second tag, `key2`, has no value.

Use the AWS CLI operation `add-tags-to-resource` with these parameters.

Parameters

- **`--resource-name`**—The ARN of the Amazon DocumentDB resource that you want to add tags to.
- **`--tags`**—A list the tags (key-value pair) that you want to add to this resource in the format `Key=`*`key-name`*`,Value=`*`tag-value`*.

Example

For Linux, macOS, or Unix:

```
aws docdb add-tags-to-resource \
```

```
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster \
    --tags Key=key1,Value=value1 Key=key2 Key=key3,Value=value3
```

For Windows:

```
aws docdb add-tags-to-resource ^
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster \
    --tags Key=key1,Value=value1 Key=key2 Key=key3,Value=value3
```

The `add-tags-to-resource` operation produces no output. To see the results of the operation, use the `list-tags-for-resource` operation.

Listing tags on an Amazon DocumentDB resource

You can use the AWS Management Console or the AWS CLI to get a listing of the tags for an Amazon DocumentDB resource.

Using the AWS Management Console

The process for listing tags on a resource is similar regardless of what resource you're adding the tag to. In this example, you list the tags for a cluster.

To list the tags on a cluster using the console

1. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. From the navigation pane, choose **clusters**.
3. Choose the name of the cluster that you want to list tags for.
4. To see a listing of the tags on this resource, scroll down to the **Tags** section.

Using the AWS CLI

The process for listing the tags on a resource is similar regardless of what resource you're listing the tag for. In this example, you list the tags on a cluster.

Use the AWS CLI operation `list-tags-for-resource` with these parameters.

Parameters

- **`--resource-name`**—Required. The ARN of the Amazon DocumentDB resource that you want to list tags for.

Example

For Linux, macOS, or Unix:

```
aws docdb list-tags-for-resource \
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster
```

For Windows:

```
aws docdb list-tags-for-resource ^
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster
```

Output from this operation looks something like the following (JSON format).

```
{
    "TagList": [
        {
            "Key": "key1",
            "Value": "value1"
        },
        {
            "Key": "key2",
            "Value": ""
        },
        {
            "Key": "key3",
            "Value": "value3"
        }
    ]
}
```

Removing tags from an Amazon DocumentDB resource

You can use the AWS Management Console or the AWS CLI to remove tags from Amazon DocumentDB resources.

Using the AWS Management Console

The process for removing tags from a resource is similar regardless of what resource you're adding the tag to. In this example, you remove tags from a cluster.

To remove tags from a cluster using the console

1. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. From the navigation pane, choose **clusters**.
3. Choose the name of the cluster that you want to remove tags from.
4. Scroll down to the **Tags** section, and then choose **Edit**.
5. If you want to remove all tags from this resource, choose **Remove all**. Otherwise, for each tag that you want to remove from this resource, do the following:
   a. Locate the name of the tag in the **Key** column.
   b. Choose **Remove** on the same row as the tag key.
   c. When finished, choose **Save**.

Using the AWS CLI

The process for removing a tag from a resource is similar regardless of what resource you're removing the tag from. In this example, you remove a tag from a cluster.

Use the AWS CLI operation `remove-tags-from-resource` with these parameters.

- **`--resource-name`**—Required. The ARN of the Amazon DocumentDB resource that you want to remove tags from.
- **`--tag-keys`**—Required. A list the tag keys that you want removed from this resource.

Example

For Linux, macOS, or Unix:

```
aws docdb remove-tags-from-resource \
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster \
    --tag-keys key1 key3
```

For Windows:

```
aws docdb remove-tags-from-resource ^
    --resource-name arn:aws:rds:us-east-1:1234567890:cluster:sample-cluster \
    --tag-keys key1 key3
```

The `removed-tags-from-resource` operation produces no output. To see the results of the operation, use the `list-tags-for-resource` operation.

Maintaining Amazon DocumentDB

Amazon DocumentDB periodically performs two kinds of maintenance:

- **Cluster maintenance** updates the database engine. Engine updates carry security fixes, bug fixes, new features, and other engine enhancements.
- **Instance maintenance** updates the operating system (OS) on the instance.

Engine patches and OS updates use the same three lifecycle categories—*optional*, *required*, and *forced*—with the same notification and apply behavior for each category. Engine releases also have a fourth category: *minor versions*, which you upgrade to manually. The categories are:

- **Optional**—contains non-critical improvements. No auto-apply date and no AHD notification; apply when it suits you. (For OS updates, you can subscribe to `RDS-EVENT-0230` to be notified when one becomes available.)
- **Required**—contains security and other critical fixes. You receive a notification through the Health Dashboard (AHD) and e-mail. A required action auto-applies during your cluster or instance maintenance window after its `AutoAppliedAfterDate`. You can defer by changing the maintenance window before that date.
- **Forced**—a rare, highly critical fix. Auto-applies outside your maintenance window after its `ForcedApplyDate`. Amazon DocumentDB only designates an action forced when no other option is available.
- **Minor version** (engine releases only)—a numbered engine release on top of a major version (for example, `5.0.1`). User-driven: you upgrade by modifying the cluster's engine version. Never auto-applies; no AHD notification. Minor versions are not published for major versions earlier than 5.0.

Engine patches are released into a single category (optional, required, or forced) and stay there. OS updates progress: most start as optional and, if not applied, transition to required and eventually forced. The exact timing depends on the patch and is published in the AHD notification and the date fields returned by `describe-pending-maintenance-actions` (see Apply dates). The Amazon DocumentDB Release notes use these category names when announcing engine changes.

Applying any engine patch takes the cluster offline briefly. The rest of this topic walks through how maintenance windows work, how to find pending work, how to apply engine patches and minor versions, how OS updates work, and special handling for global clusters.

Topics

- Maintenance actions for Amazon DocumentDB
- Engine version numbering
- Managing your Amazon DocumentDB maintenance windows
- Notifications for Amazon DocumentDB engine patches
- Viewing pending Amazon DocumentDB maintenance actions
- Amazon DocumentDB engine updates
- Minor version upgrades
- Amazon DocumentDB operating system updates
- User-initiated updates
- Global clusters patching

Maintenance actions for Amazon DocumentDB

The following maintenance actions apply to Amazon DocumentDB clusters:

- `system-update` — Upgrade the engine patch for the Amazon DocumentDB cluster. For more information, see Amazon DocumentDB engine updates.
- `os-upgrade` — Update the operating systems of all the DB instances in the Amazon DocumentDB cluster, using rolling upgrades. For more information, see Amazon DocumentDB operating system updates.
- `serverless-platform-version-update` — Update the platform version of all the serverless instances in the Amazon DocumentDB cluster, using rolling upgrades. For more information, see Checking and upgrading the platform version of a serverless cluster.

The following maintenance actions apply to Amazon DocumentDB instances:

- `system-update` — Upgrade the operating system of the Amazon DocumentDB instance. We recommend that you use the cluster-level `os-upgrade` maintenance action instead. For more information, see Amazon DocumentDB operating system updates.

Engine version numbering

Amazon DocumentDB uses two separate version identifiers:

- **Engine version**—a three-part number in the form *`major`*`.`*`major`*`.`*`minor`* (for example, `5.0.0` or `5.0.1`). The first two parts (`5.0`) are the MongoDB compatibility version; the third part is the minor version, incremented when Amazon DocumentDB publishes a minor release containing bug fixes and non-breaking improvements. This is the version you specify when creating or upgrading a cluster.
- **Engine patch version**—a separate three-part number in the form *`major`*`.0.`*`patch`* (for example, `3.0.17983`) that identifies the patch level applied to your cluster. The middle digit is always `0`. Patch versions contain critical security and stability fixes.

When the engine patch version appears on its own, you can determine the engine version from its prefix, as shown in the following table.

Engine patch version prefix	Amazon DocumentDB engine version
`1.0.`*`x`*	3.6
`2.0.`*`x`*	4.0
`3.0.`*`x`*	5.0
`4.0.`*`x`*	8.0

To check the patch version your cluster is running, connect and run `db.runCommand({getEngineVersion: 1})`.

> **Note**
>
> Starting with Amazon DocumentDB 5.0.2 and 8.0.2, `db.runCommand({getEngineVersion: 1})` returns the engine version and the engine patch version as a single string in the form *`engine-version`*+*`engine-patch-version`*. For example, a cluster on Amazon DocumentDB 5.0.2 returns `5.0.2+3.0.12345`, and a cluster on 8.0.2 returns `8.0.2+4.0.12345`. Clusters on earlier versions return the engine patch version on its own, such as `3.0.12345`.
> If you have scripts or monitoring that parse the output of `getEngineVersion`, update them to handle the combined format.

For the list of released engine patch versions and what each one contains, see Release notes.

Managing your Amazon DocumentDB maintenance windows

Every cluster and every instance has its own weekly 30-minute maintenance window—the period when scheduled modifications and software patching run. Most events complete within 30 minutes; larger ones can run longer.

If you don't choose a window when creating the resource, Amazon DocumentDB assigns one at random within an 8-hour daily block defined for the Region, on a randomly selected day. Choose windows that minimize impact to your application—evenings or weekends, for example.

For database engine upgrades, Amazon DocumentDB uses the cluster's window, not the windows of individual instances.

The following table shows the default time blocks per Region.

Region Name	Region	UTC Time Block
US East (Ohio)	us-east-2	03:00-11:00
US East (N. Virginia)	us-east-1	03:00-11:00
US West (Oregon)	us-west-2	06:00-14:00
Africa (Cape Town)	af-south-1	03:00–11:00
Asia Pacific (Hong Kong)	ap-east-1	06:00-14:00

Region Name	Region	UTC Time Block
Asia Pacific (Taipei)	ap-east-2	09:00-17:00
Asia Pacific (Hyderabad)	ap-south-2	06:30–14:30
Asia Pacific (Malaysia)	ap-southeast-5	13:00-21:00
Asia Pacific (Mumbai)	ap-south-1	06:00-14:00
Asia Pacific (Osaka)	ap-northeast-3	12:00-20:00
Asia Pacific (Seoul)	ap-northeast-2	13:00-21:00
Asia Pacific (Singapore)	ap-southeast-1	14:00-22:00
Asia Pacific (Sydney)	ap-southeast-2	12:00-20:00
Asia Pacific (Jakarta)	ap-southeast-3	08:00-16:00
Asia Pacific (Melbourne)	ap-southeast-4	11:00-19:00
Asia Pacific (New Zealand)	ap-southeast-6	13:00-21:00
Asia Pacific (Thailand)	ap-southeast-7	15:00-23:00
Asia Pacific (Tokyo)	ap-northeast-1	13:00-21:00
Canada (Central)	ca-central-1	03:00-11:00
Canada West (Calgary)	ca-west-1	18:00-02:00
China (Beijing)	cn-north-1	06:00-14:00
China (Ningxia)	cn-northwest-1	06:00-14:00
Europe (Frankfurt)	eu-central-1	21:00-05:00
Europe (Zurich)	eu-central-2	02:00-10:00
Europe (Ireland)	eu-west-1	22:00-06:00

Region Name	Region	UTC Time Block
Europe (London)	eu-west-2	22:00-06:00
Europe (Milan)	eu-south-1	02:00-10:00
Europe (Paris)	eu-west-3	23:59-07:29
Europe (Spain)	eu-south-2	02:00–10:00
Europe (Stockholm)	eu-north-1	04:00–12:00
Mexico (Central)	mx-central-1	03:00-11:00
Middle East (UAE)	me-central-1	05:00–13:00
South America (São Paulo)	sa-east-1	00:00-08:00
Israel (Tel Aviv)	il-central-1	04:00-12:00
AWS GovCloud (US-East)	us-gov-east-1	17:00-01:00
AWS GovCloud (US-West)	us-gov-west-1	06:00-14:00

Changing your Amazon DocumentDB maintenance windows

Choose the lowest-traffic window you can, and adjust over time as your traffic patterns shift. The cluster or instance is unavailable during the window only if a system change—a scale-storage operation or an instance class change, for example—requires an outage, and only for as long as that change actually needs.

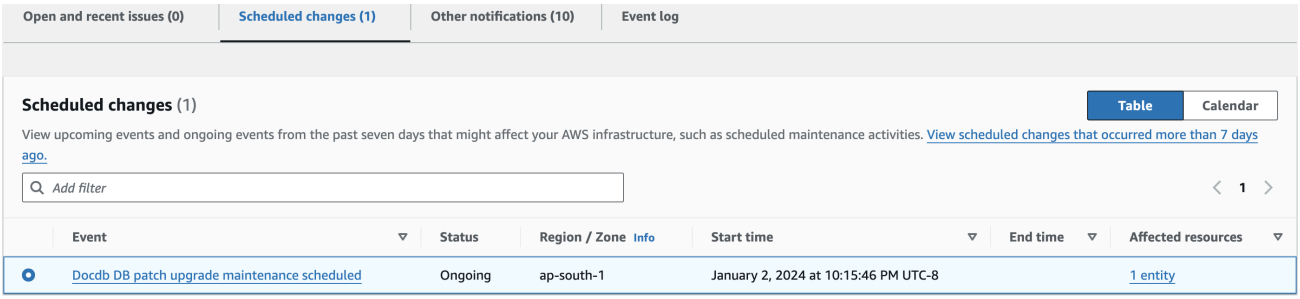
To change the maintenance window

- For a cluster, see Modifying an Amazon DocumentDB cluster.
- For an instance, see Modifying an Amazon DocumentDB instance.

Notifications for Amazon DocumentDB engine patches

When a *required* engine patch becomes available in an AWS Region, every AWS account with an affected Amazon DocumentDB cluster in that Region receives a notification through the

Health Dashboard (AHD) and through e-mail (sent to the AWS account's root user address). One notification is delivered per affected Amazon DocumentDB engine version. You can find them under **Scheduled changes** in the AHD. Each notification lists the patch availability timing, the auto-apply schedule, the affected clusters, and the release notes.



Required engine patches follow a single lead time of approximately 30 days. When a patch becomes available in your Region, Amazon DocumentDB sends the notification described above. At that point, the patch's `AutoAppliedAfterDate` is set to approximately 30 days later. Until that date, the patch stays pending: You can apply it at any time, or defer it by moving your cluster's maintenance window to a later day. On or after the `AutoAppliedAfterDate`, the patch auto-applies during the cluster's next maintenance window.

For example, a required patch that becomes available on June 1, 2026 has an `AutoAppliedAfterDate` of approximately July 1, 2026. You receive the notification on June 1, 2026, and if you take no action, the patch auto-applies during your cluster's first maintenance window on or after July 1, 2026.

You have two options after receiving the notification: self-apply the patch before the auto-apply date, or wait for it to auto-apply during an upcoming maintenance window (the default). To self-apply, open the cluster's **Maintenance & backups** tab and look for the entry of type `system-update`.

> **Note**
>
> The notification's **Status** in the AHD stays **Ongoing** until Amazon DocumentDB releases another engine patch with a new patch version.
> After the patch is applied, the cluster's engine patch version updates to match the version in the notification. Verify the new version by running `db.runCommand({getEngineVersion: 1})`.

Optional patches and new minor versions don't generate AHD or e-mail notifications. To track them, watch the Amazon DocumentDB Release notes.

Forced patches (the rarest category, reserved for the most critical security fixes) are also announced through AHD and e-mail. Unlike required patches, they apply outside your maintenance window, so the auto-apply timing example above does not apply.

Reacting to patch notifications programmatically

AWS Health integrates with Amazon EventBridge, which lets you build event-driven applications across more than 20 targets, including AWS Lambda and Amazon Simple Queue Service (SQS). To react to engine-patch availability programmatically, configure EventBridge against the `AWS_DOCDB_DB_PATCH_UPGRADE_MAINTENANCE_SCHEDULED` event. From there you can capture event data, raise additional events, send push notifications via the AWS Console Mobile Application, or take any other action you need.

If Amazon DocumentDB cancels a patch (rare), you receive an AHD notification and an e-mail about the cancellation. Use the `AWS_DOCDB_DB_PATCH_UPGRADE_MAINTENANCE_CANCELLED` event code with Amazon EventBridge to handle this case. For more about writing rules, see the Amazon EventBridge User Guide.

Viewing pending Amazon DocumentDB maintenance actions

Use the AWS Management Console or the AWS CLI to check what maintenance is pending for a cluster or instance.

Pending updates appear with the action type `system-update`, which covers both engine patches and OS updates.

When an update is pending, you can:

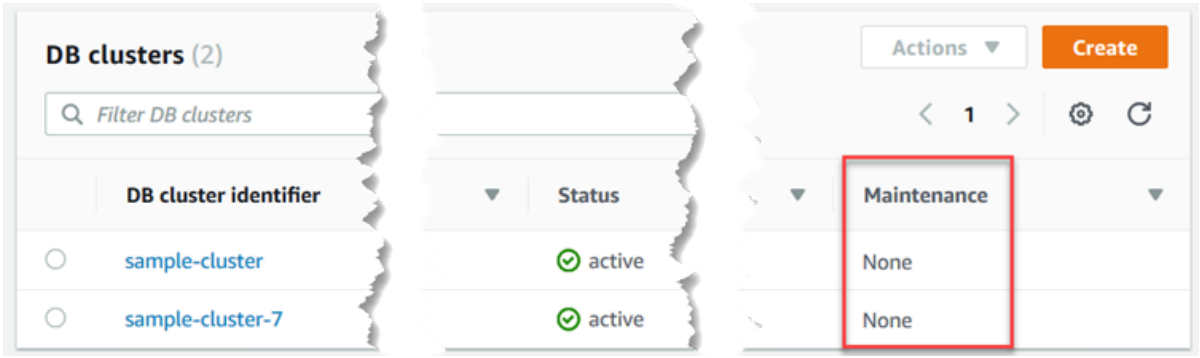
- Apply it immediately.
- Schedule it for the next maintenance window.
- Defer it (engine patches and OS updates only) by changing your maintenance window before `AutoAppliedAfterDate`. Once that date passes, the action will auto-apply during the next maintenance window. Once `ForcedApplyDate` passes, no further deferral is possible.

> **Note**
>
> If you take no action, required maintenance actions such as required engine patches auto-apply during an upcoming maintenance window. Optional patches and minor versions never auto-apply.

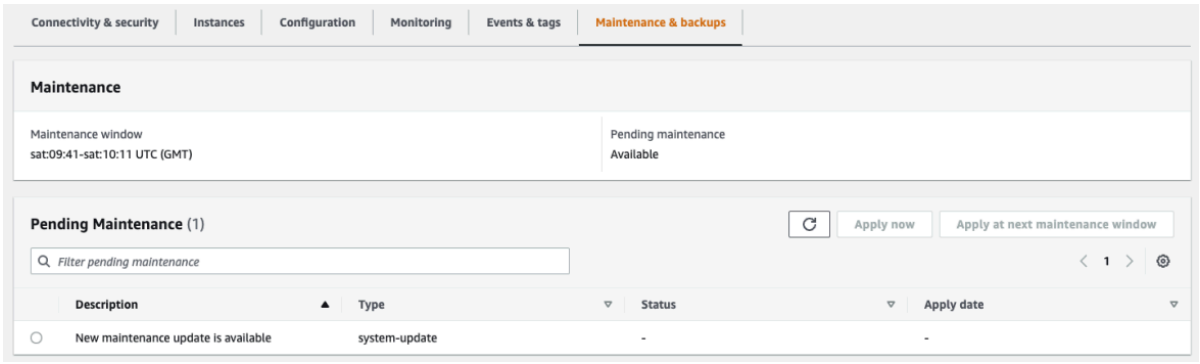
The maintenance window controls when pending operations *start*, not how long they take to complete.

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. The cluster's **Maintenance** column shows **Available**, **Required**, or **Next Window** when an update is pending.



4. Open the cluster, then choose **Maintenance & backups** to see the **Pending Maintenance** items and act on them.



Using the AWS CLI

Run `describe-pending-maintenance-actions` to see what's pending. The following example shows an account with no pending actions.

```
aws docdb describe-pending-maintenance-actions
```

Output from this operation looks something like the following (JSON format).

```
{
    "PendingMaintenanceActions": []
}
```

An account with a pending action returns output that looks like this:

```
{
    "PendingMaintenanceActions": [
        {
            "ResourceIdentifier": "arn:aws:rds:us-
east-1:123456789012:cluster:sample-cluster",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "system-update",
                    "Description": "db-version-upgrade",
                    "CurrentApplyDate": "2026-05-15T03:01:00Z",
                    "AutoAppliedAfterDate": "2026-05-15T03:01:00Z"
                }
            ]
        }
    ]
}
```

You can scope the listing to specific clusters with `--filters`, in the form `Name=`*`filter-name`*`,Values=`*`resource-id`*`,...`. The accepted filter `Name` is `db-cluster-id`, which takes a list of cluster identifiers or ARNs.

Example

For Linux, macOS, or Unix:

```
aws docdb describe-pending-maintenance-actions \
```

```
    --filters Name=db-cluster-id,Values=sample-cluster1,sample-cluster2
```

For Windows:

```
aws docdb describe-pending-maintenance-actions ^
    --filters Name=db-cluster-id,Values=sample-cluster1,sample-cluster2
```

Apply dates

Each pending maintenance action carries up to three apply dates. They appear in the AWS CLI output for `describe-pending-maintenance-actions` and indicate when the action will run. Fields are `null` for optional maintenance.

- **`CurrentApplyDate`**—when the action is scheduled to run, either now or at the next maintenance window. Populated for required and forced actions.
- **`AutoAppliedAfterDate`**—the date after which auto-apply begins during the cluster or instance maintenance window. Populated for required actions.
- **`ForcedApplyDate`**—the hard deadline. After this date the action runs automatically, regardless of your maintenance window. Populated for forced actions.

To defer a pending action, move your maintenance window to a later day before `AutoAppliedAfterDate`. Once `AutoAppliedAfterDate` passes, the action will auto-apply during the next maintenance window. Once `ForcedApplyDate` passes, no further deferral is possible. The exact deferral window varies per patch; the dates are published in the AHD notification and the AWS CLI output.

Amazon DocumentDB engine updates

When you've identified a pending engine patch, use one of the following procedures to apply or schedule it. You can run these procedures from either the AWS Management Console or the AWS CLI.

Using the AWS Management Console

To manage an update for a cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

2. In the navigation pane, choose **Clusters**.
3. Select the cluster you want to update.
4. From the **Actions** menu, choose either:

   - **Upgrade now**—run the pending maintenance immediately.
   - **Upgrade at next window**—run it during the cluster's next maintenance window.

   You can also use **Apply now** or **Apply at next maintenance window** from the **Pending Maintenance** section of the cluster's **Maintenance & backups** tab (see Viewing pending Amazon DocumentDB maintenance actions).

   > **Note**
   >
   > If nothing is pending, all of these options are inactive.

Using the AWS CLI

Apply a pending update with `apply-pending-maintenance-action`.

Parameters

- **`--resource-identifier`**—the Amazon DocumentDB Amazon Resource Name (ARN) of the resource the pending action targets.
- **`--apply-action`**—the pending maintenance action to apply. Use `system-update` to apply an engine patch.
- **`--opt-in-type`**—the type of opt-in request, or whether to undo one. Valid values:
  - `immediate`—apply now. Cannot be undone once submitted.
  - `next-maintenance`—apply during the resource's next maintenance window.
  - `undo-opt-in`—cancel an existing `next-maintenance` opt-in.

Example

For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
```

```
    --resource-identifier arn:aws:rds:us-east-1:123456789012:db:sample-cluster-
instance-1 \
    --apply-action system-update \
    --opt-in-type immediate
```

For Windows:

```
aws docdb apply-pending-maintenance-action ^
    --resource-identifier arn:aws:rds:us-east-1:123456789012:db:sample-cluster-
instance-1 ^
    --apply-action system-update ^
    --opt-in-type immediate
```

Read availability during patching

Amazon DocumentDB engine 5.0 and 8.0 preserve read availability during patching when the cluster has multiple instances. Amazon DocumentDB patches reader instances in a rolling fashion, in three groups, so the remaining readers continue serving traffic. The writer is briefly unavailable while it patches. To achieve zero read downtime, set your read preference so reads can fall back to the writer: `secondaryPreferred` or `primaryPreferred` work; `primary` or `secondary` alone may incur read downtime.

Read preference mode	During writer upgrade	During reader upgrade	Minimum number of readers needed for zero read downtime
`primary`	Read/write downtime	No impact	N/A
`primaryPreferred`	Write downtime	No impact	1
`secondary`	Write downtime	Read downtime (if only one reader)	2
`secondaryPreferred`	Write downtime	No impact	1
`nearest`	Write downtime	No impact	1

While readers are patching, overall cluster read throughput drops temporarily. To hold throughput steady, provision additional readers before the upgrade and remove them after it completes.

On engine 3.6 and 4.0, these read-availability features don't apply: an engine patch causes longer downtime that affects both reads and writes. To upgrade to a major version that does, see Amazon DocumentDB in-place major version upgrade.

Patch downtime length

Engine-patch downtime varies. The biggest factors are CPU utilization and memory pressure on the instance at the time of the patch, so right-sizing your instances matters. To minimize downtime, run the latest Amazon DocumentDB major engine version and spread instances across multiple Availability Zones.

Patch updates and replacements

Amazon DocumentDB monitors patches after release. In the rare case that an issue is identified, Amazon DocumentDB pauses the rollout while it prepares an updated version. When this happens, clusters that haven't yet received the patch no longer see it as an available maintenance action, and the corresponding scheduled-change notification in the Health Dashboard is withdrawn. Clusters already running the affected version continue to operate normally and require no action from you.

An updated patch follows shortly. When it becomes available in your Region, you receive a fresh notification through the Health Dashboard and e-mail, just as described in Notifications for Amazon DocumentDB engine patches.

Minor version upgrades

Amazon DocumentDB publishes minor versions on top of major version 5.0 and later (for example, `5.0.1`). Minor versions are not published for major versions earlier than 5.0. Minor versions behave differently from required and optional engine patches:

- They don't appear as a pending maintenance action and never auto-apply.
- They don't generate AHD or e-mail notifications. New minor versions are announced in the Amazon DocumentDB Release notes.
- To upgrade, you modify the cluster's engine version (immediately or during the next maintenance window). Minor version upgrades require brief downtime and are one-way—you can't downgrade to an earlier minor version. For global clusters, upgrade secondary clusters before the primary.

Read more: Amazon DocumentDB minor version upgrade.

Amazon DocumentDB operating system updates

Instances occasionally need OS updates. Amazon DocumentDB updates the OS to improve performance and tighten security. OS updates leave the cluster engine version and instance class unchanged. Like engine patches, OS updates use the optional / required / forced lifecycle described at the top of this topic; unlike engine patches, an OS update can transition through these categories over time if you defer it. Apply OS updates as soon as they are available, and set your cluster and instance maintenance windows to times that fit your business needs.

Use the cluster-level `os-upgrade` maintenance action to apply OS updates across all instances in a cluster. Amazon DocumentDB updates instances in a rolling manner, a few at a time, and updates the primary instance last to minimize failovers. The update runs during the cluster maintenance window—not the individual instance maintenance window—that you configure.

After an instance receives an OS update, its buffer cache starts empty. Until the working set is repopulated from the storage volume, queries on that instance can experience higher latency and a lower `BufferCacheHitRatio`.

When Amazon DocumentDB updates the primary instance, a failover promotes a replica to be the new primary. Use the cluster endpoint so your application handles this transparently. To maintain read availability while instances are being updated, set your read preference to `secondaryPreferred` or `primaryPreferred` so reads can fall back to an available instance. Keep potential failover targets (replicas with the highest-priority tier) at the same instance class as the primary. This avoids write-performance degradation after promotion. For details, see Amazon DocumentDB failover.

Both the cluster-level `os-upgrade` and instance-level `system-update` actions might appear simultaneously in `describe-pending-maintenance-actions` as available actions. However, you cannot schedule both at the same time. If instance-level `system-update` actions are actively scheduled on any instance, you must cancel or complete them before scheduling the cluster-level `os-upgrade` action, and vice versa.

> **Important**
>
> Your Amazon DocumentDB instance goes offline for the OS update. Multi-instance clusters minimize the impact. If you run a single-instance cluster, you can temporarily add a

secondary for the update and remove it afterwards. The secondary incurs the usual charges while it exists.

Note

The instance-level `system-update` action is still available for backward compatibility. If you must use it, update replicas first and the primary last—avoid patching them simultaneously, as a failover during the patch can extend downtime.

To get an event when a new optional OS update arrives, subscribe to `RDS-EVENT-0230` in the security patching event category. For more information, see Subscribing to Amazon DocumentDB events.

Note

Staying current on optional and required updates might be required for compliance. Apply `os-upgrade` actions routinely during your maintenance windows.

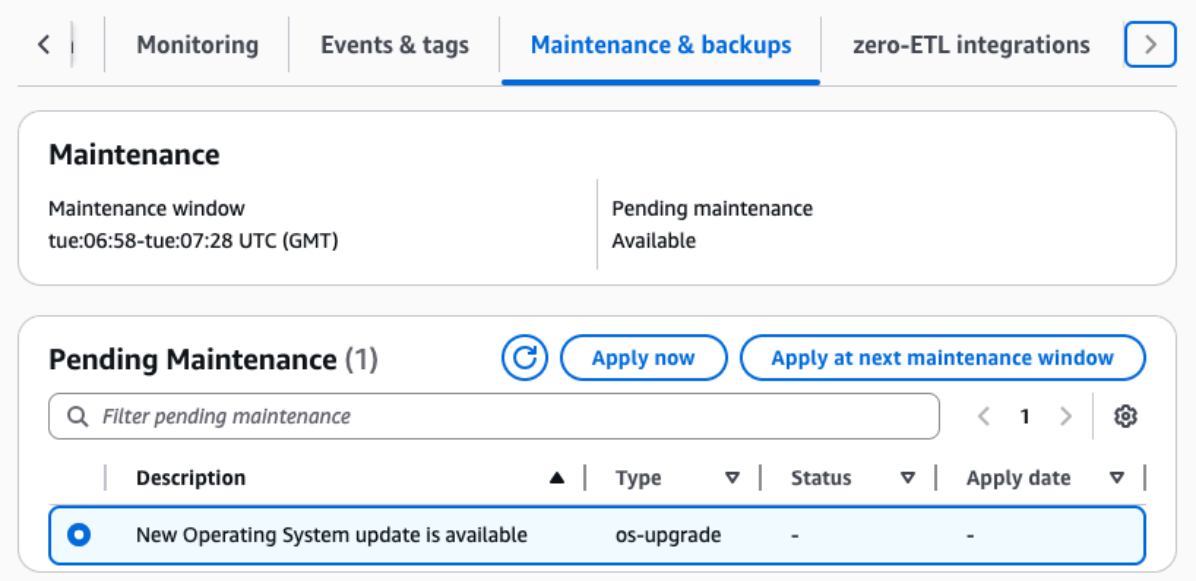
OS updates are tied to specific instance classes, so different instances become eligible at different times. If your cluster isn't on the latest engine patch, the OS update might not appear—apply the latest engine patch first (see Amazon DocumentDB engine updates).

Use the AWS Management Console or AWS CLI to check whether an update is available.

Using the AWS Management Console

To check for an OS update from the console:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**, then select the cluster name.
3. Choose the **Maintenance & backups** tab.
4. Under **Pending Maintenance**, the `os-upgrade` action appears if an OS update is available.



5. Select the `os-upgrade` action and choose **Apply now** or **Apply at next maintenance window**. If the value is **next window**, you can defer with **Defer upgrade** as long as the action has not started.

Using the AWS CLI

Check for a pending OS update:

```
aws docdb describe-pending-maintenance-actions
```

```
{
    "PendingMaintenanceActions": [
        {
            "ResourceIdentifier": "arn:aws:rds:aa-
example-1:111122223333:cluster:sample-cluster",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "os-upgrade",
                    "Description": "New Operating System update is available"
                }
            ]
        },
        {
            "ResourceIdentifier": "arn:aws:rds:aa-example-1:111122223333:db:sample-
cluster-instance-1",
            "PendingMaintenanceActionDetails": [
```

```
                {
                    "Action": "system-update",
                    "Description": "New Operating System update is available"
                }
            ]
        },
        {
            "ResourceIdentifier": "arn:aws:rds:aa-example-1:111122223333:db:sample-
cluster-instance-2",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "system-update",
                    "Description": "New Operating System update is available"
                }
            ]
        }
    ]
}
```

OS updates appear at the cluster level as `os-upgrade` and at the instance level as `system-update`. Use the cluster-level `os-upgrade` action.

Example

The following example applies the OS update immediately.

For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
    --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster \
    --apply-action os-upgrade \
    --opt-in-type immediate
```

For Windows:

```
aws docdb apply-pending-maintenance-action ^
    --resource-identifier arn:aws:rds:aa-example-1:111122223333:cluster:sample-
cluster ^
    --apply-action os-upgrade ^
    --opt-in-type immediate
```

User-initiated updates

Some changes you start yourself—for example, swapping an instance class for one with more or less memory, or changing the cluster's parameter group. Amazon DocumentDB treats these differently from updates it initiates. For details, see:

- Modifying an Amazon DocumentDB cluster
- Modifying an Amazon DocumentDB instance

To list user-initiated changes that are still pending:

Example

To list pending user-initiated changes for your instances

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
    --query 'DBInstances[*].
[DBClusterIdentifier,DBInstanceIdentifier,PendingModifiedValues]'
```

For Windows:

```
aws docdb describe-db-instances ^
    --query 'DBInstances[*].
[DBClusterIdentifier,DBInstanceIdentifier,PendingModifiedValues]'
```

Output from this operation looks something like the following (JSON format).

In this example, `sample-cluster-instance` has a pending change to `db.r5.xlarge`; `sample-cluster-instance-2` has none.

```
[
    [
        "sample-cluster",
        "sample-cluster-instance",
        {
            "DBInstanceClass": "db.r5.xlarge"
        }
```

```
    ],
    [
        "sample-cluster",
        "sample-cluster-instance-2",
        {}
    ]
]
```

Global clusters patching

In a global cluster, each member cluster—primary and secondary—upgrades during its own maintenance window. When a required engine patch is available in every Region, you receive an AHD and e-mail notification. Optional patches and new minor versions don't generate notifications; check the Amazon DocumentDB Release notes for those.

If you self-apply, always patch secondaries first and the primary last. This order keeps failover and switchover available throughout the rollout.

> **Important**
>
> If you patch the primary first by mistake, bring all secondaries up to the same version as soon as possible. Failover and switchover remain disabled until every cluster is on the same version.

If you take no action, the patch auto-applies during each cluster's next maintenance window: secondaries first, then the primary at its window once the secondaries are done.

Keep the primary and secondary DB clusters on the same version. Managed cross-Region failover only works on a global database when every cluster shares the same engine version and patch level. The same applies if you add a new secondary that uses a newer engine version than the primary—create new secondaries on the primary's version before joining them to the global database.

After a patch notification, upgrade primary and secondary to the latest version at your earliest opportunity to keep failover and switchover working. If a failover or switchover request is rejected, compare the engine patch versions across clusters; if they don't match, apply the available patch on the lagging clusters.

Understanding service-linked roles

Amazon DocumentDB (with MongoDB compatibility) uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Amazon DocumentDB. Service-linked roles are predefined by Amazon DocumentDB and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes using Amazon DocumentDB easier because you don't have to manually add the necessary permissions. Amazon DocumentDB defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon DocumentDB can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete the roles only after first deleting their related resources. This protects your Amazon DocumentDB resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS Services That Work with IAM and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Amazon DocumentDB service-linked role permissions

Amazon DocumentDB (with MongoDB compatibility) uses the service-linked role named **AWSServiceRoleForRDS** to allow Amazon DocumentDB to call AWS services on behalf of your clusters.

The AWSServiceRoleForRDS service-linked role trusts the following services to assume the role:

- `docdb.amazonaws.com`

The role permissions policy allows Amazon DocumentDB to complete the following actions on the specified resources:

- Actions on `ec2`:
  - `AssignPrivateIpAddresses`
  - `AuthorizeSecurityGroupIngress`
  - `CreateNetworkInterface`
  - `CreateSecurityGroup`

- `DeleteNetworkInterface`
- `DeleteSecurityGroup`
- `DescribeAvailabilityZones`
- `DescribeInternetGateways`
- `DescribeSecurityGroups`
- `DescribeSubnets`
- `DescribeVpcAttribute`
- `DescribeVpcs`
- `ModifyNetworkInterfaceAttribute`
- `RevokeSecurityGroupIngress`
- `UnassignPrivateIpAddresses`

- Actions on `sns`:
  - `ListTopic`
  - `Publish`
- Actions on `cloudwatch`:
  - `PutMetricData`
  - `GetMetricData`
  - `CreateLogStream`
  - `PullLogEvents`
  - `DescribeLogStreams`
  - `CreateLogGroup`

Note

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. You might encounter the following error message:
Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later.
If you see this error, ensure that you have the following permissions enabled:

```
    "Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
    "Condition": {
        "StringLike": {
            "iam:AWSServiceName":"rds.amazonaws.com"
        }
    }
}
```

For more information, see Service-Linked Role Permissions in the *IAM User Guide*.

Creating an Amazon DocumentDB service-linked role

You don't need to manually create a service-linked role. When you create a cluster, Amazon DocumentDB creates the service-linked role for you.

If you delete this service-linked role and then need to create it again, you can use the same process to re-create the role in your account. When you create a cluster, Amazon DocumentDB creates the service-linked role for you again.

Modifying an Amazon DocumentDB service-linked role

Amazon DocumentDB does not allow you to modify the AWSServiceRoleForRDS service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can modify the description of the role using IAM. For more information, see Editing a Service-Linked Role in the *IAM User Guide*.

Deleting an Amazon DocumentDB service-linked role

If you no longer need to use a feature or service that requires a service-linked role, delete that role. That way you don’t have an unused entity that isn’t actively monitored or maintained. However, you must delete all of your clusters before you can delete the service-linked role.

Cleaning up an Amazon DocumentDB service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session using the console

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane of the IAM console, choose **Roles**, and then choose the name (not the check box) of the **AWSServiceRoleForRDS** role.
3. On the **Summary** page for the selected role, choose the **Access Advisor** tab.
4. On the **Access Advisor** tab, review the recent activity for the service-linked role.

> **Note**
>
> If you are unsure whether Amazon DocumentDB is using the AWSServiceRoleForRDS role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the Regions where the role is being used. If the role is being used, then you must wait for the session to end before you can delete the role. You cannot revoke the session for a service-linked role.

If you want to remove the AWSServiceRoleForRDS role, you must first delete *all* your instances and clusters. For information about deleting instances and clusters, see the following topics:

- Deleting an Amazon DocumentDB instance
- Deleting an Amazon DocumentDB cluster

Supported Regions for Amazon DocumentDB service-linked roles

Amazon DocumentDB supports using service-linked roles in all of the Regions where the service is available. For more information, see https://docs.aws.amazon.com/documentdb/latest/devguide/regions-and-azs.html#regions-and-azs-availability.

Performance optimization

This chapter provides guidance on optimizing Amazon DocumentDB performance and troubleshooting common performance issues.

Topics

- Common performance issues
- Anti-patterns
- Performance improvement tips
- Query plan analysis
- Index-only scans (covered queries)
- Plan cache filters
- Best practices for Amazon DocumentDB
- Managing Amazon DocumentDB indexes
- Managing collection-level document compression
- Managing dictionary-based compression in Amazon DocumentDB 8.0

Common performance issues

This section covers common performance issues you might encounter with Amazon DocumentDB and how to resolve them.

Topics

- High CPU Utilization
- Connection issues with Amazon DocumentDB
- Query running slow

High CPU Utilization

High CPU utilization is one of the issues you might encounter at some point while working with Amazon DocumentDB. This section provides insights to resolve the issue.

Identification - Spot the problem

High CPU utilization on Amazon DocumentDB instances causes application slowness by increasing latencies across all database operations. When Amazon DocumentDB CPU usage rises above optimal levels, query execution times increase, connection establishment takes longer, and application responsiveness decreases. CPU peaks on the primary instance slow write operations, while peaks on replicas result in higher latency for read operations.

Some of the Common Causes of High CPU Utilization in Amazon DocumentDB could be:

- **Collection Scans**: Queries without proper indexes force full collection scans, consuming excessive CPU to examine every document.
- **Connection Spikes**: Sudden surges in application connections overwhelm CPU resources needed for authentication and session management.
- **Complex Aggregations**: Multi-stage operations (sorting, grouping, joining) consume significant CPU, especially on large datasets without optimized indexes.
- **Inefficient Queries**: High CPU usage results from scanning large result sets and regex operations.
- **Write-Heavy Workloads**: Frequent updates and bulk inserts strain CPU with document processing, index updates, and transaction logging.

Investigate - Gather metrics

Amazon DocumentDB provides monitoring through Amazon CloudWatch. Amazon DocumentDB metrics can be broadly categorized into cluster level and instance level metrics. For resources related to CPU, memory, and connections, Look at the instance level metrics as the resource consumption is specific to the instance. Identify the instances (primary or replica) that are showing the CPU spikes and note the times of spike.

The CloudWatch metric for the CPU utilization for `CPUUtilization`.

Diagnose - Find root cause

After identifying the instance and timeline of the CPU spike, next is to diagnose for the cause that is impacting CPU usage by large. As there are multiple reasons that can cause CPU spike, lets identify what is contributing to it starting with:

Connections: Check CloudWatch metrics including, `DatabaseConnections` and `DatabaseConnectionsMax`, if there are spikes in connections and it coincides with the CPU spike time, the surge in connections over a short time is likely the cause of the CPU spike.

Queries: If it is not the connection, then a query might be the cause. If you have enabled performance insights, then go to the Performance insights console for the instance and see the queries on the instance. The other thing to look at queries is from profiler.

Resolve - Fix the issue

Connection bursts: If connection bursts are the cause of CPU spikes, consider reusing connections in your application or implementing connection pooling. For more information on optimizing high connections, see the section called “Connection issues”.

Query Distribution: If the CPU spikes are caused by read queries on the primary instance, consider redirecting read operations to replica instances if your query results can be eventually consistent.

Query optimizations: If a query or set of queries are causing the CPU to spike, consider optimizing the query such as adding an index if the query is not using one. For more information on optimizing a query, see the section called “Query running slow”.

Instance Scaling: If CPU utilization remains consistently high after optimization efforts, upgrade to a larger instance class with more CPU capacity to handle your workload requirements effectively.

Serverless: For unpredictable workloads causing short-duration CPU spikes, consider using Using Amazon DocumentDB serverless. Serverless instances automatically scale resources based on demand, eliminating the need to provision fixed instance sizes for such workloads.

Connection issues with Amazon DocumentDB

Identification - Spot the problem

Common Causes

Connection problems typically stem from three main areas:

Connection pool exhaustion occurs when an application reaches its maximum allowed connections to Amazon DocumentDB, either through the client-side connection pool limits or server-side instance limits. This condition leads to degraded application performance, timeouts, and potential failures as new connection requests are either queued or rejected.

Authentication overload occurs when Amazon DocumentDB experiences excessive concurrent authentication requests, particularly when processing more than 1,000 new connections in a short time period. During authentication maintenance, Amazon DocumentDB holds an exclusive lock on the sessions map, causing subsequent authentication attempts to queue until maintenance completes.

Configuration issues in Amazon DocumentDB often stem from misconfigurations in networking, security, and client settings. This includes items such as incorrect security group settings, improper VPC configuration, or SSL/TLS certificate problems. Understanding proper configuration is essential for maintaining secure and reliable database access.

Diagnose - Find root cause

Connection pools

The connection pool initializes when creating a MongoClient instance. Each pool maintains connections based on two key parameters:

minPoolSize - Minimum number of connections maintained

maxPoolSize - Maximum allowed connections

When a request needs a connection:

1. The pool checks for available idle connections
2. If none exist and pool size < maxPoolSize, it creates new connection
3. If at maxPoolSize, the request enters a wait queue
4. If the queue is full or timeout reached, it throws MongoWaitQueueFullException

The behavior of the wait queue is handled via these parameters:

waitQueueTimeoutMS - Maximum wait time for connection

waitQueueSize - Maximum queued requests

Here's an example of a problematic approach for connecting to Amazon DocumentDB where a new pool is created each time:

```
for(Request request : requests) {
```

```
    MongoClient client = MongoClients.create(settings);
    // Process request
    client.close();
}
```

Critical CloudWatch metrics to monitor are:

- `DatabaseConnections` - The number of connections (active and idle) open on an instance taken at a 1-minute frequency.
- `DatabaseConnectionsMax` - The maximum number of open database connections (active and idle) on an instance in a 1-minute period.
- `DatabaseConnectionsLimit` - The maximum number of concurrent database connections (active and idle) allowed on an instance at any given time.
- `LowMemNumOperationsThrottled` - The number of requests that are throttled due to low available memory in a 1-minute period.

For quotas per instance class, see Instance quotas.

Common warning signs of connection pool issues at the application level include:

- Increasing connection acquisition times
- Growing wait queue size
- Rising number of timeout exceptions

Authentication overload

Connecting to Amazon DocumentDB follows a flow similar to below:

Connection Request → SSL Handshake → Authentication → Session Creation → Connection Ready

When processing >1,000 new connections, additional connection requests will enter a queue for Authentication after completing the SSL handshake. Average connection times from your application will increase during these overload events.

Critical CloudWatch metrics to monitor are:

- `DatabaseConnections` - The number of connections (active and idle) open on an instance taken at a 1-minute frequency.

- `DatabaseConnectionsMax` - The maximum number of open database connections (active and idle) on an instance in a 1-minute period.
- `DatabaseConnectionsLimit` - The maximum number of concurrent database connections (active and idle) allowed on an instance at any given time.

Configuration issues

The most common configuration issue is caused when trying to connect to a private Amazon DocumentDB cluster endpoint from an environment without access to the private network environment. Amazon DocumentDB is virtual private cloud (VPC)-only and does not currently support public endpoints. You can't connect directly to your Amazon DocumentDB cluster from your laptop or local development environment outside of your VPC.

This will manifest in errors such as below:

```
Error: couldn't connect to server...
Failed to connect to...
exception: connect failed
connection attempt failed
```

Incorrect security group configurations can also cause connection failures. A Amazon DocumentDB cluster listens for connections on TCP port 27017 by default. Your application will fail if trying to connect to a port different from what the cluster was deployed with, or if the application is not covered in the ingress security group configuration for the cluster.

Incorrect certificate management can also lead to connection issues. By default, encryption in transit is enabled for newly created Amazon DocumentDB clusters. When encryption in transit is enabled, secure connections using TLS are required to connect to the cluster using the global-bundle.pem certificate. If you attempt to use the incorrect certificate, you will receive errors such as:

```
unable to get local issuer certificate
```

If attempting to connect to a cluster with TLS enabled without specifying the TLS parameters, you will receive errors such as:

```
Server selection timed out after 30000 ms
```

Resolve - Fix the issue

Connection pools: Review connection pooling by implementing or adjusting pool sizes to match workload requirements. Optimal pool configurations depend on your workload and requirements. You should keep a minPoolSize such that core connections are ready and available, and a maxWaitTime short enough to fail fast if the pool has been exhausted.

Here's an example of how to reuse a single pool without creating a new one each time:

```
MongoClient client = MongoClients.create(settings);
    for(Request request : requests) {
    // Process request
}
```

Authentication overload: Manage authentication by implementing gradual connection ramp-up and limiting new connections to 1,000 at a time. Use connection pooling to reuse authenticated connections effectively. To avoid overloading the Amazon DocumentDB cluster with connections, implement a connection ramp-up strategy.

```
public class ConnectionManager {
    private static final int BATCH_SIZE = 100;
    private static final int DELAY_MS = 1000;

    public void establishConnections(int totalRequired) {
        int established = 0;
        while (established < totalRequired) {
            int batch = Math.min(BATCH_SIZE, totalRequired - established);
            createConnections(batch);
            Thread.sleep(DELAY_MS);
            established += batch;
        }
    }
}
```

You can also configure your connection pool settings to limit the total number of allowed connections.

```
MongoClientSettings settings = MongoClientSettings.builder()
    .applyToConnectionPoolSettings(builder -> {
        builder.maxSize(500)                    // Limit total connections
               .minSize(10)                     // Maintain base connections
               .maxConnectionLifeTime(3600000)  // Rotate connections hourly
    })
    .applyToServerSettings(builder -> {
        builder.heartbeatFrequency(10000)       // Regular server checks
    })
    .build();
```

Configuration issues: Ensure your application has access to the private VPC and subnet where your Amazon DocumentDB resources are located. If using VPC Peering, check the developer guide Troubleshoot a VPC peering connection for more information. You can also review the Knowledge Center article How do I troubleshoot connectivity issues from the internet to Amazon EC2 instances within my VPC?.

For security group configuration, you must include an ingress rule in your Amazon DocumentDB security group to allow connections from your application.

```
{
  "SecurityGroupIngress": [
    {
      "IpProtocol": "tcp",
      "FromPort": 27017,
      "ToPort": 27017,
      "SourceSecurityGroupId": "<application-security-group>",
      "Description": "DocumentDB access from application tier"
    }
  ],
  "SecurityGroupEgress": [
    {
      "IpProtocol": "-1",
      "FromPort": -1,
      "ToPort": -1,
      "CidrIp": "0.0.0.0/0"
```

```
    }
  ]
}
```

If the cluster is configured with TLS encryption, download the TLS certificate for Amazon Amazon DocumentDB named global-bundle.pem and use it when connecting to the cluster.

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

Long-term Solutions

Instance scaling may be necessary through upgrading to a larger instance class or adding read replicas to distribute connection load. Proper load balancing implementation ensures optimal resource utilization across the cluster.

Application changes should focus on implementing robust connection handling, comprehensive monitoring, and adherence to connection pooling best practices. This includes proper error handling and connection lifecycle management.

Architecture improvements might involve adopting Amazon DocumentDB Serverless for variable workloads, implementing sophisticated retry logic, and designing for fault tolerance. Consider restructuring application architecture to better handle connection management.

Best practices

Connection pools

Through proper connection pool management and monitoring, applications can maintain stable database connectivity while preventing exhaustion scenarios that could impact system reliability and performance. Configure appropriate timeouts and size your pool based on your workload's characteristics.

Connection pool setting example

```
MongoClientSettings settings = MongoClientSettings.builder()
    .applyToConnectionPoolSettings(builder ->
        builder.maxSize(10))
    .applyToConnectionPoolSettings(builder ->
        builder.maxWaitQueueSize(2))
```

```
    .applyToConnectionPoolSettings(builder ->
        builder.maxConnectionIdleTime(10, TimeUnit.MINUTES))
    .build();
```

For more information, see Building resilient applications with Amazon DocumentDB – Part 1: Client configuration.

Authentication overload

Always implement connection pooling with appropriate values for parameters based on your workload. Use a gradual connection establishment technique and maintain persistent connections, where possible. Implement proper connection cleanup to ensure no idle resources are wasted.

Configuration issues

Ensure you have configured appropriate routing from your application to the Amazon DocumentDB resources. Utilize TLS for encryption in transit and implement least privilege access. Verify your Amazon DocumentDB credentials and validate connection string values.

Query running slow

Identification - Spot the problem

A slow-running query is one that exceeds the execution time expected for your business requirements. The definition of what constitutes a slow query varies across different workloads. You can identify slow queries through profiler logs or Performance Insights.

1. Enable the profiler if not already enabled. Profiler logs are written to Amazon CloudWatch under the Log Group: `/aws/docdb/<cluster-identifier>/profiler`. Use CloudWatch Logs Insights to query them.

   Example CloudWatch log analysis query to get the 10 slowest queries for the ecommerce.products collection:

   ```
   filter ns="ecommerce.products" | sort millis desc | limit 10
   ```

2. Use Performance Insights to identify expensive queries in near real-time. Enable Performance Insights if it is not already enabled.

a. Open the AWS Console, navigate to Amazon Amazon DocumentDB, then Performance Insights, and select your cluster instance.

b. Review the DB Load (AAS) timeline and Top queries (by DB load). Expand a query digest to view literal child statements.

c. Capture the queries that require analysis.

> **Note**
>
> Not all queries in Performance Insights may be inefficient or slow queries.

Investigate — Gather information

1. The profiler provides the query execution plan and key metrics associated with it, including millis, nreturned, and planSummary (index usage):


```
{
    "op": "query",
    "ts": 1721374275673,
    "ns": "test.perf",
    "command": {
        "find": "perf",
        "filter": {
            "threadRunCount": 0
        },
        "$db": "test",
        "lsid": {
            "id": {
                "$binary": "oO2wEtpgQIK+y9KGByYnsw==",
                "$type": "4"
            }
        },
        "$readPreference": {
            "mode": "secondaryPreferred"
        }
    },
    "cursorExhausted": true,
    "nreturned": 0,
    "responseLength": 0,
    "protocol": "op_query",
```

```
    "millis": 137,
    "planSummary": "IXSCAN",
    "execStats": {
        "stage": "FETCH",
        "nReturned": "0",
        "executionTimeMillisEstimate": "100.346",
        "inputStage": {
            "stage": "IXSCAN",
            "nReturned": "0",
            "executionTimeMillisEstimate": "100.342",
            "indexName": "threadRunCount_1"
        }
    },
    "client": "172.31.6.165:43154",
    "appName": "ProdAppTester14",
    "user": "adminuser"
}
```

To find the COLLSCAN queries:

```
filter planSummary="COLLSCAN" | sort millis desc | limit 20
```

2. Use Performance Insights to analyze query execution trends, such as waits, load, and resource impact (e.g., IOPS or CPU per query) in real time.

   As Performance Insights doesn't provide the query execution plan, capture the query and run explain("executionStats") on the query on your Amazon DocumentDB cluster:

   ```
   db.ecommerce.products.find().explain("executionStats")
   ```

3. Optionally, correlate profiler metrics with Performance Insights data (e.g., match high millis queries in profiler with top queries in Performance Insights).

Diagnose — Find root cause

In this step, diagnose the query plan to identify potential optimizations. Use the following flow - symptom → likely cause:

- **Symptom:** planSummary: "COLLSCAN"

  Cause: Missing or incorrect index.
- **Symptom:** aggregation slow with $group / $sort

  Cause: Aggregation pipeline may be processing too much data in memory.
- **Symptom:** high latencies while Performance Insights shows DB Load grouped by IO waits, though indexes are utilized (planSummary: "IXSCAN").

  Cause: I/O bound queries; indexes or working set exceeds available buffer cache on the instance.
- **Symptom:** PI shows CPU wait states, high AAS due to few queries

  Cause: CPU-bound queries (complex aggregations, $regex).
- **Symptom:** many slow queries but none show expensive planSummary

  Cause: external bottleneck (network, throttling, maintenance tasks, volume of queries).

Resolve — Fix the issue

When you implement fixes, always validate with explain("executionStats") and monitor Performance Insights DB Load.

1. **planSummary: "COLLSCAN"**
   - Create a targeted index.

     Example: For frequent queries that filter by { category, price } and sort by price descending:

     ```
     db.products.createIndex({ category: 1, price: -1 })
     ```
2. **aggregation slow with $group / $sort**
   - Push $match and $project early to reduce documents flowing into $group / $sort.
   - Limit the number of fields early in the pipeline:

```
db.products.explain("executionStats").aggregate([
  { $match: { category: "Electronics", price: { $gt: 0 } } },   // early filter
  { $project: { price: 1, category: 1 } },                     // reduce document
 size
  { $group: { _id: "$category", avgPrice: { $avg: "$price" } } }
])
```

3. **high latencies while Performance Insights shows DB Load grouped by IO waits**
   - Validate if BufferCacheHitRatio is low or ReadIOPS on the instance are high. Increase instance memory (upgrade instance class, e.g., r6g.large → r6g.xlarge) or use NVMe instance class.
   - Reduce index footprint.
   - Add read replicas to offload read traffic (use readPreference setting to redirect traffic on replicas if query is tolerant with eventual consistency).
4. **PI shows CPU wait states, high AAS due to few queries**
   - Replace expensive $regex with indexed prefix searches or $text index.
   - Batch writes (insert) to reduce write amplification.

> **Note**
>
> Always test your changes in a lower environment (non-production) before promoting those changes to Production.

Anti-patterns

This section covers common anti-patterns that can negatively impact Amazon DocumentDB performance.

Topics

- Compound index with more than 3 attributes
- Long running queries
- Low used and redundant indexes
- Multi-Key Indexes with Large Arrays

Compound index with more than 3 attributes

Overview

A compound or composite index maintains references to multiple fields within a single index structure. These indexes optimize performance for queries that filter on multiple fields simultaneously or combine filtering with sorting operations. They're also effective for single-condition queries on the leftmost indexed fields. The database leverages these index entries to efficiently locate matching documents without performing full collection scans.

Amazon DocumentDB can use compound index to support queries that include any leading subset of the indexed fields, a concept known as Index Prefixes. For example, if you have a compound index on `{state: 1, city: 1, zipcode: 1}`, Amazon DocumentDB can efficiently process queries that use field 'state' alone, fields 'state' and 'city' together, or all three fields 'state', 'city', and 'zipcode'. However, queries must use the fields from left to right without skipping any fields in between. This means queries using only field 'city' or 'zipcode', or combinations like 'state' and 'zipcode' (skipping 'city'), cannot fully utilize the index.

In most real-world scenarios, compound indexes with three or fewer attributes typically achieve optimal performance and resource efficiency. While it's feasible to have more than 3 attributes in the composite index, it often leads to increased resource consumption, outweighing the benefits.

Impact on the cluster

While it is tempting to create an index covering all the conditions in a query to achieve maximum theoretical performance, most data filtering occurs through the first 1-3 attributes in a composite index. Additional fields beyond this threshold contribute primarily to index size rather than meaningful query optimization.

Storage and I/O Overhead: Composite indexes with many attributes consume significantly more storage space than simpler alternatives. The size is directly proportional to the number of indexed attributes, and the size of the indexed values themselves.

Memory footprint: The large storage footprint of composite indexes with many attributes creates a corresponding footprint in memory, making your working set larger and displacing other frequently accessed data from the buffer pool.

Write Operations: Each document modification affecting multiple indexed fields requires updating the entire composite index entry, multiplying the work required to complete the write operation.

How to identify

Start by reviewing all indexes in your collection to identify composite indexes with more than three attributes:

```
// List all indexes for the collection
db.collection.getIndexes()

// Look for indexes with 3+ fields like:

// { "userId": 1, "status": 1, "category": 1, "priority": 1, "region": 1 }
// { "orderId": 1, "customerId": 1, "productId": 1, "timestamp": 1, "warehouse": 1 }
```

Remediation

If a composite index with more than three attributes is being used, identify the queries that utilize it and look for optimization opportunities. Consider replacing these indexes with more efficient ones containing three or fewer attributes. After implementing the new indexes, remove the old indexes that contain more than three attributes.

> **Note**
>
> Always coordinate with stakeholders and validate performance impact before dropping any indexes.

Follow the Equality, Sort, Range (ESR) rule when creating composite indexes.

Equality Sort Range (ESR) Rule: This ordering maximizes index efficiency by filtering the dataset first with equality conditions, then applying sort operations on the reduced set, and finally performing range scans on the smallest possible subset

- order fields as Equality (exact matches),
- Sort (ordering),
- Range (>, <, $in).

```
// Query pattern
db.orders.find({
  userId: "user123",                    // Equality
  price: { $gte: 50, $lte: 200 }      // Range
}).sort({ createdAt: -1 })             // Sort

// Optimal index following ESR rule
db.orders.createIndex({ userId: 1, createdAt: -1, price: 1 })
//                      Equality   Sort         Range
```

Long running queries

Overview

Long-running queries can cascade into cluster-wide performance issues. These queries can interfere with Amazon DocumentDB's Multi-Version Concurrency Control (MVCC) garbage collection process, leading to accumulation of old versioned documents and degraded performance across the cluster. Amazon DocumentDB implements a 2-hour server-side timeout as a safety mechanism to limit runaway queries from consuming resources indefinitely.

What are Long running queries:

- Queries that execute for extended periods (typically > 30 minutes)
- Open cursors that remain active for hours (2 hours time-out won't be applicable if the cursor is active)

Impact on the cluster

Long running queries can interfere with Garbage collection Process

- **Old Versions Accumulate**: Garbage collector cannot reclaim old document versions
- **Collection and Index Bloat**: Collection and Index entries accumulate over time, bloat increases and which can result in more storage cost.
- **CPU and Memory Pressure**: CPU and memory pressure increases due to inefficient processing of increased number old document versions, index entries, and transaction IDs.

Long Running Query → Blocks GC → Storage Growth → CPU and Memory Pressure → More Long Queries

Monitor and detect

1. To find long running queries, use the currentOp command.

```
// To find a query running for more than 30 mins
db.adminCommand({
    aggregate: 1,
    pipeline: [
        {$currentOp: {}},
        {$match:
            {$or:
                [{secs_running: {$gt: 1800}},
                 {WaitState: {$exists: true}}]}}],
    cursor: {}
});
```

2. To find cursors which are active for more than 30 minutes

```
// To find cursor which is running more than 30 mins
db.adminCommand({
    "currentOp": true,
    "active": true,
    "$all": true
}).inprog.filter(function(op) {
    return op.desc == "Cursor" &&
           op.secs_running > 1800 &&
           op.active == true;
}).sort((a, b) => b.microsecs_running - a.microsecs_running)
```

3. Monitoring Garbage collector progress through CloudWatch

`LongestRunningGCProcess`— Duration in seconds of the longest active garbage collection process. Updates every minute and tracks only active operations, excluding processes that complete within the one-minute window.

`AvailableMVCCIds` -A counter that shows the number of remaining write operations available before reaching zero. When this counter reaches zero, your cluster enters read-only mode until

IDs are reclaimed and recycled. The counter decreases with each write operation and increases as garbage collection recycles old MVCC IDs.

> **Note**
>
> Lower MVCC IDs and extended garbage collection duration are not exclusively attributed to long-running queries. Write-intensive workloads on resource-constrained instances can also result in reduced MVCC ID availability and prolonged garbage collection cycles.

Remediation strategies

- Implement query time-outs in the application
- Do not keep the cursors alive for longer durations
- Optimize the queries for better performance.
- Prefer batching of write operations

Low used and redundant indexes

Overview

Indexes improve overall query performance of workloads, but come with costs. Each index consumes storage and I/O resources, and write operations need to update all indexes on the collection. If your database contains unused, underutilized, or redundant indexes, it results in overhead to the cluster, leading to performance degradation. Multiple unnecessary indexes create an anti-pattern that causes performance issues, increased storage costs, and higher operational overhead.

Sub optimal indexing scenarios

- **Unused Indexes**: Indexes created for one-time queries, or in earlier product iterations that are no longer accessed by current query patterns. To identify unused indexes on a collection you can use indexStats. For more information, see the section called “Analyze index usage and identify unused indexes”.
- **Redundant Indexes**: Multiple indexes covering overlapping key patterns or identical query patterns. Example: A single-key index on attribute A becomes redundant when a composite index covers both attribute A and B, since the composite index can handle queries on attribute A alone.

- **Over-Indexing**: Creating indexes "just in case" without analyzing actual query patterns or performance requirements.
- **Low Cardinality Indexes**: Indexes on fields with few distinct values (e.g., boolean fields, status flags) that provide minimal query optimization benefit.

Impact on Cluster Performance

- **Storage and IO**: Each index consumes storage space and IO resources, contributing to overall costs. If an index is unused or redundant, it represents unnecessary cost on your cluster.
- **Degraded Write Performance**: Insert, update, and delete operations must maintain indexes, creating overhead that consumes resources. Unused or redundant indexes add unnecessary overhead to these operations.
- **Memory and CPU Pressure**: Indexes compete for buffer pool memory and may evict frequently accessed data if the working set doesn't fit in instance memory resulting in suboptimal BufferCacheHitRatio. The system also consumes CPU resources maintaining index structures.

Tools available to identify index optimization opportunities

- Index Review Tool:
- Index Cardinality Detection Tool

Gathering the toolkit:

```
git clone https://github.com/awslabs/amazon-documentdb-tools.git
```

1. Index Review Tool

The index review tool analyzes all collections and indexes to provide a listing of indexes, their usage patterns, and identifies redundant indexes on your Amazon DocumentDB cluster. Execute this tool on all cluster instances for comprehensive analysis. For more information, see Index Review Tool.

Usage:

```
cd performance/index-review/
```

```
python3 index-review.py --server-alias <server-alias> --uri <mongodb-uri>
```

Output Files

- `{server-alias}-collections.csv`: Collection-level statistics
- `{server-alias}-indexes.csv`: Detailed index usage metrics
- `{server-alias}-{timestamp}-index-review.json`: Raw data for further analysis

> **Note**
>
> Run for each instance in the cluster with an independent alias to generate stats from each instance, and then run the tool to get you a cumulative analysis

Eg: `python3 index-review.py --files "{server-alias}-{timestamp}-index-review.json, {server-alias}-{timestamp}-index-review.json" --server-alias full-review`

2. Index Cardinality Detection Tool

This tool identifies indexes with low cardinality that are inefficient for query performance.

Usage

```
cd performance/index-cardinality-detection/

python3 detect-cardinality.py --uri <mongodb-uri>
```

Parameters

- `--threshold`: Cardinality threshold percentage (default: 1%)
- `--sample-count`: Number of documents to sample (default: 100,000)
- `--max-collections`: Maximum collections to scan per database (default: 100)

For more information, see https://aws.amazon.com/blogs/database/detect-and-fix-low-cardinality-indexes-in-amazon-documentdb/

Remediation strategies

1. Unused and redundant Index Removal

```
// Example: Drop unused index
db.collection.dropIndex("unused_index_name")
```

> **Note**
>
> Indexes should never be dropped without discussing with all interested parties and testing performance.

2. Low Cardinality Index Optimization

- Use partial indexes with filters
- Convert single low-cardinality indexes to compound indexes

Multi-Key Indexes with Large Arrays

Overview

Multi-key indexes in Amazon DocumentDB let you efficiently query array fields. This indexing strategy delivers significantly lower query latencies when searching within array data, making it an attractive option for applications with complex data structures.

However, when working with large arrays or multiple array fields within a collection, it's important to understand the resource implications and performance characteristics to make informed architectural decisions. When you create an index on an array field, Amazon DocumentDB generates individual index entries for each element in the array.

Impact on the cluster

- **Storage and IO Overhead**: Multi-key indexes can sometimes consume storage space upto multiple times of the base table, the size is directly proportional to documents in the collection with the indexed array attribute, number of elements in the array and size of the element.
- **Memory Usage**: As the storage footprint is relatively large, these indexes have a corresponding footprint in memory, resulting in a larger working set.
- **Write operations**: Multi-key indexes create overhead during the write operations. Each array element generates separate index entries, multiplying the work required to complete the write operation.

Remediation strategies

- Create multi-key index if it is necessary and limit the number of fields in the array.
- Limit the number of multi key indexes on the collection
- Consider modifying your data model

Performance improvement tips

This section provides five performance optimization strategies for Amazon DocumentDB to improve application efficiency and query execution.

1. Use $match as First Stage in Aggregation Pipelines

Always place $match as the first stage for filtering in your aggregation pipeline to maximize performance. Amazon DocumentDB will utilize indexes effectively when $match leads the pipeline, allowing the database to filter data early and reduce processing overhead.

```
// Optimized approach
db.orders.aggregate([
  { $match: { status: "active", category: "electronics" } }, // Index utilization
  { $group: { _id: "$category", total: { $sum: "$price" } } },
  { $sort: { total: -1 } }
])
```

Impact: Early filtering reduces the number of documents processed in subsequent pipeline stages, resulting in faster query execution and lower resource consumption.

2. Use $project in Aggregation Pipeline to Minimize Pipeline Data Size

Carry only essential fields through your aggregation pipeline stages to minimize data size and improve performance. Use $project strategically to include just the data you need.

```
// Efficient pipeline design
db.orders.aggregate([
  { $match: { orderDate: { $gte: new Date("2024-01-01") } } },
  { $project: { customerId: 1, totalAmount: 1, status: 1 } }, // Only needed fields
  { $group: { _id: "$customerId", totalSpent: { $sum: "$totalAmount" } } }
])
```

Impact: Smaller documents reduce memory usage and improve pipeline processing efficiency, resulting in enhanced overall query performance.

3. Enable Document Compression for Lower Storage Costs, I/O Costs, and Improved Query Performance

Enable document compression from cluster parameter group to lower storage costs, I/O costs, and boost query performance. Amazon DocumentDB stores compressed documents on disk as well as in RAM, reducing memory footprint and I/O costs.

Impact:

- More documents fit in available memory
- Faster data access with reduced disk reads
- Lower storage costs, I/O costs, and improved query performance

> **Note**
>
> Amazon DocumentDB doesn't enable compression by default for version 5.0. You can enable compression at collection or cluster level for 5.0 cluster. Use Amazon DocumentDB's compression review utility to analyze compression ratios for your collections.

For Amazon DocumentDB 8.0, compression is enabled by default.

4. Leverage Indexes for Optimal Query Performance

Ensure your queries always utilize indexes for optimal performance. Amazon DocumentDB offers multiple index types to match different use cases.

Indexing principles:

- Every query should leverage an appropriate index
- Amazon DocumentDB provides multiple index types
- Compound indexes offer the most flexibility by supporting various query shapes with a single index
- Design indexes to support sorting and filtering operations together

Understanding Index Prefixes: Compound indexes work through index prefixes - Amazon DocumentDB can use any left-to-right subset of the index fields. For example, the index `{ category: 1, price: -1, inStock: 1 }` creates these usable prefixes:

- `{ category: 1 }` - supports queries filtering by category only
- `{ category: 1, price: -1 }` - supports queries filtering by category and sorting/filtering by price
- `{ category: 1, price: -1, inStock: 1 }` - supports the full compound query

Queries on price, inStock, or inStock alone will not use this index since they don't start with the first field (category).

How to Identify Queries Not Using Indexes: Use the explain() method to analyze query execution and identify queries performing collection scans instead of using indexes.

Impact: Queries without index utilization result in collection scans, causing increased memory and CPU pressure on the instance and elevated query latency.

5. Optimize Data Models Based on Query Patterns

Align your data model with how your application queries and updates data. Data modeling is the foundation of high-performance Amazon DocumentDB applications.

Optimization strategies:

Embedding for Performance

- Store related data together when it's frequently accessed as a unit
- Embed documents that are always retrieved together
- Suitable for one-to-few relationships

```
// Embedded approach for frequently accessed data
{
  _id: ObjectId("..."),
  customerName: "John Doe",
  address: {
    street: "123 Main St",
    city: "Seattle",
    zipCode: "98101"
  },
  recentOrders: [
    { orderId: "ORD001", amount: 99.99, date: "2024-01-15" }
  ]
}
```

Referencing for Flexibility

- Use references for large or infrequently accessed data
- Recommended for one-to-many relationships with large datasets
- Prevents document bloat and improves update performance

Collection Splitting Strategy

When only a few fields in large documents get updated frequently, or when large infrequently accessed data bloats documents, consider splitting collections:

- Keep frequently updated fields in a separate, smaller collection
- Store static or infrequently accessed data in another collection
- Link them with references when needed

```
// Before: Large document with mixed access patterns
{
  _id: ObjectId("..."),
  productId: "PROD123",
  name: "Wireless Headphones",        // Frequently accessed
  price: 99.99,                       // Frequently accessed
  inventory: 45,                      // Updated frequently
  lastSold: "2024-01-15",           // Updated frequently
  detailedSpecs: { /* large object */ }, // Infrequently accessed
  manualPDF: "base64...",           // Large, rarely accessed
  reviewHistory: [/* large array */] // Infrequently accessed
}

// After: Split into collections based on access patterns
// products collection (frequently accessed data)
{
  _id: ObjectId("..."),
  productId: "PROD123",
  name: "Wireless Headphones",
  price: 99.99,
  inventory: 45,
  lastSold: "2024-01-15"
}

// product_details collection (infrequently accessed data)
{
  _id: ObjectId("..."),
  productId: "PROD123",           // Reference to products collection
  detailedSpecs: { /* large object */ },
  manualPDF: "base64...",
  reviewHistory: [/* large array */]
}
```

Performance gain: Smaller documents mean faster updates, reduced memory usage, and improved cache efficiency.

Impact: Inefficient data modeling results in suboptimal queries, increased document sizes, and elevated memory usage, leading to degraded application performance and higher operational costs.

Query plan analysis

Query plan analysis through explain plan provides essential insights into Amazon DocumentDB query performance. Query plan with executionStats reveals key metrics including:

- Documents returned per stage (nReturned)
- Stage-specific execution times (executionTimeMillisEstimate)
- Plan generation duration (planningTimeMillis)

By examining the query plan output, developers can analyze execution patterns, evaluate index utilization, and identify potential optimization opportunities across the query pipeline stages.

To analyze the query plan, you can use the explain() command in the following formats.

```
db.runCommand({explain: {query document}, verbosity: "executionStats"})
db.collection.find().explain("executionStats");
```

The following is an example operation:

```
db.collection.find({ companyname: { '$eq': 'ANYCOMPANY' }, isDeleted: { '$eq':
 false } }).sort({"createdAt":1}).limit(2).explain("executionStats");
```

Output from this operation looks something like the following:

```
{
  queryPlanner: {
    plannerVersion: 2,
    namespace: 'limit_test.test',
    winningPlan: {
      stage: 'LIMIT_SKIP',
      inputStage: {
        stage: 'SORT',
        sortPattern: { createdAt: 1 },
        inputStage: {
          stage: 'IXSCAN',
```

```
          indexName: 'companyname_1_createdAt_1_isDeleted_1',
          direction: 'forward',
          indexCond: {
            '$and': [
              { companyname: { '$eq': 'ANYCOMPANY' } },
              { isDeleted: { '$eq': false } }
            ]
          }
        }
      }
    }
  },
  indexFilterSet: false,
  indexFilterApplied: false,
  executionStats: {
    executionSuccess: true,
    executionTimeMillis: '4.186',
    planningTimeMillis: '3.909',
    executionStages: {
      stage: 'LIMIT_SKIP',
      nReturned: '2',
      executionTimeMillisEstimate: '0.199',
      inputStage: {
        stage: 'SORT',
        nReturned: '2',
        executionTimeMillisEstimate: '0.197',
        sortPattern: { createdAt: 1 },
        inputStage: {
          stage: 'IXSCAN',
          nReturned: '34',
          executionTimeMillisEstimate: '0.151',
          indexName: 'companyname_1_createdAt_1_isDeleted_1',
          direction: 'forward',
          indexCond: {
            '$and': [
              { companyname: { '$eq': 'ANYCOMPANY' } },
              { isDeleted: { '$eq': false } }
            ]
          }
        }
      }
    }
  },
  serverInfo: { host: 'demo-cluster', port: 27017, version: '5.0.0' },
```

```
  ok: 1,
  operationTime: Timestamp({ t: 1759915116, i: 1 })
}
```

Below is a detailed analysis of a Amazon DocumentDB query execution plan, breaking down each component and its performance characteristics.

Overall timing

executionTimeMillis represents the total time taken by the query including planning time.

planningTimeMillis represents the total planning time taken by the query.

Execution stages

It describes the step-by-step process Amazon DocumentDB uses to execute a query, showing how data flows through different operations.

```
"executionStages": {
    "stage": "[STAGE_NAME]",
    "nReturned": "[NUMBER_OF_DOCS]",
    "executionTimeMillisEstimate": "[TIME]",
    "inputStage": {
        // Nested stages
    }
}
```

Common stages in a query plan

Below are the common execution stages in a query plan. Each stage returns executionTimeMillisEstimate (execution time) and nReturned (number of documents) metrics to help evaluate query performance at every stage.

COLLSCAN (Collection Scan)

- Scans the entire collection document by document
- nReturned: Returns all matching documents in the collection

- Can be an expensive operation, seen when no index is available

IXSCAN (Index Scan)

- Uses an index to find matching documents
- nReturned: Only matching documents based on the index
- Efficient operation, preferred over COLLSCAN

SORT

- Sorts documents based on specified fields
- If the sort attribute in the query is not part of an index, this stage will appear explicitly
- nReturned: Same number as input documents
- Memory-intensive for large result sets. To optimize, add the sort field name to an index

LIMIT_SKIP

- Controls the number of documents returned and skipped
- nReturned: Limited by the LIMIT value
- Used for pagination or LIMIT operations

SUBSCAN

- Performs nested query operations
- nReturned: Varies based on subquery results
- Used in complex queries with multiple stages

Stages with high executionTimeMillisEstimate are good candidates for optimization.

> **Note**
>
> The executionStats parameter does not currently support update and delete commands.

Understanding docsExamined in Explain Plan Execution Stats

When you run a query with `explain("executionStats")`, Amazon DocumentDB provides examination metrics that help you understand how many documents were scanned to produce the query results. These metrics are useful for identifying inefficient query plans and optimizing performance.

> **Note**
>
> Available only in Amazon DocumentDB 5.0.1+ and 8.0.0+.

Fields

Field	Description	Level
totalDocsExamined	Total number of documents examined across all execution stages.	Top-level execution Stats
docsExamined	Number of documents examined by a specific execution stage.	Stage-level

The `docsExamined` field appears on the following stages:

Stage	Description
COLLSCAN	Collection scan. All documents in the collection are examined.
IXSCAN	Index scan. Only documents matching the index condition are examined.
FETCH	When the optimizer retrieves documents in a separate stage from IXSCAN, the FETCH stage reports docsExamined. In index scan plans, the child IXSCAN stages do not report docsExamined. In $lookup plans, both the FETCH stage and its child stages may report docsExamined.

> **Note**
>
> `docsExamined` does not appear on IXONLYSCAN stages because the query is satisfied entirely from the index without accessing documents.

Show docsExamined in Explain Plan executionStats Output

The following examples use a collection with 500,000 documents and demonstrate how `docsExamined` changes across different query execution stages.

Create sample data:

```
for (let i = 0; i < 500000; i++) {
    db.coll.insertOne({ number: i, arr: [i, [i+1]], value: "test", bool: i % 2 ===
 0 });
}
```

Collection scan (COLLSCAN)

Without an index, Amazon DocumentDB performs a full collection scan.

```
db.coll.find({ "number": { "$lt": 500 } }).explain("executionStats").executionStats
```

Output:

```
{
    "executionSuccess" : true,
    "nReturned" : "500",
    "executionTimeMillis" : "282.055",
    "planningTimeMillis" : "0.085",
    "totalDocsExamined" : "500000",
    "executionStages" : {
        "stage" : "COLLSCAN",
        "nReturned" : "500",
        "executionTimeMillisEstimate" : "281.915",
        "docsExamined" : "500000",
```

```
        "filter" : {
            "number" : {
                "$lt" : 500
            }
        }
    }
}
```

All 500,000 documents were examined to return 500 results. Creating an index on the number field improves this.

Index scan (IXSCAN)

```
db.coll.createIndex({ number: 1 });
db.coll.find({ "number": { "$lt": 5000 } }).explain("executionStats").executionStats
```

Output:

```
{
    "executionSuccess" : true,
    "nReturned" : "5000",
    "executionTimeMillis" : "3.047",
    "planningTimeMillis" : "0.296",
    "totalDocsExamined" : "5000",
    "executionStages" : {
        "stage" : "IXSCAN",
        "nReturned" : "5000",
        "executionTimeMillisEstimate" : "2.576",
        "indexName" : "number_1",
        "direction" : "forward",
        "docsExamined" : "5000",
        "indexCond" : {
            "$and" : [
                {
                    "number" : {
                        "$lt" : 5000
                    }
                }
            ]
```

```
        }
    }
}
```

With the index, only 5,000 out of 500,000 documents were examined and fetched, matching the number returned. The index condition filtered out 495,000 documents that did not match the query.

Index scan with residual filter

```
db.coll.find({ "number": { "$lt": 5000 }, "arr": { "$gt":
 4000 } }).explain("executionStats").executionStats
```

Output:

```
{
    "executionSuccess" : true,
    "nReturned" : "999",
    "executionTimeMillis" : "15.367",
    "planningTimeMillis" : "0.115",
    "totalDocsExamined" : "5000",
    "executionStages" : {
        "stage" : "IXSCAN",
        "nReturned" : "999",
        "executionTimeMillisEstimate" : "15.170",
        "indexName" : "number_1",
        "direction" : "forward",
        "docsExamined" : "5000",
        "indexCond" : {
            "$and" : [
                {
                    "number" : {
                        "$lt" : 5000
                    }
                }
            ]
        },
        "filter" : {
            "arr" : {
```

```
                "$gt" : 4000
            }
        }
    }
}
```

The index condition matched 5,000 documents out of 500,000, then the residual filter on `arr` reduced the result to 999. The `docsExamined` value of 5,000 reflects all documents examined after the index condition but before the residual filter was applied.

Fetch with IXSCAN

```
db.coll.find({ "$or": [{ "number": { "$lt": 100000 } }, { "number": { "$gt":
 400000 } }] }).explain("executionStats").executionStats
```

Output:

```
{
    "executionSuccess" : true,
    "nReturned" : "199999",
    "executionTimeMillis" : "899.801",
    "planningTimeMillis" : "0.183",
    "totalDocsExamined" : "199999",
    "executionStages" : {
        "stage" : "FETCH",
        "nReturned" : "199999",
        "executionTimeMillisEstimate" : "894.141",
        "docsExamined" : "199999",
        "inputStage" : {
            "stage" : "IXOR",
            "nReturned" : "0",
            "executionTimeMillisEstimate" : "874.897",
            "inputStages" : [
                {
                    "stage" : "IXSCAN",
                    "nReturned" : "100000",
                    "executionTimeMillisEstimate" : "462.208",
                    "indexName" : "number_1",
                    "indexCond" : {
```

```
                        "$and" : [
                            {
                                "number" : {
                                    "$lt" : 100000
                                }
                            }
                        ]
                    }
                },
                {
                    "stage" : "IXSCAN",
                    "nReturned" : "99999",
                    "executionTimeMillisEstimate" : "412.684",
                    "indexName" : "number_1",
                    "indexCond" : {
                        "$and" : [
                            {
                                "number" : {
                                    "$gt" : 400000
                                }
                            }
                        ]
                    }
                }
            ]
        }
    }
}
```

When Amazon DocumentDB optimizer uses a fetch stage to retrieve documents, the FETCH stage reports `docsExamined`. The child IXSCAN stages do not report `docsExamined` because they only scan index keys without accessing documents directly.

FETCH with Aggregate Lookup

```
db.coll.explain("executionStats").aggregate([
    { $match: { "number": { "$lt": 5 } } },
    { $lookup: { from: "coll", pipeline: [{ $match: { "number": { "$lt": 3 } } }], as:
 "sub" } }
]).executionStats
```

Output:

```
{
    "executionSuccess" : true,
    "nReturned" : "5",
    "executionTimeMillis" : "0.525",
    "planningTimeMillis" : "0.327",
    "totalDocsExamined" : "9",
    "executionStages" : {
        "stage" : "NESTED_LOOP_LOOKUP",
        "nReturned" : "5",
        "executionTimeMillisEstimate" : "0.163",
        "inputStages" : [
            {
                "stage" : "IXSCAN",
                "nReturned" : "5",
                "executionTimeMillisEstimate" : "0.039",
                "indexName" : "number_1",
                "direction" : "forward",
                "docsExamined" : "5",
                "indexCond" : {
                    "$and" : [
                        {
                            "number" : {
                                "$lt" : 5
                            }
                        }
                    ]
                }
            },
            {
                "stage" : "FETCH",
                "nReturned" : "1",
                "executionTimeMillisEstimate" : "0.009",
                "docsExamined" : "1",
                "inputStage" : {
                    "stage" : "AGGREGATE",
                    "nReturned" : "1",
                    "executionTimeMillisEstimate" : "0.044",
                    "inputStage" : {
                        "stage" : "IXSCAN",
                        "nReturned" : "3",
                        "executionTimeMillisEstimate" : "0.013",
```

```
                        "indexName" : "number_1",
                        "direction" : "forward",
                        "docsExamined" : "3",
                        "indexCond" : {
                            "$and" : [
                                {
                                    "number" : {
                                        "$lt" : 3
                                    }
                                }
                            ]
                        }
                    }
                }
            }
        ]
    }
}
```

The outer IXSCAN examined 5 documents matching number < 5. The inner IXSCAN examined 3 documents matching number < 3, and the FETCH stage examined 1 aggregated result. The `totalDocsExamined` of 9 is the sum across all stages (5 + 3 + 1).

Index-only scans (covered queries)

An **index-only scan** (also called a **covered query**) is an optimization in which Amazon DocumentDB answers a query entirely from the fields stored in an index, without reading the underlying documents. Because the values a query needs are already present in the index, Amazon DocumentDB can skip fetching each matching document, avoiding the random I/O and decoding cost of reading full documents. In the query plan, an index-only scan appears as the `IXONLYSCAN` stage instead of an `IXSCAN` (index scan) or `COLLSCAN` (collection scan). This reduces I/O and can significantly lower query latency for supported queries.

A query is a good candidate for an index-only scan when it touches only a small number of fields that all exist in a single index, especially when the documents themselves are large. Common examples are dashboards, analytics rollups, and lookups that read a few fields from a wider compound index.

The results of a query are the same whether or not an index-only scan is used. Only the query plan, and the amount of I/O it performs, change.

Note

Index-only scan support depends on your Amazon DocumentDB engine version:

- Covered `find` projections that include **more than one** non-`_id` field require engine version 8.0.2 or later. In earlier versions, a covered `find` projection can include at most one non-`_id` field. A projection with two or more fields falls back to a normal index scan or collection scan.
- Index-only scans for aggregation pipelines, such as a covered `$group`, require engine version 8.0.2 or later.

When Amazon DocumentDB uses an index-only scan

Amazon DocumentDB considers an index-only scan for the following operations when a single index contains every field the operation reads.

Covered `find` projections

A `find` query qualifies when its projection is a simple inclusion of one or more top-level fields and every projected field is present in a single index. For example, given a compound index on `{ customerId: 1, status: 1, total: 1 }`, the following query is covered. All three projected fields live in the index, and `_id` is explicitly excluded. (Covering a projection with more than one non-`_id` field requires engine version 8.0.2 or later.)

```
db.orders.createIndex({ customerId: 1, status: 1, total: 1 })

db.orders.find(
  { customerId: 42 },
  { _id: 0, customerId: 1, status: 1, total: 1 }
)
```

The winning plan uses an `IXONLYSCAN` stage and never fetches the matching documents from storage:

```
{
```

```
  "stage": "IXONLYSCAN",
  "indexName": "customerId_1_status_1_total_1",
  "direction": "forward",
  "indexCond": { "$and": [ { "customerId": { "$eq": 42 } } ] }
}
```

> **Note**
>
> In a covered `find` query, the fields in the result are returned in **index order** (the order the fields appear in the index definition), not the order they were listed in the projection or the order they are stored in the document. This matches MongoDB behavior for covered queries. A query that is answered by a collection scan instead returns fields in stored-document order.

Covered $group aggregations

A `$group` stage typically reads only a few fields (the group key and the inputs to its accumulators) but is often run without a leading `$match`. Without a covering index, Amazon DocumentDB reads and decodes every full document in the collection just to feed those few fields into the grouping.

Index-only scans for aggregation pipelines, including covered `$group` stages, are available in engine version 8.0.2 and later.

When a single index contains every field the pipeline prefix reads, Amazon DocumentDB can place an index-only scan beneath the group and read those fields directly from the index. Consider a pipeline that groups orders by status and sums their totals, with a compound index on `{ status: 1, total: 1 }`:

```
db.orders.createIndex({ status: 1, total: 1 })

db.orders.aggregate([
  { $group: { _id: "$status", revenue: { $sum: "$total" } } }
])
```

The pipeline reads only `status` and `total`, both of which are in the index, so the group is fed from an `IXONLYSCAN` instead of a collection scan:

```
// Without a covering index
$group  <-  COLLSCAN     (reads and decodes every full document)

// With a covering index on { status, total }
$group  <-  IXONLYSCAN over { status, total }   (reads only the referenced fields)
```

A leading `$match`, `$sort`, `$skip`, or `$limit` before the `$group` is supported. However, every field these stages reference (for example, a `$match` filter field or a `$sort` key) must also be present in the same index for the pipeline to be covered. If any referenced field is missing from the index, Amazon DocumentDB falls back to a collection or index scan.

> **Note**
>
> Unlike a `find` projection, a `$group` does not implicitly include the document `_id`. The `_id` in a `$group` is the group key, not the document identifier, so the index does not need to contain the document `_id` to cover the pipeline.

Requirements for a covered query

For Amazon DocumentDB to use an index-only scan, all of the following must hold:

- **A single index contains every field the query reads.** This includes projected fields for `find`, and the group key, accumulator inputs, and any filter or sort fields for a covered `$group` pipeline. If even one field is missing from the index, the query cannot be covered.
- **The projection is an inclusion of top-level fields** (for `find`). Exclusion projections, computed fields, and projection operators such as `$slice` and `$elemMatch` are not eligible.
- **The `_id` field is accounted for** (for `find`). Because `_id` is included by default, a common compound index such as `{ a: 1, b: 1 }` does *not* cover `find({}, { a: 1, b: 1 })`: the implicit `_id` is not in the index. Either exclude `_id` from the projection (`{ _id: 0, a: 1, b: 1 }`) or add `_id` to the index.
- **The filter is fully satisfied by the index.** If Amazon DocumentDB cannot turn part of the query predicate into an index bound, it keeps that part as a residual filter, reads the document to evaluate it, and does not convert the scan to an index-only scan. Operators such as `$mod`, `$ne`, `$nin`, and `$exists` leave a residual filter.

- **No projected field is a multikey (array) column.** See Restrictions.

Restrictions

The following cases cannot use an index-only scan and fall back to a normal index scan or collection scan. This is expected behavior and never affects the correctness of results.

- **Multikey (array) fields.** If a field has ever held an array value in any document, its index column is multikey. A multikey index stores one entry per array element, so the original array cannot be reconstructed from the index. Any query that reads such a field is not covered.
- **Dotted or nested paths.** A projection or field reference on a nested path such as `"address.city"` is not eligible, even if the index contains that path.
- **Exclusion, computed, or operator projections.** Projections such as `{ b: 0 }`, `{ total: { $add: ["$price", "$tax"] } }`, or `{ a: { $slice: 5 } }` require the full document and are not covered.
- **Whole-document references in aggregation.** A `$group` that references the entire document (for example, `{ $push: "$$ROOT" }`) cannot be covered, because no index can reconstruct the whole document.
- **Reshaping stages before `$group`.** A stage that reshapes documents (such as `$project`, `$addFields`, `$unwind`, or `$lookup`) appearing before the `$group` disqualifies the pipeline from being covered.
- **Collation mismatch on string predicates.** If the query's collation differs from the index's collation, string predicates cannot be served by the index and leave a residual filter, which forces a document fetch. Numeric and type-based predicates are unaffected.

Confirming an index-only scan

Use the `explain("executionStats")` output to confirm that a query is served by an index-only scan and that it is not touching documents. Look for two things:

- The winning plan uses the `IXONLYSCAN` stage.
- The `docsExamined` value on that stage is `0` (and `totalDocsExamined` is `0`), which means the query was answered entirely from the index without reading any documents.

```
db.orders.find(
  { customerId: 42 },
  { _id: 0, customerId: 1, status: 1, total: 1 }
).explain("executionStats").executionStats
```


```
{
    "executionSuccess": true,
    "nReturned": "2",
    "executionTimeMillis": "0.714",
    "totalDocsExamined": "0",
    "executionStages": {
        "stage": "IXONLYSCAN",
        "nReturned": "2",
        "indexName": "customerId_1_status_1_total_1",
        "direction": "forward",
        "docsExamined": "0"
    }
}
```


Note

A non-zero `docsExamined` value on an `IXONLYSCAN` stage means the index contained the data, but Amazon DocumentDB still checked storage to confirm that each document version was visible. This can happen when the collection has recent writes whose obsolete document versions garbage collection has not yet reclaimed. Once garbage collection processes those documents, `docsExamined` drops to `0`. At that point, the query realizes the full benefit of the covered scan. For more information about garbage collection, see Garbage collection in Amazon DocumentDB. For more information about the `docsExamined` metric, see Query plan analysis.

Best practices

- Design compound indexes that contain the fields your most frequent queries project or group on, so those queries can be covered.

- Exclude `_id` from `find` projections (`{ _id: 0, ... }`) unless `_id` is part of the covering index.
- Avoid indexing array (multikey) fields when you intend a query to be covered, because multikey columns disqualify the index-only path.
- Allow garbage collection to keep up with your write workload so that covered scans skip document access and `docsExamined` stays at `0`. For more information, see Garbage collection in Amazon DocumentDB.

Plan cache filters

A plan cache filter, also called an index filter, restricts the set of indexes that the query planner is allowed to consider for a specific query shape. When a query matches a shape that has a filter, the planner chooses a plan only from the indexes named in that filter instead of evaluating every index on the collection.

A query shape is the combination of the query predicate, the sort specification, and the collection namespace, with the predicate values normalized. Two queries that differ only in their values share a shape, so a single filter covers all of them. In the output of `planCacheListFilters`, normalized values are shown as `"@"`.

A plan cache filter gives you a way to pin the plan, so that you can avoid choosing a slower index because of data distribution or index changes. Filters have the following properties:

- **No application change is required.** Filters are set with a database command and applied on the server, so you can mitigate a regression without a code deployment. This is the main difference from a `hint`, which has to be added at every call site.
- **The scope is one query shape.** A shape is defined by the collection, the predicate, and the sort, not by the full query, so a filter constrains only the queries that match that shape. Other queries on the same collection continue to be planned normally.
- **The change is durable.** Filters persist across instance restarts and engine patching. Dropping the collection removes its filters.
- **The change is reversible.** Clearing the filter returns the shape to normal cost-based planning, so a filter is a low-risk way to stabilize a workload while you investigate the underlying cause.

Topics

- Supported commands

- Supported query shapes
- Setting, listing, and clearing filters
- Verifying that a filter is applied
- Examples
- Behavior with hints and index changes

Supported commands

Plan cache filters require planner version 2.0 or later. The command that a filter can be applied to depends on the planner version and the Amazon DocumentDB minor version, as shown in the following table.

Command support for plan cache filters

Command	Minimum planner version	Minimum minor version
`find`	2.0	5.0.0, 8.0.0
`count`	2.0	5.0.0, 8.0.0
`update`	2.0	5.0.2, 8.0.2
`delete`	2.0	5.0.2, 8.0.2
`findAndModify`	2.0	5.0.2, 8.0.2
`distinct`	3.0	8.0.0
`aggregate`	3.0	8.0.2

Note

Planner version 3.0 is available only in Amazon DocumentDB 8.0. Because the `distinct` and `aggregate` commands require planner version 3.0, plan cache filters for those two commands are available only in Amazon DocumentDB 8.0. In Amazon DocumentDB 5.0, which supports planner version 2.0, filters apply to the `find`, `count`, `update`, `delete`, and `findAndModify` commands.

For the `aggregate` command, the shape is taken from the front of the pipeline: the leading `$match` supplies the filter, and a `$sort` immediately after it supplies the sort. `$skip` and `$limit` do not affect the shape, and neither do the stages beyond that point.

Amazon DocumentDB optimizes the pipeline before planning it, and a plan cache filter is matched against the optimized pipeline. The filter and sort that the planner sees can therefore differ from the stages you wrote. For example, two adjacent `$match` stages at the front of a pipeline are combined into a single `$and` predicate:

```
db.orders.aggregate([
  { $match: { status: "SHIPPED" } },
  { $match: { region: "us-east-1" } },
  { $group: { _id: "$customerId", total: { $sum: "$amount" } } }
])
```

The planner sees one `$match` on `{$and: [{status: ...}, {region: ...}]}`, so the filter has to be set on that combined predicate.

```
db.runCommand({planCacheSetFilter: "orders",
query: { $and: [ { status: "SHIPPED" }, { region: "us-east-1" } ] },
indexes: [ "status_1_region_1" ]})
```

A filter set on the foreign collection of a `$lookup` or `$graphLookup` stage drives the index used to scan that foreign collection.

Supported query shapes

Starting with Amazon DocumentDB 5.0.2 and 8.0.2, the following operators can appear in a query shape that has a plan cache filter:

- `$text`. The search string is normalized, so queries that differ only in the terms they search for share one shape.
- `$near` and `$nearSphere`. Both operators normalize to the same shape. The coordinates, `$minDistance`, and `$maxDistance` are not part of the shape.
- `$geoWithin` and `$geoIntersects`. The GeoJSON geometry type is part of the shape, so a `Polygon` query and a `MultiPolygon` query are different shapes. The coordinates are not part of the shape.
- `$regex`, when used directly on a field. The pattern and options are normalized, so all regular expressions on the same field share one shape.

Setting a filter fails with error code 303 for a query shape that uses `$jsonSchema`, `$expr`, or `$sampleRate`. It also fails for the following shapes. A query that uses any of them is planned without a filter:

- A `$regex` nested inside `$in`, `$nin`, or `$all`. An array that mixes regular expressions with literal values cannot be evaluated against an index, so a filter on that shape could never be applied.
- A sort specification of `{$natural: 1}`. A `$natural` sort requests a collection scan, which is incompatible with restricting the planner to an index.

Setting, listing, and clearing filters

To set a plan cache filter, use the `planCacheSetFilter` command. The `query` and `sort` fields together specify the query shape, and `indexes` lists the index names that the planner is allowed to consider for that shape.

```
db.runCommand({
planCacheSetFilter: <collection>,
query: <query>,
sort: <sort>, // optional
indexes: [ <index1>, <index2>, ...],
comment: <any> // optional
})
```

For example, the following command restricts the planner to the a_1 index for the shape `{a: {$eq: "@"}, b: {$eq: "@"}}` with no sort:

```
db.runCommand({planCacheSetFilter: "foo",
query: { a: 1, b: 1 },
sort: {},
indexes: [ "a_1" ]})
```

Setting a filter for a shape that already has one replaces the existing filter. After running the following two commands, the filter on the shape `{a: {$eq: "@"}, b: {$eq: "@"}}` is `["b_1"]`:

```
db.runCommand({planCacheSetFilter: "foo",
query: { a: 1, b: 1 },
sort: {},
```

```
indexes: [ "a_1" ]})

db.runCommand({planCacheSetFilter: "foo",
query: { a: 6, b: 10 },
sort: {},
indexes: [ "b_1" ]})
```

To list every filter on a collection, use the `planCacheListFilters` command:

```
db.runCommand({planCacheListFilters: <collection>})
```

The output shows each stored shape with its normalized values and its allowed index list:

```
{
"filters" : [
{
"query" : { "a" : { "$eq" : "@" } },
"sort" : { },
"indexes" : [ "a_1" ]
},
{
"query" : { "a" : { "$gt" : "@" } },
"sort" : { "a" : 1 },
"indexes" : [ "a_1_b_1" ]
}
],
"ok" : 1
}
```

The operator is preserved and only the value is replaced with `"@"`. An implicit equality such as `{a: 1}` is listed in its explicit form, `{"a": {"$eq": "@"}}`, and the elements of an `$in` array are replaced as a whole, as `{"a": {"$in": "@"}}`. A filter set without a sort is listed with an empty `sort` document.

To remove filters, use the `planCacheClearFilters` command.

```
db.runCommand({
planCacheClearFilters: <collection>,
query: <query pattern>, // optional
sort: <sort specification>, // optional
comment: <any> // optional
```

```
})
```

To clear every filter on the collection, omit both `query` and `sort`:

```
db.runCommand({planCacheClearFilters: "foo"})
```

Supplying both `query` and `sort` clears only the filter that has that exact sort. The values in `query` are normalized away, so any representative value works. To clear only the filter that was set without a sort, pass an empty `sort` document:

```
db.runCommand({planCacheClearFilters: "foo", query: {a: 1}, sort: {a: 1}})

db.runCommand({planCacheClearFilters: "foo", query: {a: 1}, sort: {}})
```

Verifying that a filter is applied

The `explain` output includes two fields that report plan cache filter status:

- `indexFilterSet` is `true` when a filter exists on the collection for the query's shape.
- `indexFilterApplied` is `true` only when the query was planned using an index from that filter's list.

Starting with Amazon DocumentDB 5.0.2 and 8.0.2, these fields are reported for every command that supports plan cache filters, and in the profiler output for those commands. In earlier minor versions they were reported for the `find` command only. For more information about reading explain output, see Query plan analysis.

A filter can match a query shape without being applied. If none of the indexes in the filter's list can serve the query, the planner replans without the filter and chooses an index on cost. In that case `indexFilterSet` is `true` and `indexFilterApplied` is `false`. This happens when the named indexes do not cover the query's fields, when the named indexes do not exist, or when a `$regex` shape cannot produce index bounds, such as an unanchored or case-insensitive pattern.

Examples

The following examples use an `orders` collection, and a `customers` collection for the `$lookup` example, with these indexes:

```
db.orders.createIndex({ status: 1 })                          // status_1
db.orders.createIndex({ status: 1, orderDate: 1 })            // status_1_orderDate_1
db.orders.createIndex({ customerId: 1 })                      // customerId_1
db.customers.createIndex({ customerId: 1 })                   // customerId_1
```

Example: pinning a `find` query to a compound index

Consider a query that filters on `status` and sorts on `orderDate`:

```
db.orders.find({ status: "SHIPPED" }).sort({ orderDate: 1 })
```

The compound index `status_1_orderDate_1` satisfies both the predicate and the sort, so no separate sort is needed. If the planner instead chooses `status_1`, the query has to sort its results, which adds a `SORT` stage and gets more expensive as the number of matching orders grows. To restrict the planner to the compound index for this shape:

```
db.runCommand({planCacheSetFilter: "orders",
query: { status: "SHIPPED" },
sort: { orderDate: 1 },
indexes: [ "status_1_orderDate_1" ]})
```

Because values are normalized out of the shape, this one filter also covers `{ status: "PENDING" }`, `{ status: "CANCELLED" }`, and every other value of `status` with the same sort. It does not cover the same predicate with a different sort, or with no sort, which are separate shapes. Confirm the filter took effect with `explain`:

```
db.orders.find({ status: "SHIPPED" }).sort({ orderDate: 1 }).explain()
```

The output reports `indexFilterSet: true` and `indexFilterApplied: true`, and the `IXSCAN` stage names `status_1_orderDate_1`.

Example: pinning an `update` to a selective index

An update has to locate the documents to modify before it can write them, and that lookup uses an index in the same way a read does. When a predicate can be served by more than one index, the index the planner chooses determines how many documents the update examines.

```
db.orders.updateMany(
```

```
  { customerId: 4815, status: "PENDING" },
  { $set: { status: "CANCELLED" } }
)
```

Both `customerId_1` and `status_1` can serve this predicate. To pin the update to `customerId_1`:

```
db.runCommand({planCacheSetFilter: "orders",
query: { customerId: 4815, status: "PENDING" },
indexes: [ "customerId_1" ]})
```

Filters on write commands require Amazon DocumentDB 5.0.2 or 8.0.2 and planner version 2.0 or later. Verify with an `explain` of the update:

```
db.runCommand({explain: {update: "orders",
updates: [{ q: { customerId: 4815, status: "PENDING" },
             u: { $set: { status: "CANCELLED" } },
             multi: true }]}})
```

The winning plan is an `UPDATE` stage over an `IXSCAN` on `customerId_1`. The same filter applies to `delete` and `findAndModify` commands that use this query shape, because the shape does not include the type of write.

Example: pinning an `aggregate` pipeline and a `$lookup` foreign collection

For an aggregation, the shape is taken from the leading `$match` stage and a `$sort` immediately after it. Stages beyond that, such as `$group` or `$project`, are not part of the shape.

```
db.orders.aggregate([
  { $match: { status: "SHIPPED" } },
  { $group: { _id: "$customerId", total: { $sum: "$amount" } } }
])
```

A filter set on the `$match` predicate drives the index used to read `orders`:

```
db.runCommand({planCacheSetFilter: "orders",
query: { status: "SHIPPED" },
indexes: [ "status_1" ]})
```

Filters also apply to the collection that a `$lookup` or `$graphLookup` stage joins to. The filter is set on the foreign collection, not on the collection the pipeline runs against. In the following pipeline, `orders` is the outer collection and `customers` is the foreign collection:

```
db.orders.aggregate([
  { $lookup: { from: "customers",
               localField: "customerId",
               foreignField: "customerId",
               as: "customer" } }
])
```

The join probes `customers` once per outer document, so the index chosen there is applied repeatedly and its cost is multiplied across the pipeline. To pin that scan to a specific index, set the filter on `customers` for the join key's equality shape:

```
db.runCommand({planCacheSetFilter: "customers",
query: { customerId: 1 },
indexes: [ "customerId_1" ]})
```

Because values are normalized out of the shape, the value 1 above is a placeholder and any value produces the same filter. When a foreign-collection filter is applied, the top-level `indexFilterSet` and `indexFilterApplied` fields report `true`, and the index that the filter selected appears on the foreign side of the winning plan. Filters on `aggregate` require Amazon DocumentDB 8.0.2 and planner version 3.0.

Behavior with hints and index changes

- A plan cache filter takes precedence over a `hint`. Amazon DocumentDB applies the filter first and then applies the hint within the filter's allowed index list. If the hint names an index that the filter permits, that index is used. If the hint names an index that the filter does not permit, the hint is ignored and the planner chooses among the permitted indexes on cost. Both hint forms behave the same way, whether you name the index or give its key pattern.

  For example, if the collection `foo` has the indexes `a_1`, `b_1`, and `a_1_b_1`, and you set a filter on the shape `{query: {a: {$eq: "@"}, b: {$eq: "@"}}, sort: {a: 1}}` with the index list `["a_1", "a_1_b_1"]`, then running `db.foo.find({ a: 10, b: 20 }).sort({a: 1}).hint({ a: 1 })` uses `a_1`, because it is named in the hint and permitted by the filter.
- Dropping an index does not modify filters. The filter keeps the index name. While that index is missing, planning skips it. If another index in the filter's list can serve the query, the filter still

applies. Only when none of the named indexes are available does `indexFilterApplied` report `false`. Recreating an index with the same name makes it eligible again.

- Dropping a collection removes all of that collection's filters.

Best practices for Amazon DocumentDB

Learn best practices for working with Amazon DocumentDB (with MongoDB compatibility). This section is continually updated as new best practices are identified.

Topics

- Basic operational guidelines
- Instance sizing
- Working with indexes
- Security best practices
- Cost optimization
- Using metrics to identify performance issues
- TTL and time series workloads
- Migrations
- Working with cluster parameter groups
- Aggregation pipeline queries
- batchInsert and batchUpdate

Basic operational guidelines

The following are basic operational guidelines that everyone should follow when working with Amazon DocumentDB. The Amazon DocumentDB Service Level Agreement requires that you follow these guidelines.

- Deploy a cluster consisting of two or more Amazon DocumentDB instances in two AWS Availability Zones. For production workloads, deploy a cluster consisting of three or more Amazon DocumentDB instances in three Availability Zones.
- Use the service within the stated service limits. For more information, see Amazon DocumentDB quotas.

- Monitor your memory, CPU, connections, and storage usage. To help you maintain system performance and availability, set up Amazon CloudWatch to notify you when usage patterns change or when you approach the capacity of your deployment.
- Scale up your instances when you are approaching capacity limits. Your instances should be provisioned with enough compute resources (i.e., RAM, CPU) to accommodate unforeseen increases in demand from your applications.
- Set your backup retention period to align with your recovery point objective.
- Test failover for your cluster to understand how long the process takes for your use case. For more information, see Amazon DocumentDB failover.
- Connect to your Amazon DocumentDB cluster with the cluster endpoint (see Amazon DocumentDB endpoints) and in replica set mode (see Connecting to Amazon DocumentDB as a replica set) to minimize the impact of a failover on your application.
- Choose a driver read preference setting that maximizes read scaling while meeting your application's read consistency requirements. The `secondaryPreferred` read preference enables replica reads and frees up the primary instance to do more work. For more information, see Read preference options.
- Design your application to be resilient in the event of network and database errors. Use your driver's error mechanism to distinguish between transient errors and persistent errors. Retry transient errors using an exponential backoff mechanism when appropriate. Ensure that your application considers data consistency when implementing retry logic.
- Enable cluster deletion protection for all production clusters, or any cluster that has valuable data. Before deleting an Amazon DocumentDB cluster, take a final snapshot. If you are deploying resources with CloudFormation, enable termination protection. For more information, see Termination protection and deletion protection.
- When you create an Amazon DocumentDB cluster, `--engine-version` is an optional parameter. If you omit it, Amazon DocumentDB uses the newest 8.0 minor version available in the AWS Region. To use a specific minor version, specify it with `--engine-version`. For more information, see Amazon DocumentDB cluster settings.

Instance sizing

One of the most critical aspects of choosing an instance size in Amazon DocumentDB is the amount of RAM for your cache. Amazon DocumentDB reserves one-third of the RAM for its own services, meaning that only two-thirds of the instance RAM is available for the cache. Thus, it is an Amazon DocumentDB best practice to choose an instance type with enough RAM to fit your

working set (i.e., data and indexes) in memory. Having properly sized instances will help optimize for overall performance and potentially minimize I/O cost. You can use the third party Amazon DocumentDB sizing calculator to estimate the instance size for a particular workload.

To determine whether your application's working set fits in memory, monitor the `BufferCacheHitRatio` using Amazon CloudWatch for each instance in a cluster that is under load.

The `BufferCacheHitRatio` CloudWatch metric measures the percentage of data and indexes served from an instance's memory cache (versus the storage volume). Generally speaking, the value of `BufferCacheHitRatio` should be as high as possible, as reading data from working set memory is faster and more cost-effective than reading from the storage volume. While it is desirable to keep `BufferCacheHitRatio` as close to 100% as possible, the best achievable value will depend on your application's access patterns and performance requirements. To maintain the highest possible `BufferCacheHitRatio`, it is recommended that the instances in your cluster are provisioned with enough RAM to be able to fit your indexes and working data set in memory.

If your indexes do not fit into memory, you will see a lower `BufferCacheHitRatio`. Continually reading from disk incurs additional I/O costs and is not better than reading from memory. If your `BufferCacheHitRatio` ratio is lower than expected, scale up the instance size for your cluster to provide more RAM to fit working set data in memory. If scaling up the instance class results in a dramatic increase in `BufferCacheHitRatio`, then your application's working set did not fit in memory. Continue to scale up until `BufferCacheHitRatio` no longer increases dramatically after a scaling operation. For information about monitoring an instance's metrics, see Amazon DocumentDB metrics.

Depending on your workload and latency requirements, it may be acceptable for your application to have higher `BufferCacheHitRatio` values during steady state usage, but have the `BufferCacheHitRatio` dip periodically as analytic queries that need to scan an entire collection are run on an instance. These periodic dips in `BufferCacheHitRatio` may manifest as higher latency for subsequent queries that need to repopulate the working set data from the storage volume back into the buffer cache. **Test your workloads in a pre-production environment with a representative production workload first to understand the performance characteristics and `BufferCacheHitRatio` before deploying the workload to production.**

The `BufferCacheHitRatio` is an instance-specific metric, so different instances within the same cluster may have different `BufferCacheHitRatio` values depending on how reads are distributed among the primary and replica instances. If your operational workload cannot handle periodic increases in latency from repopulating the working set cache after running analytic queries, you

should try to isolate the regular workload's buffer cache from that of the analytic queries. You can achieve complete `BufferCacheHitRatio` isolation by directing operational queries to the primary instance and analytic queries only to the replica instances. You can also achieve partial isolation by directing analytic queries to a specific replica instance with the understanding that some percentage of regular queries will also run on that replica and could potentially be affected.

Appropriate `BufferCacheHitRatio` values depend on your use case and application requirements. There is no one best or minimum value for this metric; only you can decide if the tradeoff from a temporarily lower `BufferCacheHitRatio` is acceptable from a cost and performance perspective.

Working with indexes

Building indexes

When importing data into Amazon DocumentDB, you should create your indexes before importing large datasets. You can use the Amazon DocumentDB Index Tool to extract indexes from a running MongoDB instance or `mongodump` directory, and create those indexes in an Amazon DocumentDB cluster. For more guidance on migrations, see Migrating and upgrading Amazon DocumentDB.

Index selectivity

Limit index creation to fields where the number of duplicate values is less than 1% of the total number of documents in the collection. As an example, if your collection contains 100,000 documents, only create indexes on fields where the same value occurs 1000 times or fewer.

Choosing an index with a high number of unique values (i.e., a high cardinality) ensures that filter operations return a small number of documents, thereby yielding good performance during index scans. An example of a high-cardinality index is a unique index, which guarantees that equality predicates return at most a single document. Examples of low-cardinality include an index over a Boolean field and an index over day of the week. Due to their poor performance, low cardinality indexes are unlikely to be chosen by the database's query optimizer. At the same time, low cardinality indexes continue to consume resources such as disk space and I/Os. As a rule of thumb, you should target indexes on fields where the typical value frequency is 1% of the total collection size or less.

Additionally, it is recommended to only create indexes on fields that are commonly utilized as a filter and regularly look for unused indexes. For more information, see How do I analyze index usage and identify unused indexes?.

Impact of indexes on writing data

While indexes can improve query performance by avoiding the need to scan every document in a collection, this improvement comes with a tradeoff. For each index on a collection, every time a document is inserted, updated, or deleted, the database must update the collection and write the fields to each of the indexes for the collection. For example, if a collection has nine indexes, the database must perform ten writes before acknowledging the operation to the client. Thus, each additional index incurs additional write latency, I/O's, and increase in the overall utilized storage.

Cluster instances need to be appropriately sized to keep all working set memory. This avoids the need to continuously read index pages from the storage volume, which negatively impacts performance and generates higher I/O costs. For more information, see Instance sizing.

For best performance, minimize the number of indexes in your collections, adding only those indexes necessary to improve performance for common queries. While workloads vary, a good guideline is to keep the number of indexes per collection to five or fewer.

Identifying missing indexes

As a best practice, identify missing indexes on a regular basis. For more information, see How do I identify missing indexes?.

Identifying unused indexes

As a best practice, identify and remove unused indexes on a regular basis. For more information, see How do I analyze index usage and identify unused indexes?.

Security best practices

For security best practices for Amazon DocumentDB, see Security best practices for Amazon DocumentDB.

Cost optimization

The following best practices can help you manage and minimize your costs when using Amazon DocumentDB. For pricing information, see Amazon DocumentDB (with MongoDB compatibility) pricing and Amazon DocumentDB (with MongoDB compatibility) FAQs.

- Create billing alerts at thresholds of 50 percent and 75 percent of your expected bill for the month. For more information about creating billing alerts, see Creating a Billing Alarm.

- Amazon DocumentDB's architecture separates storage and compute, so even a single-instance cluster is highly durable. The cluster storage volume replicates data six ways across three Availability Zones, providing extremely high durability regardless of the number of instances in the cluster. A typical production cluster has three or more instances to provide high availability. However, you can optimize costs by using a single instance development cluster when high availability is not required.
- For development and test scenarios, stop a cluster when it is no longer needed and start the cluster when development resumes. For more information, see Stopping and starting an Amazon DocumentDB cluster.
- Both TTL and change streams incur I/O's when data is written, read, and deleted. If you have enabled these features but are not utilizing them in your application, disabling the features can help reduce costs.

Using metrics to identify performance issues

Topics

- Viewing performance metrics
- Setting a CloudWatch alarm
- Evaluating performance metrics
- Evaluating Amazon DocumentDB instance usage with CloudWatch metrics
- Tuning queries

To identify performance issues caused by insufficient resources and other common bottlenecks, you can monitor the metrics available for your Amazon DocumentDB cluster.

Viewing performance metrics

Monitor performance metrics on a regular basis to see the average, maximum, and minimum values for a variety of time ranges. This helps you identify when performance is degraded. You can also set Amazon CloudWatch alarms for particular metric thresholds so that you are alerted if they are reached.

To troubleshoot performance issues, it's important to understand the baseline performance of the system. After you set up a new cluster and get it running with a typical workload, capture the average, maximum, and minimum values of all the performance metrics at different intervals (for

example, 1 hour, 24 hours, 1 week, 2 weeks). This gives you an idea of what is normal. It helps to get comparisons for both peak and off-peak hours of operation. You can then use this information to identify when performance is dropping below standard levels.

You can view performance metrics using the AWS Management Console or AWS CLI. For more information, see Viewing CloudWatch data.

Setting a CloudWatch alarm

To set a CloudWatch alarm, see Using Amazon CloudWatch Alarms in the *Amazon CloudWatch User Guide*.

Evaluating performance metrics

An instance has several different categories of metrics. How you determine acceptable values depends on the metric.

CPU

- **CPU Utilization** — The percentage of the computer processing capacity used.

  An instance shows some CPU utilization even when it has no connections or workload. This baseline comes from processes that Amazon DocumentDB runs continuously on every instance, such as monitoring, health checks, and instance management. The baseline is a larger share of capacity on smaller instance classes. When you set CPU alarms, measure the idle baseline for your instance class and watch for increases above it.

Memory

- **Freeable Memory** — How much RAM is available on the instance.
- **Swap Usage** — How much swap space is used by the instance, in megabytes.

Input/output operations

- **Read IOPS**, **Write IOPS** — The average number of disk read or write operations per second.
- **Read Latency**, **Write Latency** — The average time for a read or write operation in milliseconds.
- **Read Throughput**, **Write Throughput** — The average number of megabytes read from or written to disk per second.

- **Disk Queue Depth** — The number of I/O operations that are waiting to be written to or read from disk.

Network traffic

- **Network Receive Throughput**, **Network Transmit Throughput** — The rate of network traffic to and from the instance in megabytes per second.

Database connections

- **DB Connections** — The number of client sessions that are connected to the instance.

Generally speaking, acceptable values for performance metrics depend on what your baseline looks like and what your application is doing. Investigate consistent or trending variances from your baseline.

The following are recommendations and advice about specific types of metrics:

- **High CPU consumption** — High values for CPU consumption might be appropriate, provided that they are in keeping with your goals for your application (like throughput or concurrency) and are expected. If your CPU consumption is consistently over 80 percent, consider scaling up your instances.
- **High RAM consumption** — If your `FreeableMemory` metric frequently dips below 10% of the total instance memory, consider scaling up your instances. For more information on what happens when your DocumentDB instance is experiencing high memory pressure, see Amazon DocumentDB Resource Governance.
- **Swap usage** — This metric should remain at or near zero. If your swap usage is significant, consider scaling up your instances.
- **Network traffic** — For network traffic, talk with your system administrator to understand what the expected throughput is for your domain network and internet connection. Investigate network traffic if throughput is consistently lower than expected.
- **Database connections** — Consider constraining database connections if you see high numbers of user connections together with decreases in instance performance and response time. The best number of user connections for your instance varies based on your instance class and the complexity of the operations being performed. For issues with any performance metrics, one of

the first things you can do to improve performance is tune the most used and most expensive queries to see if that lowers the pressure on system resources.

If your queries are tuned and an issue persists, consider upgrading your Amazon DocumentDB instance class to one with more of the resource (CPU, RAM, disk space, network bandwidth, I/O capacity) that is related to the issue you're experiencing.

Evaluating Amazon DocumentDB instance usage with CloudWatch metrics

You can use CloudWatch metrics to watch your instance throughput and discover if your instance class provides sufficient resources for your applications. For information about your instance class quotas, see Instance quotas and locate the specifications for your instance class to find your network performance.

If your instance usage is near the instance class limit, then performance may begin to slow. The CloudWatch metrics can confirm this situation so you can plan to manually scale-up to a larger instance class.

Combine the following CloudWatch metrics values to find out if you are nearing the instance class limit:

- **NetworkThroughput**—The amount of network throughput received and transmitted by the clients for each instance in the Amazon DocumentDB cluster. This throughput value doesn't include network traffic between instances in the cluster and the cluster storage volume.
- **StorageNetworkThroughput**—The amount of network throughput received and sent to the Amazon DocumentDB cluster storage volume by each instance in the Amazon DocumentDB cluster.

Add the **NetworkThroughput** to the **StorageNetworkThroughput** to find the network throughput received from and sent to the Amazon DocumentDB cluster storage volume by each instance in your Amazon DocumentDB cluster. The instance class limit for your instance should be greater than the sum of these two combined metrics.

You can use the following metrics to review additional details of the network traffic from your client applications when sending and receiving:

- **NetworkReceiveThroughput**—The amount of network throughput received from clients by each instance in the Amazon DocumentDB cluster. This throughput doesn't include network traffic between instances in the cluster and the cluster storage volume.
- **NetworkTransmitThroughput**—The amount of network throughput sent to clients by each instance in the Amazon DocumentDB cluster. This throughput doesn't include network traffic between instances in the cluster and the cluster storage volume.
- **StorageNetworkReceiveThroughput**—The amount of network throughput received from the Amazon DocumentDB cluster storage volume by each instance in the cluster.
- **StorageNetworkTransmitThroughput**—The amount of network throughput sent to the Amazon DocumentDB cluster storage volume by each instance in the cluster.

Add all of these metrics together to evaluate how your network usage compares to the instance class limit. The instance class limit should be greater than the sum of these combined metrics.

The network limits and CPU utilization by an instance are mutual. When the network throughput increases, then the CPU utilization also increases. Monitoring the CPU and network usage provides information about how and why the resources are being exhausted.

To help minimize network usage, you can consider:

- Using a larger instance class.
- Dividing the write requests in batches to reduce overall transactions.
- Directing the read-only workload to a read-only instance.
- Deleting any unused indexes.

Tuning queries

One of the best ways to improve cluster performance is to tune your most commonly used and most resource-intensive queries to make them less expensive to run.

You can use the profiler (see Profiling Amazon DocumentDB operations) to log the execution time and details of operations that were performed on your cluster. Profiler is useful for monitoring the slowest operations on your cluster to help you improve individual query performance and overall cluster performance.

You can also use the `explain` command to learn how to analyze a query plan for a particular query. Use this information to modify a query or underlying collection to improve your query performance (for example, adding an index).

TTL and time series workloads

Document deletion resulting from TTL index expiry is a best effort process. Documents are not guaranteed to be deleted within any specific period. Factors like instance size, instance resource utilization, document size, overall throughput, the number of indexes, and whether indexes and the working set fit in memory can all affect the timing of when expired documents are deleted by the TTL process.

When the TTL monitor deletes your documents, each deletion incurs I/O costs, which increases your bill. If throughput and TTL delete rates increase, you should expect a higher bill due to increased I/O usage. However, if you do not create a TTL index to delete documents, but instead segment documents into collections based on time and simply drop those collections when they are no longer needed, you will not incur any IO costs. This can be significantly more cost effective than using a TTL index.

For time-series workloads, you can consider creating rolling collections instead of a TTL index as rolling collections can be a better way to delete data and less I/O intensive. If you have large collections (especially collections over 1TB) or TTL deletion I/O costs are a concern, partition documents into collections based on time, and drop collections when the documents are no longer needed. You can create one collection per day or one per week, depending on your data ingest rate. While requirements will vary depending on your application, a good rule of thumb is to have more smaller collections rather than a few large collections. Dropping these collections does not incur I/O costs, and can be faster and more cost effective than using a TTL index.

Migrations

As a best practice for migrating data to Amazon DocumentDB, first create your indexes in Amazon DocumentDB and then migrate your data. Creating the indexes first can reduce the overall time and increase the speed of the migration. To do this, you can use the Amazon DocumentDB Index Tool. For more information on migrations, see the Migrating and upgrading Amazon DocumentDB.

We also recommend that before you migrate your production database, it is a best practice to fully test your application on Amazon DocumentDB, taking into consideration functionality, performance, operations, and cost.

Working with cluster parameter groups

Try out cluster parameter group changes on a test cluster before applying the changes to your production clusters. For information about backing up your cluster, see Backing up and restoring in Amazon DocumentDB.

Aggregation pipeline queries

When creating an aggregation pipeline query with multiple stages and evaluating only a subset of the data in the query, use the `$match` stage as the first stage or in the beginning of the pipeline. Using `$match` first will reduce the number of documents subsequent stages within the aggregation pipeline query will need to process, thus improving the performance of your query.

`batchInsert` and `batchUpdate`

When performing a high rate of concurrent `batchInsert` and/or `batchUpdate` operations, and the amount of `FreeableMemory` (CloudWatch Metric) goes to zero on your primary instance, you can either reduce the concurrency of the batch insert or update workload or, if concurrency of the workload cannot be reduced, increase the instance size to increase the amount of `FreeableMemory`.

Managing Amazon DocumentDB indexes

Topics

- Amazon DocumentDB index creation
- Maintaining Amazon DocumentDB indexes

Amazon DocumentDB index creation

Building indexes in Amazon DocumentDB requires a number of decisions to be made:

- How quickly does it need to be completed?
- Can the collection be inaccessible while the build is occurring?
- How much of an instance's compute power can be allocated to the build?
- What type of index should be created?

This section helps you answer these questions and provides the commands and monitoring examples to create an Amazon DocumentDB index on your instance-based cluster collection.

Guidelines

The following guidelines include basic limits and configuration tradeoffs when creating new indexes:

- **Amazon DocumentDB version support** - While single worker indexing is supported on all Amazon DocumentDB versions, multiple worker indexing is supported only on Amazon DocumentDB versions 4.0, 5.0, and 8.0.
- **Performance trade-off** - Increasing the number of workers in the index creation process increases CPU utilization and read IO on the primary instance of your Amazon DocumentDB database. The resources needed to create a new index will not be available to your running workload.
- **Elastic clusters** - Parallel indexing is not supported on Amazon DocumentDB elastic clusters.
- **Serverless clusters** - Parallel indexing on serverless clusters requires a minimum capacity of at least 32 DCUs. Below this threshold, the `workers` parameter is accepted but the index build executes with a single worker. The maximum number of parallel workers scales with your configured minimum capacity: approximately `min_DCU / 8` workers are available. For example, at 32 DCUs you can use up to 4 workers, and at 64 DCUs you can use up to 8 workers.

  > **Note**
  >
  > A cluster restart is required after changing the minimum capacity setting for the new parallel worker limit to take effect.

- **Maximum workers (instance-based)** - The maximum number of workers you can configure depends on the size of your primary instance in your database cluster. It is half the total number of vCPUs on the primary instance of your database cluster. For example, you can run a maximum of 32 workers on a db.r6g.16xlarge instance that has 64 vCPUs.

  > **Note**
  >
  > Parallel workers are only supported on instances 2xlarge and above.

- **Minimum workers** - The minimum number of workers you can configure is one. The default setting for index creation on instance-based clusters is two workers. However, you can reduce the

number of workers to one by using the “worker threads” option. This will run the process with a single worker.

- **Index compression** - Amazon DocumentDB doesn't support index compression. Data sizes for indexes might be larger than when you use other options.
- **Indexing multiple collections** - Half the vCPUs on your database cluster's primary instance can be used for configured workers performing index creation on multiple collections.
- **Index types** - For a complete explanation of supported index types on Amazon DocumentDB, see How to index on Amazon DocumentDB on the AWS Database Blog.

Getting started

To start index creation on a collection, use the `createIndexes` command. By default, the command will run two parallel workers that increases the speed of the index creation process by two times.

For example, the following command process demonstrates how to create an index for the “user_name” field in a document and increase the indexing process speed to four workers:

1. Create indexes using two parallel workers on the cluster:

```
db.runCommand({"createIndexes":"test","indexes":[{"key": {"user_name":1},
 "name":"username_idx"}]})
```

2. To optimize the speed of the index creation process, you can specify the number of workers by using the “worker threads” option (`"workers":<number>`) in the `db.runCommand createIndexes` command.

   Increase the speed of the process to four parallel workers:

```
db.runCommand({"createIndexes":"test","indexes":[{"key": {"user_name":1},
 "name":"username_idx", "workers":4}]})
```

> **Note**
>
> The higher the number of workers, the faster the index creation progresses. However, the higher the number of workers increases, the higher the load increases on the

vCPUs and read IO of your primary instance. Ensure that your cluster is sufficiently provisioned to handle the increased burden without degrading other workloads.

Indexing progress status

The index creation process works by initializing, scanning collections, sorting keys, and, finally, inserting keys by way of an index builder. The process has up to six stages when you run it in the foreground, and up to nine stages when you run it in the background. You can view status metrics such as percentage completion, total number of scanned storage blocks, sorted keys, and inserted keys on stage by stage basis.

Monitor the progress on the indexing process by using the `db.currentOp()` command in the mongo shell. A 100% completion of the last stage shows that all the indexes have been successfully created:

```
db.currentOp({"command.createIndexes": { $exists : true } })
```

Note

Viewing the indexing progress status is only supported on Amazon DocumentDB 5.0.

Index build types

The four types of index builds are:

- **Foreground** — The foreground index build blocks all other database operations until the index is created. The Amazon DocumentDB foreground build is comprised of five stages.
- **Foreground (unique)** — Single document (unique) foreground index builds block other database operations like regular foreground builds. Unlike the basic foreground build, the unique build uses an additional stage (sorting keys 2) to look for duplicate keys. The foreground (unique) build is comprised of six stages.
- **Background** — The background index build allows other database operations to run in the foreground while the index is being created. The Amazon DocumentDB background build is comprised of eight stages.
- **Background (unique)** — Single document (unique) background index builds allow other database operations to run in the foreground while the index is being created. Unlike the basic

background build, the unique build uses an additional stage (sorting keys 2) to look for duplicate keys. The background (unique) build is comprised of nine stages.

Index build stages

Stage	Foreground	Foreground (unique)	Background	Background (unique)
Initializing	1	1	1	1
building index: initializing	2	2	2	2
building index: scanning collection	3	3	3	3
building index: sorting keys 1	4	4	4	4
building index: sorting keys 2		5		5
building index: inserting keys	5	6	5	6
validating: scanning index			6	7
validating: sorting tuples			7	8
validating: scanning collection			8	9

- **initializing** - createIndex is preparing the index builder. This phase should be very brief.

- **building index: initializing** - The index builder is preparing to create the index. This phase should be very brief.
- **building index: scanning collection** - The index builder is performing a collection scan to collect index keys. The unit of measure is “blocks”.

> **Note**
>
> If more than one worker is configured for the index build, it is displayed in this stage. The “scanning collection” stage is the only stage that uses multiple workers during the index build process. All other stages will display a single worker.

- **building index: sorting keys 1** - The index builder is sorting the collected index keys. The unit of measure is “keys”.
- **building index: sorting keys 2** - The index builder is sorting the collected index keys that correspond to dead tuples. This phase only exists for unique index building. The unit of measure is “keys”.
- **building index: inserting keys** - The index builder is inserting index keys into the new index. The unit of measure is “keys”.
- **validating: scanning index** - createIndex is scanning the index to find keys that need to be validated. The unit of measure is “blocks”.
- **validating: sorting tuples** - createIndex is sorting the output of the index scanning phase.
- **validating: scanning collection** - createIndex is scanning the collection to validate the index keys found in the previous two phases. The unit of measure is “blocks”.

Index build output example

In the following output example (foreground index build), the status of the index creation is shown. The “msg” field summarizes the build progress by indicating the stage and the completion percentage of the build. The “workers” field indicates the number of workers used during that stage of the index build. The “progress” field shows the actual numbers used to calculate the percentage of completion.

> **Note**
>
> The “currentIndexBuildName”, “msg”, and “progress” fields are not supported on Amazon DocumentDB version 4.0.

```
{
    "inprog" : [{
    …
        "command": {
            "createIndexes": "test",
            "indexes": [{
                "v": 2,
                "key": {
                    "user_name": 1
                },
                "name": "user_name_1"
            }],
            "lsid": {
                "id": UUID(“094d0fba-8f41-4373-82c3-7c4c7b5ff13b”)
            },
            "$db": "test"
        },
        "currentIndexBuildName": user_name_1,
        "msg": "Index Build: building index number_1, stage 6/6 building index:
 656860/1003520 (keys) 65%",
        "workers": 1,
        "progress": {
            "done": 656861,
            "total": 1003520
        },
    …
    ],

    "ok" : 1
}
```

Maintaining Amazon DocumentDB indexes

Topics

- Index bloat
- Index maintenance using reIndex

Index bloat

Amazon DocumentDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent transactions. When documents are deleted or updated, their previous versions remain in collections

and indexes as "dead" versions. The garbage collection process automatically reclaims space from these dead versions for future operations.

Index bloat occurs when a collection's indexes become larger due to the accumulation of dead or obsolete index entries or fragmentation within the pages. The percentage reported represents the amount of index space that can be used by future index entries. This bloat consumes space in both the buffer cache and storage. If you want to remove the bloat, you will need to rebuild indexes.

Example

Run the following command to determine unused storage for your index:

```
db.coll.aggregate({$indexStats:{}});
```

The result looks similar to this:

```
{
    "name" : "_id_",
    "key" : {
        "_id" : 1
    },
    "host" : "devbox-test.localhost.a2z.com:27317",
    "size" : NumberLong(827392),
    "accesses" : {
        "ops" : NumberLong(40000),
        "docsRead" : NumberLong(46049),
        "since" : ISODate("2025-04-03T21:44:51.251Z")
    },
    "cacheStats" : {
        "blksRead" : NumberLong(264),
        "blksHit" : NumberLong(140190),
        "hitRatio" : 99.8121
    },
    "unusedStorageSize" : {
        "unusedSizeBytes" : 409600,
        "unusedSizePercent" : 49.51
    }
}
```

You can rebuild indexes without downtime using the `reIndex` command, which requires a scan of the entire collection. See Index maintenance using `reIndex`.

Index maintenance using `reIndex`

`reIndex` is a command used to rebuild an index. It is typically used when an index has become corrupted or inefficient. Over time, indexes can accumulate unused space due to many updates, inserts, or deletes, leading to degraded performance. Reindexing helps to remove such unused space and restore the efficiency of the index.

`reIndex` guidelines

- `reIndex` is only supported on Amazon DocumentDB 5.0.
- Amazon DocumentDB supports `reindex` of a single index in the background, allowing for multiple workers. The old index is usable by queries when the `reIndex` process is running.
- Amazon DocumentDB supports indexing progress report through `currentOp`. You can see index build stages similar to the Index build stages viewed during index creation. The only difference is that `reIndex` always has eight stages, regardless if it's unique or not. There's no "building index: sorting keys 2" stage.
- `reIndex` can run concurrently with any command except index-related commands on the same collection: `createIndexes`, `dropIndexes`, `collMod`, and `renameCollection`.
- `reIndex` is currently not supported for text, geospatial, vector, and partial indexes.

`reIndex` build

Use the following command to rebuild your index:

```
db.runCommand({ reIndex: "collection-name", index: "index-name"})
```

Optionally, you can also control the number of workers assigned to the rebuild process:

```
db.runCommand({ reIndex: "collection-name", index: "index-name", workers: number })
```

Managing collection-level document compression

Amazon DocumentDB 5.0 collection-level document compression allows you to lower storage and IO costs by compressing the documents in your collections. You can enable document compression at a collection level and view compression metrics as needed by measuring the storage gains

through compression metrics such as storage size of compressed documents and compression status. Amazon DocumentDB 5.0 uses the LZ4 compression algorithm to compress documents.

Amazon DocumentDB supports document compression starting with version 5.0. The following are collection-level document compression functions:

- **Default behavior** — The default compression setting for new collections on a 5.0 cluster is determined by the cluster parameter `default_collection_compression`. This parameter is set to “disabled” by default.
- **Compressing existing collections** — The compression setting for existing collections can be changed using the `collMod` command.
- **Changing the compression threshold** — The default compression threshold is 2KB. This value can be specified for new collections using the `createCollection` command, and changed for existing collections using `collMod` command.

> **Note**
>
> Amazon DocumentDB document compression is not supported on Amazon DocumentDB versions 3.6 and 4.0.

Topics

- Managing document compression
- Monitoring document compression

Managing document compression

Enabling document compression in a collection

Enable document compression while creating a collection on Amazon DocumentDB 5.0 by using `db.createCollection()` method:

```
db.createCollection( sample_collection,{
    storageEngine : {
        documentDB: {
            compression:{enable: <true | false>}
```

```
        }
    }
})
```

Enabling document compression in a cluster

Document compression can be enabled by default for all new collections on a cluster level by setting the cluster parameter `default_collection_compression` to “enabled”. When this parameter is set to “enabled”, newly created collections on the cluster will have compression enabled by default with a compression threshold of 2 KB.

Compressing existing collections

You can also modify the compression settings for an existing collection using the `collMod` operation and specifying the following `storageEngine` configuration. Note that the change made using this command will only apply to newly inserted/updated documents and the compression on previously inserted documents will not change.

```
db.runCommand({
    collMod: "orders",
    storageEngine: {
        documentDB: {compression: {enable: <true | false>} }
    }
})
```

Setting the compression thresholds

By default, the compression threshold for compressed collections is 2032 bytes. This threshold value can be set in the `createCollection` command when creating a new collection with compression enabled:

```
db.createCollection( sample_collection, {
    storageEngine : {
        documentDB: {
            compression: {
                enable: true,
                threshold: <128 - 8000>
            }
        }
    }
```

```
})
```

You can also modify the compression threshold for an existing compressed collection using the `collMod` operation and specifying the following `storageEngine` configuration:

```
db.runCommand({
    collMod: "orders",
    storageEngine: {
        documentDB: {
            compression: {
                enable: true,
                threshold: <128 - 8000>
            }
        }
    }
})
```

Note that the compression threshold can only be set to a value between 128 to 8000 bytes. Additionally, the `enable` option needs to be set to “true” when specifying the compression threshold.

Monitoring document compression

You can check if a collection is compressed and calculate it's compression ratio as follows.

View compression statistics by running the `db.printCollectionStats()` or `db.collection.stats()` command from the mongo shell. The output shows you the original size and compressed size that you can compare to analyze the storage gains from document compression. In this example, statistics for a collection named “sample_collection” are shown below. A scaling factor of 1024*1024 is used below to output the `size` and `storageSize` values in MB.

```
db.sample_collection.stats(1024*1024)
```

The following is an example of the output for the above command:

```
{
    "ns" : "test.sample_collection",
    "count" : 1000000,
    "size" : 3906.3,
    "avgObjSize" : 4096,
```

```
    "storageSize" : 1953.1,
    compression:{"enabled" : true,"threshold" : 2032},
    ...
}
```

- **size** - The original size of the document collection.
- **avgObjSize** - The average document size before compression rounded off to first decimal. The unit of measure is bytes.
- **storageSize** - The storage size of the collection after compression. The unit of measure is bytes.
- **enabled** - Indicates if compression is enabled or disabled.

To calculate the actual compression ratio, divide the collection size by the storage size (size/storageSize). For the example above, the calculation is 3906.3/1953.1 which translates to a 2:1 compression ratio.

Managing dictionary-based compression in Amazon DocumentDB 8.0

Amazon DocumentDB 8.0 introduces a new document compression algorithm, Zstd, as an improved alternative to LZ4. When you enable dictionary compression on an Amazon DocumentDB 8.0 collection by selecting Zstd as the compression algorithm, documents from your collection are sampled to create a custom compression dictionary. New and updated documents are compressed using this dictionary and the Zstd algorithm. This approach typically achieves higher compression ratios than standard compression methods, especially for collections with consistent document schemas or repeated field names.

LZ4 is an algorithm designed for fast compression and decompression. It is known to be light on the CPU while achieving noticeable compression. Zstd is a general-purpose algorithm that with default settings uses more CPU but achieves better compression ratios than LZ4. The usage of dictionaries improves compression even further for most JSON documents. Some benefits of the Zstd algorithm are:

- **Reduced Storage Costs** — Higher compression ratios mean less storage usage and lower costs.
- **Lower I/O** — Compressed documents require less I/O, potentially improving performance.
- **Optimized for your collection** — The dictionary is trained specifically for your collection's data patterns.

> **Note**
>
> Dictionary-based compression is not supported on Amazon DocumentDB versions 3.6, 4.0, and 5.0.

Performance considerations

Zstd compression involves the following trade-offs:

- **Storage versus CPU** — Zstd compression achieves better compression ratios but may use slightly more CPU resources compared to LZ4 compression.
- **Initial Compression** — New collections might not achieve optimal compression until enough documents are inserted to train an effective dictionary. Currently, Amazon DocumentDB trains a dictionary when a collection has at least 100 documents.
- **Workload Type** — Read-intensive workloads where the entire data fits into the buffer cache may experience an increase in latency and CPU usage due to decompression overhead.

Zstd compression is particularly effective for collections with small documents, document arrays and repeated field names.

Enabling dictionary-based compression

For new collections you can use the below command to enable Zstd compression:

```
db.createCollection("myCollection",
    {
        storageEngine: {
            documentDB: {
                compression: {
                    enable: true,
                    algorithm: "zstd"
                }
            }
        }
    }
 )
```

You can also enable or modify compression on an existing collection:

```
db.runCommand({
    collMod: "myCollection",
    storageEngine: {
        documentDB: {
            compression: {
                enable: true,
                algorithm: "zstd"
            }
        }
    }
 })
```

To enable the Zstd algorithm for all new collections in your cluster, set the `default_collection_compression` parameter in your cluster parameter group to `zstd`.

Getting Started

In Amazon DocumentDB 8.0, new collections use Zstd compression by default. To turn off compression for new collections, set the `default_collection_compression` parameter in your cluster parameter group to `disabled`.

Starting with Amazon DocumentDB 8.0, `enabled` is no longer a valid value for `default_collection_compression`. The valid values are `zstd`, `lz4`, and `disabled`.

Monitoring

You can view compression information for a collection using one of the following commands:

- db.runCommand({ collStats: "myCollection" }) OR
- db.collection.stats()

These commands return key statistics that you can use to compute compression ratio:

- **compression.algorithm** — The algorithm used (`lz4` or `zstd`)
- **storageSize** — The actual storage used by the collection, after compression. Note that this number includes fragmentation (that is, the unused space in database pages)

- **avgObjSize** — The average logical size of the collection's documents, decompressed. Note that if your collection has more than 20k documents, this number will be an approximation based on a sample of 20k documents.
- **size** — The logical size of the collection without compression. This number is obtained by multiplying avgObjSize by the total number of documents in the collection, so if avgObjSize is an approximation, this number will also be an approximation.
- **count** — Number of documents in the collection

The following CloudWatch metrics can be helpful while evaluating dictionary-based compression:

- CPUUtilization
- FreeableMemory
- VolumeBytesUsed
- VolumeReadIOPs
- VolumeWriteIOPs

collStats metrics:

- storageSize
- size

In addition, it can be useful to keep track of metrics specific to your application, like latency and throughput per query type or API.

Compressor statistics

Starting with Amazon DocumentDB 8.0.3, the `collStats` command and the `db.collection.stats()` method return a `compressor_stats` object inside the `compression` field. It reports how much data the compression algorithm has processed for the collection, and works for both Zstd and LZ4.

Use compressor statistics to see how effectively compression reduces the size of the data that you write to a collection. For example, you can use them to compare Zstd and LZ4 on your workload, or to evaluate a change to the compression threshold.

To view compressor statistics for a collection, run the following command:

```
db.runCommand({ collStats: "myCollection" })
```

For a collection that uses Zstd, the output includes:

```
"compression": {
    "enable": true,
    "threshold": 128,
    "algorithm": "zstd",
    "usingDictionary": true,
    "compressor_stats": {
        "input_bytes": 10000000,
        "output_bytes": 2500000,
        "compression_operations": 10000,
        "compression_ratio": 4
    }
}
```

For a collection that uses LZ4, the output doesn't include `usingDictionary`:

```
"compression": {
    "enable": true,
    "threshold": 128,
    "algorithm": "lz4",
    "compressor_stats": {
        "input_bytes": 10000000,
        "output_bytes": 2500000,
        "compression_operations": 10000,
        "compression_ratio": 4
    }
}
```

The following table describes the fields in the `compression` object.

Field	Description
`enable`	Specifies whether compression is enabled for the collection.
`threshold`	The minimum document size for compression. The compressor doesn't compress documents smaller than this value.
`algorithm`	The compression algorithm used by the collection (`zstd` or `lz4`).
`usingDictionary`	Specifies whether the collection uses a trained compression dictionary. Returned only for Zstd.
`compressor_stats.input_bytes`	Total uncompressed size, in bytes, of the data processed by the compressor.
`compressor_stats.output_bytes`	Total size, in bytes, of the data after compression.
`compressor_stats.compression_operations`	Number of documents that Amazon DocumentDB attempted to compress. This includes documents that meet the compression `threshold` but that Amazon DocumentDB stores uncompressed because compression doesn't reduce their size enough.
`compressor_stats.compression_ratio`	`input_bytes` divided by `output_bytes`. Higher values mean better compression. If the compressor hasn't processed any documents, the ratio is 1, which indicates no compression.

In these examples, compression reduces 10,000 documents totaling 10 MB to 2.5 MB, a 4:1 compression ratio.

How compressor statistics accumulate

Compressor statistics are cumulative totals of compression operations, and they don't represent the collection's current data size. The following table describes how operations on the collection affect these statistics.

Operation	Counted in statistics?	Details
Updating a document	Yes, on every update	If you update a document 10 times, the statistics include it 10 times.
Deleting a document	Not subtracted	The compressor doesn't subtract deleted documents from `input_bytes` or `output_bytes` .
Writing small documents	No	The compressor skips documents smaller than the compression `threshold` .
Writing documents before compression is enabled or before the statistics were last reset	Not until updated	The compressor counts these documents when you update them.
Writing documents while compression is disabled	No	`collStats` doesn't return `compressor_stats` while compression is disabled. Disabling compression doesn't reset the statistics. When you re-enable compression, the compressor adds new values to the previous totals.

Choosing the right metric

Compressor statistics and collection size statistics measure different things, so their ratios usually don't match. `compression_ratio` covers only data that the compressor processed. `size/storageSize` covers the whole collection, including uncompressed documents and page fragmentation.

The following table shows which metric to use, depending on your goal.

Goal	Use
See how well recent writes compress, for example after switching from LZ4 to Zstd	`compressor_stats.compression_ratio`
See the collection's overall storage savings	`size/storageSize`

Security in Amazon DocumentDB

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. This documentation helps you understand how to apply the shared responsibility model when using Amazon DocumentDB. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** — AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to Amazon DocumentDB (with MongoDB compatibility), see AWS Services in Scope by Compliance Program.
- **Security in the cloud** — Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

Amazon DocumentDB is authorized under Federal Risk and Authorization Management Program (FedRAMP). It has FedRAMP High authorization for AWS GovCloud (US) Regions and FedRAMP Moderate authorization for AWS US East/West Regions. For details about AWS and compliance efforts, see AWS Services in Scope by Compliance Program.

> **Note**
>
> This chapter applies to both instance-based clusters and Elastic Clusters. For more information, see the topics below.

You also learn how to use other AWS services that help you monitor and secure your Amazon DocumentDB resources. The following topics show you how to configure Amazon DocumentDB to meet your security and compliance objectives.

Topics

- Password management with Amazon DocumentDB and AWS Secrets Manager
- Data protection in Amazon DocumentDB

- Identity and Access Management for Amazon DocumentDB
- Cross-service confused deputy prevention
- Authentication using IAM identity
- Authentication using SCRAM-SHA-256
- Managing Amazon DocumentDB users
- Database access using Role-Based Access Control
- Updating your Amazon DocumentDB TLS certificates
- Compliance validation in Amazon DocumentDB
- Resilience in Amazon DocumentDB
- Infrastructure security in Amazon DocumentDB
- Amazon DocumentDB API and interface VPC endpoints (AWS PrivateLink)
- Security best practices for Amazon DocumentDB
- Auditing Amazon DocumentDB events
- Amazon VPC and Amazon DocumentDB

Password management with Amazon DocumentDB and AWS Secrets Manager

Amazon DocumentDB integrates with Secrets Manager to manage primary user passwords for your clusters.

Topics

- Limitations for Secrets Manager integration with Amazon DocumentDB
- Overview of managing primary user passwords with AWS Secrets Manager
- Enforcing Amazon DocumentDB management of the primary user password in AWS Secrets Manager
- Managing the primary user password for a cluster with Secrets Manager

Limitations for Secrets Manager integration with Amazon DocumentDB

Managing primary user passwords with Secrets Manager isn't supported for the following features:

- Clusters that are part of an Amazon DocumentDB global database

- Amazon DocumentDB cross-Region read replicas

Overview of managing primary user passwords with AWS Secrets Manager

With AWS Secrets Manager, you can replace hard-coded credentials in your code, including database passwords, with an API call to Secrets Manager to retrieve the secret programmatically. For more information about Secrets Manager, see AWS Secrets Manager User Guide.

When you store database secrets in Secrets Manager, your AWS account incurs charges. For information about pricing, see AWS Secrets Manager pricing.

You can specify that Amazon DocumentDB manages the primary user password in Secrets Manager for an Amazon DocumentDB cluster when you perform one of the following operations:

- Create the cluster
- Modify the cluster

When you specify that Amazon DocumentDB manages the primary user password in Secrets Manager, Amazon DocumentDB generates the password and stores it in Secrets Manager. You can interact directly with the secret to retrieve the credentials for the primary user. You can also specify a customer managed key to encrypt the secret, or use the KMS key that Secrets Manager provides.

Amazon DocumentDB manages the settings for the secret and rotates the secret every seven days by default. You can modify some of the settings, such as the rotation schedule. If you delete a cluster that manages a secret in Secrets Manager, the secret and its associated metadata are also deleted.

To connect to a cluster with the credentials in a secret, you can retrieve the secret from Secrets Manager. For more information, see Get secrets from AWS Secrets Manager and Connect to a SQL database using JDBC with credentials in an AWS Secrets Manager secret in the *AWS Secrets Manager User Guide*.

Enforcing Amazon DocumentDB management of the primary user password in AWS Secrets Manager

You can use IAM condition keys to enforce Amazon DocumentDB management of the primary user password in AWS Secrets Manager. The following policy doesn't allow users to create or restore

instances or clusters unless the primary user password is managed by Amazon DocumentDB in Secrets Manager.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "rds:CreateDBCluster"
            ],
            "Resource": "*",
            "Condition": {
                "Bool": {
                    "rds:ManageMasterUserPassword": false
                }
            }
        }
    ]
}
```

Managing the primary user password for a cluster with Secrets Manager

You can configure Amazon DocumentDB management of the primary user password in Secrets Manager when you perform the following actions:

- Creating an Amazon DocumentDB cluster
- Modifying an Amazon DocumentDB cluster

You can use the Amazon DocumentDB console or the AWS CLI to perform these actions.

Data protection in Amazon DocumentDB

The AWS shared responsibility model applies to data protection in Amazon DocumentDB. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this

infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Amazon DocumentDB or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Topics

- Client-side field level encryption
- Encrypting Amazon DocumentDB data at rest
- Encrypting data in transit
- Key management

Client-side field level encryption

Amazon DocumentDB client-side field level encryption (FLE) allows you to encrypt sensitive data in your client applications before it is transferred to a Amazon DocumentDB cluster. Sensitive data remains encrypted when it is stored and processed in a cluster and is decrypted at the client application when retrieved.

Topics

- Getting started
- Querying in client-side FLE

Getting started

The initial configuration of client-side FLE in Amazon DocumentDB is a four-step process that includes creating an encryption key, associating a role to the application, configuring the application, and defining CRUD operation with encryption options.

Topics

- Step 1: Create the encryption keys
- Step 2: Associate a role with the application
- Step 3: Configure the application
- Step 4: Define a CRUD operation
- Example: client-side field level encryption configuration file

Step 1: Create the encryption keys

Using AWS Key Management Service, create a symmetric customer managed key that encrypts and decrypts the sensitive data field, and grant it the necessary IAM usage permissions. AWS KMS stores the customer managed key, which is used to encrypt data keys. Storing the customer managed key in AWS KMS strengthens your security posture. The data key is the secondary key, which is stored in an Amazon DocumentDB collection and is required to encrypt sensitive fields before storing the document in Amazon DocumentDB. The customer managed key encrypts the data key, which in turn encrypts and decrypts your data. If you use a global cluster, you can create a multi-Region key that different service roles can use in different Regions.

For more information about the AWS Key Management Service, including how to create a key, see AWS Key Management Service concepts in the *AWS Key Management Service Developer Guide*.

Step 2: Associate a role with the application

Create an IAM policy with appropriate AWS KMS permissions. This policy allows the IAM identities to which it is attached to encrypt and decrypt the KMS key specified in the resource field. Your application assumes this IAM role to authenticate with AWS KMS.

The policy should look similar to this:

```
{ "Effect": "Allow",
"Action": ["kms:Decrypt", "kms:Encrypt"],
"Resource": "Customer Key ARN"
}
```

Step 3: Configure the application

You have defined a customer managed key in AWS KMS and created an IAM role with the necessary permissions to access it. Import the required packages.

```
import boto3
import json
import base64
from pymongo import MongoClient
from pymongo.encryption import (Algorithm,
                                ClientEncryption)
```

```
# create a session object:
my_session = boto3.session.Session()

# get access_key and secret_key programmatically using get_frozen_credentials() method:
current_credentials = my_session.get_credentials().get_frozen_credentials()
```

1. Specify aws as the AWS KMS provider type and enter the account credentials that you retrieved in the previous step.

   ```
   provider = "aws"
   kms_providers = {
       provider: {
   ```

```
        "accessKeyId": current_credentials.access_key,
        "secretAccessKey": current_credentials.secret_key
    }
}
```

2. Specify the customer managed key to encrypt the data key:

```
customer_key = {
“region”: “AWS region of the customer_key”,
    “key”: “customer_key ARN”
}

key_vault_namespace = "encryption.dataKeys"

key_alt_name = 'TEST_DATA_KEY'
```

3. Configure the MongoClient object:

```
client = MongoClient(connection_string)

coll = client.test.coll
coll.drop()

client_encryption = ClientEncryption(
    kms_providers, # pass in the kms_providers variable from the previous step
    key_vault_namespace = key_vault_namespace,
    client,
    coll.codec_options
)
```

4. Generate your data key:

```
data_key_id = client_encryption.create_data_key(provider,
    customer_key,
    key_alt_name = [key_alt_name])
```

5. Retrieve your existing data key:

```
data_key = DataKey("aws",
    master_key = customer_key)
key_id = data_key["_id"]
data_key_id = client[key_vault_namespace].find_one({"_id": key_id})
```

Step 4: Define a CRUD operation

Define the CRUD operation with encryption options.

1. Define the collection to write/read/delete a single document:

```
coll = client.gameinfo.users
```

2. Explicit Encryption - encrypt fields and insert:

> **Note**
>
> Exactly one of "key_id" or "key_alt_name" must be provided.

```
encrypted_first_name = client_encryption.encrypt(
    "Jane",
    Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
    key_id=data_key_id
)
encrypted_last_name = client_encryption.encrypt(
    "Doe",
    Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
    key_id=data_key_id
)
encrypted_dob = client_encryption.encrypt(
    "1990-01-01",
    Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Random,
    key_id=data_key_id
)

coll.insert_one(
    {"gamerTag": "jane_doe90",
    "firstName": encrypted_first_name,
    "lastName": encrypted_last_name,
    "dateOfBirth":encrypted_dob,
    "Favorite_games":["Halo","Age of Empires 2","Medal of Honor"]
})
```

Example: client-side field level encryption configuration file

In the following example, replace each *placeholder* with your own information.

```
# import python packages:
import boto3
import json
import base64
from pymongo import MongoClient
from pymongo.encryption import (Algorithm,
                                ClientEncryption)

def main():

    # create a session object:
    my_session = boto3.session.Session()

    # get aws_region from session object:
    aws_region = my_session.region_name

    # get access_key and secret_key programmatically using get_frozen_credentials()
 method:
    current_credentials = my_session.get_credentials().get_frozen_credentials()
    provider = "aws"

    # define the kms_providers which is later used to create the Data Key:
    kms_providers = {
        provider: {
            "accessKeyId": current_credentials.access_key,
            "secretAccessKey": current_credentials.secret_key
        }
    }

    # enter the kms key ARN. Replace the example ARN value.
    kms_arn = "arn:aws:kms:us-east-1:123456789:key/abcd-efgh-ijkl-mnop"
    customer_key = {
        "region": aws_region,
        "key":kms_arn
    }

    # secrets manager is used to store and retrieve user credentials for connecting to
 an Amazon DocumentDB cluster.
    # retrieve the secret using the secret name. Replace the example secret key.
    secret_name = "/dev/secretKey"
```

```
    docdb_credentials = json.loads(my_session.client(service_name = 'secretsmanager',
 region_name = "us-east-1").get_secret_value(SecretId = secret_name)['SecretString'])

    connection_params = '/?tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false'
    conn_str = 'mongodb://' + docdb_credentials["username"] + ':' +
 docdb_credentials["password"] + '@' + docdb_credentials["host"] + ':' +
 str(docdb_credentials["port"]) + connection_params
    client = MongoClient(conn_str)

    coll = client.test.coll
    coll.drop()

    # store the encryption data keys in a key vault collection (having naming
 convention as db.collection):
    key_vault_namespace = "encryption.dataKeys"
    key_vault_db_name, key_vault_coll_name = key_vault_namespace.split(".", 1)

    # set up the key vault (key_vault_namespace) for this example:
    key_vault = client[key_vault_db_name][key_vault_coll_name]
    key_vault.drop()
    key_vault.create_index("keyAltNames", unique=True)

    client_encryption = ClientEncryption(
        kms_providers,
        key_vault_namespace,
        client,
        coll.codec_options)

    # create a new data key for the encrypted field:
    data_key_id = client_encryption.create_data_key(provider, master_key=customer_key,
 key_alt_names=["some_key_alt_name"], key_material = None)

    # explicitly encrypt a field:
    encrypted_first_name = client_encryption.encrypt(
    "Jane",
    Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
    key_id=data_key_id
    )
    coll.insert_one(
    {"gamerTag": "jane_doe90",
    "firstName": encrypted_first_name
    })
    doc = coll.find_one()
```

```
    print('Encrypted document: %s' % (doc,))

    # explicitly decrypt the field:
    doc["firstName"] = client_encryption.decrypt(doc["firstName"])
    print('Decrypted document: %s' % (doc,))

    # cleanup resources:
    client_encryption.close()
    client.close()

if __name__ == "__main__":
    main()
```

Querying in client-side FLE

Amazon DocumentDB only supports equality queries with client-side FLE encrypted values.

For example, to query for documents where encrypted gamerscore equals 500 the client uses an explicit encryption method to encrypt the query value:

```
encrypted_gamerscore_filter = client_encryption.encrypt(
    500,
    Algorithm.AEAD_AES_256_CBC_HMAC_SHA_512_Deterministic,
    key_id=data_key_id
)

coll.find( {
    "gamerscore" : { "$eq" : encrypted_gamerscore_filter }
} )
```

Encrypting Amazon DocumentDB data at rest

> **Note**
>
> AWS KMS is replacing the term *customer master key (CMK)* with *AWS KMS key* and *KMS key*. The concept has not changed. To prevent breaking changes, AWS KMS is keeping some variations of this term.

You encrypt data at rest in your Amazon DocumentDB cluster by specifying the storage encryption option when you create your cluster. Storage encryption is enabled cluster-wide and is applied to

all instances, including the primary instance and any replicas. It is also applied to your cluster's storage volume, data, indexes, logs, automated backups, and snapshots.

Amazon DocumentDB uses the 256-bit Advanced Encryption Standard (AES-256) to encrypt your data using encryption keys stored in AWS Key Management Service (AWS KMS). When using an Amazon DocumentDB cluster with encryption at rest enabled, you don't need to modify your application logic or client connection. Amazon DocumentDB handles encryption and decryption of your data transparently, with minimal impact on performance.

Amazon DocumentDB integrates with AWS KMS and uses a method known as envelope encryption to protect your data. When an Amazon DocumentDB cluster is encrypted with an AWS KMS, Amazon DocumentDB asks AWS KMS to use your KMS key to generate a ciphertext data key to encrypt the storage volume. The ciphertext data key is encrypted using the KMS key that you define, and is stored along with the encrypted data and storage metadata. When Amazon DocumentDB needs to access your encrypted data, it requests AWS KMS to decrypt the ciphertext data key using your KMS key and caches the plaintext data key in memory to efficiently encrypt and decrypt data in the storage volume.

The storage encryption facility in Amazon DocumentDB is available for all supported instance sizes and in all AWS Regions where Amazon DocumentDB is available.

Enabling encryption at rest for an Amazon DocumentDB cluster

You can enable or disable encryption at rest on an Amazon DocumentDB cluster when the cluster is provisioned using either the AWS Management Console or the AWS Command Line Interface (AWS CLI). Clusters that you create using the console have encryption at rest enabled by default. Clusters that you create using the AWS CLI have encryption at rest disabled by default. Therefore, you must explicitly enable encryption at rest using the `--storage-encrypted` parameter. In either case, after the cluster is created, you can't change the encryption at rest option.

Amazon DocumentDB uses AWS KMS to retrieve and manage encryption keys, and to define the policies that control how these keys can be used. If you don't specify an AWS KMS key identifier, Amazon DocumentDB uses the default AWS managed service KMS key. Amazon DocumentDB creates a separate KMS key for each AWS Region in your AWS account. For more information, see AWS Key Management Service Concepts.

To get started on creating your own KMS key, see Getting Started in the *AWS Key Management Service Developer Guide*.

Important

You must use a symmetric encryption KMS key to encrypt your cluster as Amazon DocumentDB supports only symmetric encryption KMS keys. Do not use an asymmetric KMS key to attempt to encrypt the data in your Amazon DocumentDB clusters. For more information, see Asymmetric keys in AWS KMS in the *AWS Key Management Service Developer Guide*.

If Amazon DocumentDB can no longer access the KMS key for a cluster — for example, when the AWS account that owns the key is suspended, the key is disabled, the key is scheduled for deletion, or the key policy or grant that Amazon DocumentDB relies on is removed — the cluster first transitions to the `inaccessible-encryption-credentials-recoverable` status. While the cluster is in this status, Amazon DocumentDB stops the cluster's instances and you can't read from or write to the cluster, but the cluster can still be recovered if access to the KMS key is restored within 7 days. If access is not restored within 7 days, the cluster transitions to the terminal `inaccessible-encryption-credentials` status. From the terminal status, the cluster is no longer available and the current state of the database can't be recovered — you can only restore from a backup or perform a point-in-time restore using the original KMS key. For Amazon DocumentDB, backups are always enabled for at least 1 day.

Note

Clusters that are part of a global cluster behave differently. When Amazon DocumentDB detects that it can no longer access the KMS key, all clusters in the global cluster transition directly to the terminal `inaccessible-encryption-credentials` status, skipping the recoverable status. This is because a cluster that is part of a global cluster can be stopped and started only when it is the only cluster in the global cluster. To recover, you must restore from a snapshot or perform a point-in-time restore. To delete the original clusters, you must first remove each cluster from the global cluster and then delete it.

> **Important**
>
> You cannot change the KMS key for an encrypted cluster after you have already created it. Be sure to determine your encryption key requirements before you create your encrypted cluster.

Using the AWS Management Console

You specify the encryption at rest option when you create a cluster. Encryption at rest is enabled by default when you create a cluster using the AWS Management Console. It can't be changed after the cluster is created.

To specify the encryption at rest option when creating your cluster

1. Create an Amazon DocumentDB cluster as described in the Getting Started section. However, in step 6, do not choose **Create cluster**.
2. Under the **Authentication** section, choose **Show advanced settings**.
3. Scroll down to the **Encryption-at-rest** section.
4. Choose the option that you want for encryption at rest. Whichever option you choose, you can't change it after the cluster is created.
   - To encrypt data at rest in this cluster, choose **Enable encryption**.
   - If you don't want to encrypt data at rest in this cluster, choose **Disable encryption**.
5. Choose the primary key that you want. Amazon DocumentDB uses the AWS Key Management Service (AWS KMS) to retrieve and manage encryption keys, and to define the policies that control how these keys can be used. If you don't specify an AWS KMS key identifier, Amazon DocumentDB uses the default AWS managed service KMS key. For more information, see AWS Key Management Service Concepts.

   > **Note**
   >
   > After you create an encrypted cluster, you can't change the KMS key for that cluster. Be sure to determine your encryption key requirements before you create your encrypted cluster.

6. Complete the other sections as needed, and create your cluster.

Using the AWS CLI

To encrypt an Amazon DocumentDB cluster using the AWS CLI, run the `create-db-cluster` command and specify the `--storage-encrypted` option. Amazon DocumentDB clusters created using the AWS CLI do not enable storage encryption by default.

The following example creates an Amazon DocumentDB cluster with storage encryption enabled.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

Example

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
  --db-cluster-identifier mydocdbcluster \
  --port 27017 \
  --engine docdb \
  --master-username SampleUser1 \
  --master-user-password primaryPassword \
  --storage-encrypted
```

For Windows:

```
aws docdb create-db-cluster ^
  --db-cluster-identifier SampleUser1 ^
  --port 27017 ^
  --engine docdb ^
  --master-username SampleUser1 ^
  --master-user-password primaryPassword ^
  --storage-encrypted
```

When you create an encrypted Amazon DocumentDB cluster, you can specify an AWS KMS key identifier, as in the following example.

Example

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
```

```
  --db-cluster-identifier SampleUser1 \
  --port 27017 \
  --engine docdb \
  --master-username primaryUsername \
  --master-user-password yourPrimaryPassword \
  --storage-encrypted \
  --kms-key-id key-arn-or-alias
```

For Windows:

```
aws docdb create-db-cluster ^
  --db-cluster-identifier SampleUser1 ^
  --port 27017 ^
  --engine docdb ^
  --master-username SampleUser1 ^
  --master-user-password primaryPassword ^
  --storage-encrypted ^
  --kms-key-id key-arn-or-alias
```

> **Note**
>
> After you create an encrypted cluster, you can't change the KMS key for that cluster. Be sure to determine your encryption key requirements before you create your encrypted cluster.

Resolving an Amazon DocumentDB cluster in an inaccessible encryption state

An Amazon DocumentDB cluster moves to an inaccessible encryption status when Amazon DocumentDB can't access the KMS key that the cluster was encrypted with. There are two such statuses, and the recovery path depends on which one the cluster is in.

`inaccessible-encryption-credentials-recoverable` status

While the cluster is in the `inaccessible-encryption-credentials-recoverable` status, you can return the cluster to `available` by restoring Amazon DocumentDB's access to the KMS key and then starting the cluster. To resolve this status, do the following:

1. Confirm that the AWS account that owns the KMS key is active. If the account is suspended, reactivate it.

2. Confirm that the KMS key is enabled. For more information, see Enabling and disabling keys in the *AWS Key Management Service Developer Guide*.
3. Check whether the KMS key is scheduled for deletion. If it is, cancel the scheduled key deletion. For more information, see Scheduling and canceling key deletion in the *AWS Key Management Service Developer Guide*.
4. Confirm that the KMS key policy and any grants that Amazon DocumentDB relies on still permit Amazon DocumentDB to use the key.
5. After access to the KMS key is restored, start the cluster by using the AWS Management Console or by running the `start-db-cluster` AWS CLI command.

   Example

   For Linux, macOS, or Unix:

   ```
   aws docdb start-db-cluster \
     --db-cluster-identifier example-cluster
   ```

   For Windows:

   ```
   aws docdb start-db-cluster ^
     --db-cluster-identifier example-cluster
   ```

> **Important**
>
> If access to the KMS key is not restored within 7 days, the cluster transitions to the terminal `inaccessible-encryption-credentials` status, from which it can't be started.

`inaccessible-encryption-credentials` status

The `inaccessible-encryption-credentials` status is terminal. The cluster can't be started, and the running state of the database can't be recovered. To recover your data, restore from a snapshot or perform a point-in-time restore to a new cluster. You must still have access to the original KMS key to perform the restore. If the KMS key was deleted, the data can't be recovered.

For more information, see Restoring from a cluster snapshot and Restoring to a point in time.

> **Note**
>
> Clusters that are part of a global cluster transition directly to the `inaccessible-encryption-credentials` status when access to the KMS key is lost, because a cluster that is part of a global cluster can be stopped and started only when it is the only cluster in the global cluster. To delete a cluster that is in this status, first remove each cluster from the global cluster, and then delete the clusters individually.

If you can't delete a cluster that is in the `inaccessible-encryption-credentials` status because deletion protection is enabled, turn off deletion protection by using the AWS CLI before retrying the delete.

Example

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
  --db-cluster-identifier example-cluster \
  --no-deletion-protection
```

For Windows:

```
aws docdb modify-db-cluster ^
  --db-cluster-identifier example-cluster ^
  --no-deletion-protection
```

You can then delete the cluster by using the `delete-db-cluster` command.

Example

For Linux, macOS, or Unix:

```
aws docdb delete-db-cluster \
  --db-cluster-identifier example-cluster \
  --skip-final-snapshot
```

For Windows:

```
aws docdb delete-db-cluster ^
  --db-cluster-identifier example-cluster ^
```

```
  --skip-final-snapshot
```

If the cluster does not delete after you run the preceding commands, contact AWS Support.

Limitations for Amazon DocumentDB encrypted clusters

The following limitations exist for Amazon DocumentDB encrypted clusters.

- You can enable or disable encryption at rest for an Amazon DocumentDB cluster only at the time that it is created, not after the cluster has been created. However, you can create an encrypted copy of an unencrypted cluster by creating a snapshot of the unencrypted cluster, and then restoring the unencrypted snapshot as a new cluster while specifying the encryption at rest option.

  For more information, see the following topics:

  - Creating a manual cluster snapshot
  - Restoring from a cluster snapshot
  - Copying Amazon DocumentDB cluster snapshots
- Amazon DocumentDB clusters with storage encryption enabled can't be modified to disable encryption.
- All instances, automated backups, snapshots, and indexes in an Amazon DocumentDB cluster are encrypted with the same KMS key.

Encrypting data in transit

You can use Transport Layer Security (TLS) to encrypt the connection between your application and an Amazon DocumentDB cluster. By default, encryption in transit is enabled for newly created Amazon DocumentDB clusters. It can optionally be disabled when the cluster is created, or at a later time. When encryption in transit is enabled, secure connections using TLS are required to connect to the cluster. For more information connecting to Amazon DocumentDB using TLS, see Connecting programmatically to Amazon DocumentDB.

Amazon DocumentDB post-quantum key exchange

With Amazon DocumentDB 8.0.3 and later, you can use hybrid post-quantum key exchange for TLS 1.3 connections on instance-based clusters. Post-quantum key exchange helps protect your data in transit. It protects against attackers who record encrypted traffic now and decrypt it later with a quantum computer.

Hybrid post-quantum key exchange combines the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) with a classical elliptic curve key exchange. Your cluster supports the following hybrid key exchange groups, from most to least preferred:

- `X25519MLKEM768`
- `SecP256r1MLKEM768`
- `SecP384r1MLKEM1024`

Your cluster uses post-quantum key exchange by default for all `tls` parameter values except `disabled`. You don't need to change your cluster configuration. The supported TLS versions, cipher suites, and certificates don't change. Engine versions earlier than 8.0.3 use only classical key exchange. To use post-quantum key exchange, upgrade your cluster to Amazon DocumentDB 8.0.3 or later.

A connection uses post-quantum key exchange only when the client uses TLS 1.3 and its TLS library supports ML-KEM. Other connections use classical key exchange. The following list shows the minimum requirements for each language or driver:

C, C++, PHP, and Ruby

A driver that is linked with OpenSSL 3.5 or later

Go

Go 1.24 or later

Java

Java Development Kit (JDK) 27 or later, or the Bouncy Castle Java Secure Socket Extension (JSSE) provider

.NET

.NET on Linux with OpenSSL 3.5 or later

Node.js

Node.js 24.5 or later

Python

Python that is built with OpenSSL 3.5 or later

Rust

The mongodb crate with the `rustls-tls-aws-lc` feature

A *key share* is the public key that a client sends for a key exchange group. If a client supports ML-KEM but doesn't send an ML-KEM key share, the cluster sends a `HelloRetryRequest` message that asks the client to retry with an ML-KEM key share. The connection then uses post-quantum key exchange, but the handshake takes one extra round trip, so it takes about as long as a TLS 1.2 handshake. To avoid the retry, configure your client to prefer an ML-KEM group, so that it sends an ML-KEM key share in its first message.

Important

A TLS ClientHello message that offers ML-KEM is about 1.2–1.5 KB, which is larger than a classical ClientHello message. The message can span more than one network packet. Some older firewalls and TLS-inspecting proxies reset these connections. If a client fails after the upgrade but connects with only classical key exchange groups, a firewall or proxy is the likely cause. To fix a connection that fails for this reason, do one of the following:

- Update your firewall or proxy to support larger ClientHello messages.
- Configure the client to offer only classical key exchange groups.
- If you can't change the firewall, proxy, or client, contact Support.

To monitor post-quantum key exchange, use the `TLSHandshakesPostQuantum` and `TLSHandshakesClassical` Amazon CloudWatch metrics. For more information about these metrics, see Amazon DocumentDB metrics.

To check the key exchange of one connection, you need to have OpenSSL 3.5 or later and the Amazon DocumentDB certificate authority (CA) bundle. For instructions on downloading the CA bundle, see Connecting with TLS enabled.

The following OpenSSL command shows how to connect to your cluster and offer only the `X25519MLKEM768` group. Replace *`sample-cluster.node.aa-example-1.docdb.amazonaws.com`* with your cluster endpoint, and run the command in the directory that contains `global-bundle.pem`:

```
openssl s_client -connect sample-cluster.node.aa-example-1.docdb.amazonaws.com:27017 \
```

```
  -groups X25519MLKEM768 \
  -CAfile global-bundle.pem
```

If the connection uses post-quantum key exchange, the output includes the following line:

```
Negotiated TLS1.3 group: X25519MLKEM768
```

Managing Amazon DocumentDB cluster TLS settings

Encryption in transit for an Amazon DocumentDB cluster is managed via the TLS parameter in a cluster parameter group. You can manage your Amazon DocumentDB cluster TLS settings using the AWS Management Console or the AWS Command Line Interface (AWS CLI). See the following sections to learn how to verify and modify your current TLS settings.

Using the AWS Management Console

Follow these steps to perform management tasks for TLS encryption using the console—such as identifying parameter groups, verifying the TLS value, and making needed modifications.

> **Note**
>
> Unless you specify differently when you create a cluster, your cluster is created with the default cluster parameter group. The parameters in the `default` cluster parameter group can't be modified (for example, `tls` enabled/disabled). So if your cluster is using a `default` cluster parameter group, you need to modify the cluster to use a non-default cluster parameter group. First, you might need to create a custom cluster parameter group. For more information, see Creating Amazon DocumentDB cluster parameter groups.

1. **Determine the cluster parameter group that your cluster is using.**

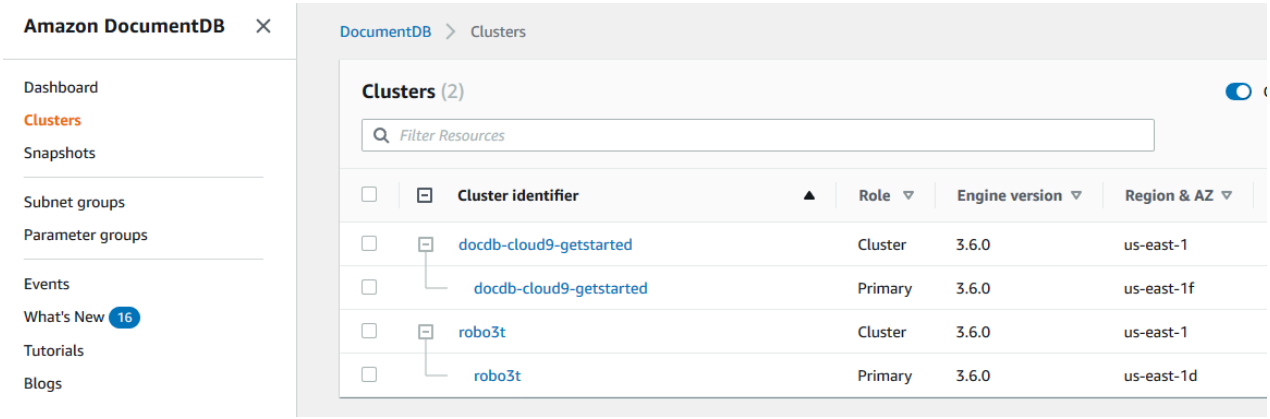
   a. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

   b. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon

(≡) in the upper-left corner of the page.

c. Note that in the **Clusters** navigation box, the column **Cluster Identifier** shows both clusters and instances. Instances are listed underneath clusters. See the following screenshot for reference.



d. Choose the cluster that you're interested in.

e. Choose the **Configuration** tab and scroll down to the bottom of **Cluster details** and locate the **Cluster parameter group**. Note the name of the cluster parameter group.

If the name of the cluster's parameter group is `default` (for example, `default.docdb3.6`), you must create a custom cluster parameter group and make it the cluster's parameter group before you continue. For more information, see the following:

1. Creating Amazon DocumentDB cluster parameter groups — If you don't have a custom cluster parameter group that you can use, create one.
2. Modifying an Amazon DocumentDB cluster — Modify your cluster to use the custom cluster parameter group.

2. **Determine the current value of the `tls` cluster parameter.**

a. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

b. In the navigation pane, choose **Parameter groups**.

c. In the list of cluster parameter groups, choose the name of the cluster parameter group you are interested in.

d. Locate the **Cluster parameters** section. In the list of cluster parameters, locate the `tls` cluster parameter row. At this point, the following four columns are important:

- **Cluster parameter name** — The name of the cluster parameters. For managing TLS, you're interested in the `tls` cluster parameter.
- **Values** — The current value of each cluster parameter.
- **Allowed values** — A list of values that can be applied to a cluster parameter.
- **Apply type** — Either **static** or **dynamic**. Changes to static cluster parameters can be applied only when the instances are rebooted. Changes to dynamic cluster parameters can be applied either immediately or when the instances are rebooted.

3. **Modify the value of the `tls` cluster parameter.**

   If the value of `tls` is not what is needs to be, modify its value for this cluster parameter group. To change the value of the `tls` cluster parameter, continue from the preceding section by following these steps.

   a. Choose the button to the left of the cluster parameter's name (`tls`).

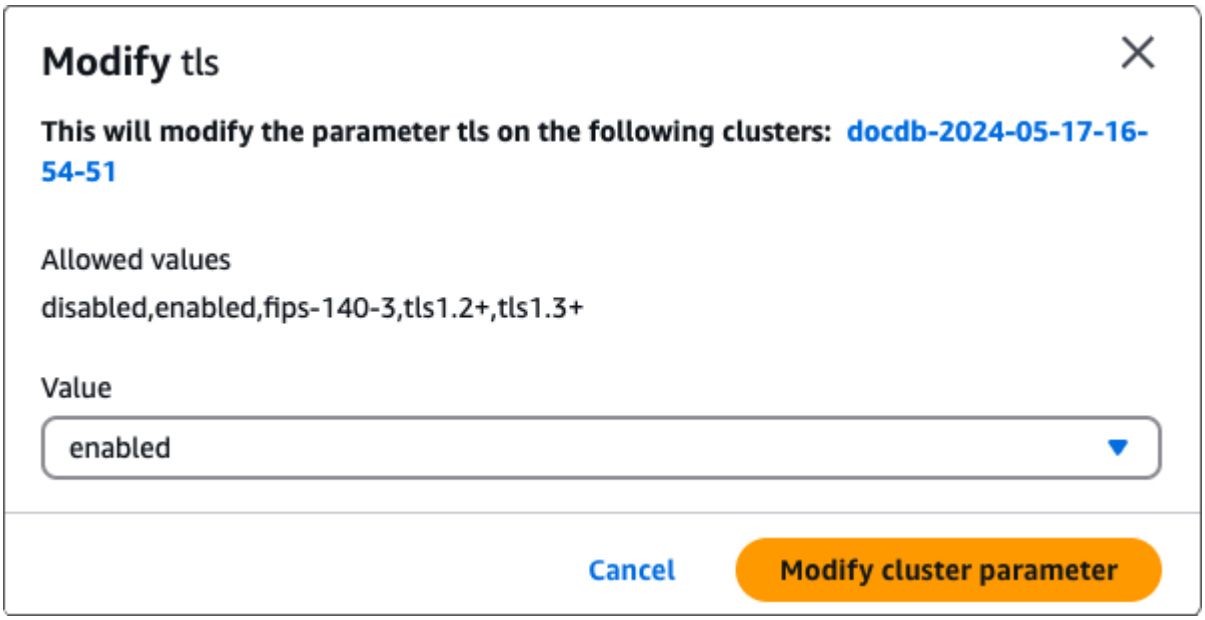
   b. Choose **Edit**.

   c. To change the value of `tls`, in the **Modify `tls`** dialog box, choose the value that you want for the cluster parameter in the drop-down list.

      Valid values are:

      - **disabled** — Disables TLS
      - **enabled** — Enables TLS. For Amazon DocumentDB 5.0 and earlier, this enables TLS 1.0 through 1.3, but use TLS 1.2 or higher. Only TLS 1.2 and higher is supported on Amazon DocumentDB 5.0 starting with minor version 5.0.1, Amazon DocumentDB 8.0 and newer.
      - **fips-140-3** — Enables TLS with FIPS. The cluster only accepts secure connections per the requirements of the Federal Information Processing Standards (FIPS) publication 140-3. This is only supported starting with Amazon DocumentDB 5.0 (engine version 3.0.3727) clusters in these Regions: ca-central-1, us-west-2, us-east-1, us-east-2, us-gov-east-1, us-gov-west-1.
      - **tls1.2+** — Enables TLS version 1.2 and above. This is only supported starting with Amazon DocumentDB 4.0 (engine version 2.0.10980) and Amazon DocumentDB (engine version 3.0.11051).

- **tls1.3+** — Enables TLS version 1.3 and above. This is only supported starting with Amazon DocumentDB 4.0 (engine version 2.0.10980) and Amazon DocumentDB (engine version 3.0.11051).



d. Choose **Modify cluster parameter**. The change is applied to each cluster instance when it is rebooted.

4. **Reboot the Amazon DocumentDB instance.**

   Reboot each instance of the cluster so that the change is applied to all instances in the cluster.

   a. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

   b. In the navigation pane, choose **Instances**.

   c. To specify an instance to reboot, locate the instance in the list of instances, and choose the button to the left of its name.

   d. Choose **Actions**, and then **Reboot**. Confirm that you want to reboot by choosing **Reboot**.

Using the AWS CLI

Follow these steps to perform management tasks for TLS encryption using the AWS CLI—such as identifying parameter groups, verifying the TLS value, and making needed modifications.

> **Note**
>
> Unless you specify differently when you create a cluster, the cluster is created with the default cluster parameter group. The parameters in the `default` cluster parameter group can't be modified (for example, `tls` enabled/disabled). So if your cluster is using a `default` cluster parameter group, you need to modify the cluster to use a non-default cluster parameter group. You might need to first create a custom cluster parameter group. For more information, see Creating Amazon DocumentDB cluster parameter groups.

1. **Determine the cluster parameter group that your cluster is using.**

   Run the `describe-db-clusters` command with the following options:

   - `--db-cluster-identifier`
   - `--query`

   In the following example, replace each *`user input placeholder`* with your cluster's information.

   ```
   aws docdb describe-db-clusters \
     --db-cluster-identifier mydocdbcluster \
     --query 'DBClusters[*].[DBClusterIdentifier,DBClusterParameterGroup]'
   ```

   Output from this operation looks something like the following (JSON format):

   ```
   [
       [
           "mydocdbcluster",
           "myparametergroup"
       ]
   ]
   ```

   If the name of the cluster's parameter group is `default` (for example, `default.docdb3.6`), you must have a custom cluster parameter group and make it the cluster's parameter group before you continue. For more information, see the following topics:

1. Creating Amazon DocumentDB cluster parameter groups — If you don't have a custom cluster parameter group that you can use, create one.
2. Modifying an Amazon DocumentDB cluster — Modify your cluster to use the custom cluster parameter group.

2. **Determine the current value of the `tls` cluster parameter.**

   To get more information about this cluster parameter group, run the `describe-db-cluster-parameters` command with the following options:

   - `--db-cluster-parameter-group-name`
   - `--query`

     Limits the output to just the fields of interest: `ParameterName`, `ParameterValue`,`AllowedValues`, and `ApplyType`.

   In the following example, replace each *`user input placeholder`* with your cluster's information.

```
aws docdb describe-db-cluster-parameters \
  --db-cluster-parameter-group-name myparametergroup \
  --query 'Parameters[*].[ParameterName,ParameterValue,AllowedValues,ApplyType]'
```

   Output from this operation looks something like the following (JSON format):

```
[
    [
        "audit_logs",
        "disabled",
        "enabled,disabled",
        "dynamic"
    ],
    [
        "tls",
        "disabled",
        "disabled,enabled,fips-140-3,tls1.2+,tls1.3+",
        "static"
    ],
    [
        "ttl_monitor",
```

```
        "enabled",
        "disabled,enabled",
        "dynamic"
    ]
]
```

3. **Modify the value of the `tls` cluster parameter.**

   If the value of `tls` is not what it needs to be, modify its value for this cluster parameter group. To change the value of the `tls` cluster parameter, run the [`modify-db-cluster-parameter-group`] command with the following options:

   - `--db-cluster-parameter-group-name` — Required. The name of the cluster parameter group to modify. This cannot be a `default.*` cluster parameter group.
   - `--parameters` — Required. A list of the cluster parameter group's parameters to modify.
     - `ParameterName` — Required. The name of the cluster parameter to modify.
     - `ParameterValue` — Required. The new value for this cluster parameter. Must be one of the cluster parameter's `AllowedValues`.
       - `enabled` — The cluster accepts secure connections using TLS. For Amazon DocumentDB 5.0 and earlier, this enables TLS 1.0 through 1.3, but use TLS 1.2 or higher. Only TLS 1.2 and higher is supported on Amazon DocumentDB 5.0 starting with minor version 5.0.1, Amazon DocumentDB 8.0 and newer.
       - `disabled` — The cluster does not accept secure connections using TLS.
       - `fips-140-3` — The cluster only accepts secure connections per the requirements of the Federal Information Processing Standards (FIPS) publication 140-3. This is only supported starting with Amazon DocumentDB 5.0 (engine version 3.0.3727) clusters in these Regions: ca-central-1, us-west-2, us-east-1, us-east-2, us-gov-east-1, us-gov-west-1.
       - `tls1.2+` — The cluster accepts secure connections using TLS version 1.2 and above. This is only supported starting with Amazon DocumentDB 4.0 (engine version 2.0.10980) and Amazon DocumentDB 5.0 (engine version 3.0.11051).
       - `tls1.3+` — The cluster accepts secure connections using TLS version 1.3 and above. This is only supported starting with Amazon DocumentDB 4.0 (engine version 2.0.10980) and Amazon DocumentDB 5.0 (engine version 3.0.11051).

- `ApplyMethod` — When this modification is to be applied. For static cluster parameters like `tle`, this value must be `pending-reboot`.
  - `pending-reboot` — Change is applied to an instance only after it is rebooted. You must reboot each cluster instance individually for this change to take place across all of the cluster's instances.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

The following code *disables* `tls`, applying the change to each instance when it is rebooted.

```
aws docdb modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=tls,ParameterValue=disabled,ApplyMethod=pending-
reboot"
```

The following code *enables* `tls`, applying the change to each instance when it is rebooted.

```
aws docdb modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup \
  --parameters "ParameterName=tls,ParameterValue=enabled,ApplyMethod=pending-
reboot"
```

The following code *enables* TLS with `fips-140-3`, applying the change to each instance when it is rebooted.

```
aws docdb modify-db-cluster-parameter-group \
  --db-cluster-parameter-group-name myparametergroup2 \
  --parameters "ParameterName=tls,ParameterValue=fips-140-3,ApplyMethod=pending-
reboot"
```

Output from this operation looks something like the following (JSON format):

```
{
    "DBClusterParameterGroupName": "myparametergroup"
}
```

4. **Reboot your Amazon DocumentDB instance.**

Reboot each instance of the cluster so that the change is applied to all instances in the cluster. To reboot an Amazon DocumentDB instance, run the `reboot-db-instance` command with the following option:

- `--db-instance-identifier`

The following code reboots the instance `mydocdbinstance`.

In the following examples, replace each *`user input placeholder`* with your cluster's information.

Example

For Linux, macOS, or Unix:

```
aws docdb reboot-db-instance \
  --db-instance-identifier mydocdbinstance
```

For Windows:

```
aws docdb reboot-db-instance ^
  --db-instance-identifier mydocdbinstance
```

Output from this operation looks something like the following (JSON format):

```
{
    "DBInstance": {
        "AutoMinorVersionUpgrade": true,
        "PubliclyAccessible": false,
        "PreferredMaintenanceWindow": "fri:09:32-fri:10:02",
        "PendingModifiedValues": {},
        "DBInstanceStatus": "rebooting",
        "DBSubnetGroup": {
            "Subnets": [
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1a"
                    },
                    "SubnetIdentifier": "subnet-4e26d263"
```

```
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1c"
                    },
                    "SubnetIdentifier": "subnet-afc329f4"
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1e"
                    },
                    "SubnetIdentifier": "subnet-b3806e8f"
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1d"
                    },
                    "SubnetIdentifier": "subnet-53ab3636"
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1b"
                    },
                    "SubnetIdentifier": "subnet-991cb8d0"
                },
                {
                    "SubnetStatus": "Active",
                    "SubnetAvailabilityZone": {
                        "Name": "us-east-1f"
                    },
                    "SubnetIdentifier": "subnet-29ab1025"
                }
            ],
            "SubnetGroupStatus": "Complete",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-91280df6",
            "DBSubnetGroupName": "default"
        },
        "PromotionTier": 2,
        "DBInstanceClass": "db.r5.4xlarge",
```

```
        "InstanceCreateTime": "2018-11-05T23:10:49.905Z",
        "PreferredBackupWindow": "00:00-00:30",
        "KmsKeyId": "arn:aws:kms:us-east-1:012345678901:key/0961325d-a50b-44d4-
b6a0-a177d5ff730b",
        "StorageEncrypted": true,
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-77186e0d"
            }
        ],
        "EngineVersion": "3.6.0",
        "DbiResourceId": "db-SAMPLERESOURCEID",
        "DBInstanceIdentifier": "mydocdbinstance",
        "Engine": "docdb",
        "AvailabilityZone": "us-east-1a",
        "DBInstanceArn": "arn:aws:rds:us-east-1:012345678901:db:sample-cluster-
instance-00",
        "BackupRetentionPeriod": 1,
        "Endpoint": {
            "Address": "mydocdbinstance.corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
            "Port": 27017,
            "HostedZoneId": "Z2R2ITUGPM61AM"
        },
        "DBClusterIdentifier": "mydocdbcluster"
    }
}
```

It takes a few minutes for your instance to reboot. You can use the instance only when its status is *available*. You can monitor the instance's status using the console or AWS CLI. For more information, see Monitoring an Amazon DocumentDB instance's status.

Key management

Amazon DocumentDB uses AWS Key Management Service (AWS KMS) to retrieve and manage encryption keys. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud. Using AWS KMS, you can create encryption keys and define the policies that control how these keys can be used. AWS KMS supports AWS CloudTrail, so you can audit key usage to verify that keys are being used appropriately.

You can use your KMS keys in combination with Amazon DocumentDB and supported AWS services such as Amazon Simple Storage Service (Amazon S3), Amazon Relational Database Service (Amazon RDS), Amazon Elastic Block Store (Amazon EBS), and Amazon Redshift. For a list of services that support AWS KMS, see How AWS services use AWS KMS in the *AWS Key Management Service Developer Guide*. For information about AWS KMS, see What is AWS Key Management Service?

Identity and Access Management for Amazon DocumentDB

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon DocumentDB resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon DocumentDB works with IAM
- Identity-based policy examples for Amazon DocumentDB
- Troubleshooting Amazon DocumentDB identity and access
- Managing access permissions to your Amazon DocumentDB resources
- Using identity-based policies (IAM policies) for Amazon DocumentDB
- AWS managed policies for Amazon DocumentDB
- Amazon DocumentDB API permissions: actions, resources, and conditions reference

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting Amazon DocumentDB identity and access)
- **Service administrator** - determine user access and submit permission requests (see How Amazon DocumentDB works with IAM)

- **IAM administrator** - write policies to manage access (see Identity-based policy examples for Amazon DocumentDB)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more

information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between

managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon DocumentDB works with IAM

Before you use IAM to manage access to Amazon DocumentDB, learn what IAM features are available to use with Amazon DocumentDB.

IAM features you can use with Amazon DocumentDB

IAM feature	Instance-based clusters	Elastic clusters
Identity-based policies	Yes	Yes
Resource-based policies	No	No
Policy actions	Yes	Yes
Policy resources	Yes	Yes
Policy condition keys (service-specific)	Yes	Yes
ACLs	No	No
ABAC (tags in policies)	Partial	Yes
Temporary credentials	Yes	Yes
Principal permissions	Yes	Yes
Service roles	Yes	Yes
Service-linked roles	No	Yes

To get a high-level view of how Amazon DocumentDB and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Amazon DocumentDB

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Amazon DocumentDB

To view examples of Amazon DocumentDB identity-based policies, see Identity-based policy examples for Amazon DocumentDB.

Resource-based policies within Amazon DocumentDB

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Policy actions for Amazon DocumentDB

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

> **Note**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).
> To see a list of RDS actions, see Actions defined by Amazon Relational Database Service in the *Service Authorization Reference*.
> To view policy actions for Amazon DocumentDB elastic clusters, see Actions defined by Amazon DocumentDB elastic clusters in the *Service Authorization Reference*.

Policy actions in Amazon DocumentDB use the following prefix before the action:

```
aws
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "aws:action1",
      "aws:action2"
         ]
```

To view examples of Amazon DocumentDB identity-based policies, see Identity-based policy examples for Amazon DocumentDB.

Policy resources for Amazon DocumentDB

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

> **Note**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).
> To view a list of RDS resource types and their ARNs, see Resources defined by Amazon Relational Database Service in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon Relational Database Service.
> To view resource types for Amazon DocumentDB elastic clusters, see Resource types defined by Amazon DocumentDB elastic clusters in the *Service Authorization Reference*.

To view examples of Amazon DocumentDB identity-based policies, see Identity-based policy examples for Amazon DocumentDB.

Policy condition keys for Amazon DocumentDB

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

> **Note**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).
> To see a list of RDS condition keys, see Condition keys for Amazon Relational Database Service in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon Relational Database Service.
> To view condition keys for Amazon DocumentDB elastic clusters, see Condition keys for Amazon DocumentDB elastic clusters in the *Service Authorization Reference*.

To view examples of Amazon DocumentDB identity-based policies, see Identity-based policy examples for Amazon DocumentDB.

ACLs in Amazon DocumentDB

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Amazon DocumentDB

> **Note**
>
> ABAC is only partially supported for instance-based clusters but is supported for elastic clusters.

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Amazon DocumentDB

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate

temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Cross-service principal permissions for Amazon DocumentDB

Supports forward access sessions (FAS): Yes

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Amazon DocumentDB

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

> **Warning**
>
> Changing the permissions for a service role might break Amazon DocumentDB functionality. Edit service roles only when Amazon DocumentDB provides guidance to do so.

Service-linked roles for Amazon DocumentDB

> **Note**
>
> Service-linked roles are not supported for instance-based clusters but are supported for elastic clusters.

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon DocumentDB

By default, users and roles don't have permission to create or modify Amazon DocumentDB resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Amazon DocumentDB, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for Amazon Relational Database Service in the *Service Authorization Reference*.

> **Note**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).
> For policy actions for Amazon DocumentDB elastic clusters, see Actions, resources, and condition keys for Amazon DocumentDB Elastic Clusters in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the Amazon DocumentDB console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Amazon DocumentDB resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Amazon DocumentDB console

To access the Amazon DocumentDB (with MongoDB compatibility) console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon DocumentDB resources in your AWS account. If you create an identity-based policy that is

more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Amazon DocumentDB console, also attach the Amazon DocumentDB *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
```

```
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshooting Amazon DocumentDB identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon DocumentDB and IAM.

Topics

- I am not authorized to perform an action in Amazon DocumentDB
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Amazon DocumentDB resources

I am not authorized to perform an action in Amazon DocumentDB

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `aws:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 aws:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `aws:GetWidget` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon DocumentDB.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon DocumentDB. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Amazon DocumentDB resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon DocumentDB supports these features, see How Amazon DocumentDB works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.

- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access permissions to your Amazon DocumentDB resources

Every AWS resource is owned by an AWS account, and permissions to create or access the resources are governed by permissions policies. An account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles), and some services (such as AWS Lambda) also support attaching permissions policies to resources.

> **Note**
>
> An *account administrator* (or administrator user) is a user with administrator permissions. For more information, see IAM Best Practices in the *IAM User Guide*.

Topics

- Amazon DocumentDB resources and operations
- Understanding resource ownership
- Managing access to resources
- Specifying policy elements: actions, effects, resources, and principals
- Specifying conditions in a policy

Amazon DocumentDB resources and operations

In Amazon DocumentDB, the primary resource is a *cluster*. Amazon DocumentDB supports other resources that can be used with the primary resource such as *instances*, *parameter groups*, and *event subscriptions*. These resources are referred to as *subresources*.

These resources and subresources have unique Amazon Resource Names (ARNs) associated with them, as shown in the following table.

Resource Type	ARN Format
Cluster	`arn:aws:rds:` *region*`:`*account-id* `:cluster:` *db-cluster-name*
Cluster parameter group	`arn:aws:rds:` *region*`:`*account-id* `:cluster-pg:` *cluster-parameter-group-name*
Cluster snapshot	`arn:aws:rds:` *region*`:`*account-id* `:cluster-snapshot:` *cluster-snapshot-name*
Instance	`arn:aws:rds:` *region*`:`*account-id* `:db:`*db-instance-name*
Security group	`arn:aws:rds:` *region*`:`*account-id* `:secgrp:`*security-group-name*
Subnet group	`arn:aws:rds:` *region*`:`*account-id* `:subgrp:`*subnet-group-name*

Amazon DocumentDB provides a set of operations to work with the Amazon DocumentDB resources. For a list of available operations, see Actions.

Understanding resource ownership

A *resource owner* is the AWS account that created a resource. That is, the resource owner is the AWS account of the *principal entity* (the root account, an IAM user, or an IAM role) that authenticates the request that creates the resource. The following examples illustrate how this works:

- If you use the root account credentials of your AWS account to create an Amazon DocumentDB resource, such as an instance, your AWS account is the owner of the Amazon DocumentDB resource.
- If you create an IAM user in your AWS account and grant permissions to create Amazon DocumentDB resources to that user, the user can create Amazon DocumentDB resources. However, your AWS account, to which the user belongs, owns the Amazon DocumentDB resources.
- If you create an IAM role in your AWS account with permissions to create Amazon DocumentDB resources, anyone who can assume the role can create Amazon DocumentDB resources. Your AWS account, to which the role belongs, owns the Amazon DocumentDB resources.

Managing access to resources

A *permissions policy* describes who has access to what. The following section explains the available options for creating permissions policies.

> **Note**
>
> This section discusses using IAM in the context of Amazon DocumentDB. It doesn't provide detailed information about the IAM service. For complete IAM documentation, see What Is IAM? in the *IAM User Guide*. For information about IAM policy syntax and descriptions, see AWS IAM Policy Reference in the *IAM User Guide*.

Policies that are attached to an IAM identity are referred to as *identity-based* policies (IAM policies). Policies that are attached to a resource are referred to as *resource-based* policies. Amazon DocumentDB supports only identity-based policies (IAM policies).

Topics

- Identity-based policies (IAM policies)
- Resource-based policies

Identity-based policies (IAM policies)

You can attach policies to IAM identities. For example, you can do the following:

- **Attach a permissions policy to a user or a group in your account** – An account administrator can use a permissions policy that is associated with a particular user to grant permissions for that user to create an Amazon DocumentDB resource, such as an instance.
- **Attach a permissions policy to a role (grant cross-account permissions)** – You can attach an identity-based permissions policy to an IAM role to grant cross-account permissions. For example, an administrator can create a role to grant cross-account permissions to another AWS account or an AWS service as follows:
  1. Account A administrator creates an IAM role and attaches a permissions policy to the role that grants permissions on resources in Account A.
  2. Account A administrator attaches a trust policy to the role identifying Account B as the principal who can assume the role.

3. Account B administrator can then delegate permissions to assume the role to any users in Account B. Doing this allows the users in Account B to create or access resources in Account A. The principal in the trust policy can also be an AWS service principal if you want to grant permissions to an AWS service to assume the role.

For more information about using IAM to delegate permissions, see Access Management in the *IAM User Guide*.

The following is an example policy that allows the user with the ID `123456789012` to create instances for your AWS account. The new instance must use an option group and a parameter group that starts with `default`, and it must use the `default` subnet group.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateDBInstanceOnly",
            "Effect": "Allow",
            "Action": [
                "rds:CreateDBInstance"
            ],
            "Resource": [
                "arn:aws:rds:*:123456789012:db:test*",
                "arn:aws:rds:*:123456789012:pg:cluster-pg:default*",
                "arn:aws:rds:*:123456789012:subgrp:default"
            ]
        }
    ]
}
```

For more information about using identity-based policies with Amazon DocumentDB, see Using identity-based policies (IAM policies) for Amazon DocumentDB. For more information about users, groups, roles, and permissions, see Identities (Users, Groups, and Roles) in the *IAM User Guide*.

Resource-based policies

Other services, such as Amazon Simple Storage Service (Amazon S3), also support resource-based permissions policies. For example, you can attach a policy to an Amazon S3 bucket to manage access permissions to that bucket. Amazon DocumentDB doesn't support resource-based policies.

Specifying policy elements: actions, effects, resources, and principals

For each Amazon DocumentDB resource (see Amazon DocumentDB resources and operations), the service defines a set of API operations. For more information, see Actions. To grant permissions for these API operations, Amazon DocumentDB defines a set of actions that you can specify in a policy. Performing an API operation can require permissions for more than one action.

The following are the basic policy elements:

- **Resource** – In a policy, you use an Amazon Resource Name (ARN) to identify the resource to which the policy applies.
- **Action** – You use action keywords to identify resource operations that you want to allow or deny. For example, the `rds:DescribeDBInstances` permission allows the user to perform the `DescribeDBInstances` operation.
- **Effect** – You specify the effect when the user requests the specific action—this can be either allow or deny. If you don't explicitly grant access to (allow) a resource, access is implicitly denied. You can also explicitly deny access to a resource, which you might do to make sure that a user cannot access it, even if a different policy grants access.
- **Principal** – In identity-based policies (IAM policies), the user that the policy is attached to is the implicit principal. For resource-based policies, you specify the user, account, service, or other entity that you want to receive permissions (applies to resource-based policies only). Amazon DocumentDB doesn't support resource-based policies.

To learn more about IAM policy syntax and descriptions, see AWS IAM Policy Reference in the *IAM User Guide*.

For a table showing all of the Amazon DocumentDB API actions and the resources that they apply to, see Amazon DocumentDB API permissions: actions, resources, and conditions reference.

Specifying conditions in a policy

When you grant permissions, you can use the IAM policy language to specify the conditions when a policy should take effect. For example, you might want a policy to be applied only after a specific

date. For more information about specifying conditions in a policy language, see Condition in the *IAM User Guide*.

To express conditions, you use predefined condition keys. Amazon DocumentDB has no service-specific context keys that can be used in an IAM policy. For a list of global condition context keys that are available to all services, see Available global condition keys in the *IAM User Guide*.

Using identity-based policies (IAM policies) for Amazon DocumentDB

> **Important**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon RDS. Amazon DocumentDB console, AWS CLI, and API calls are logged as calls made to the Amazon RDS API.
> First review the introductory topics that explain the basic concepts and options for managing access to your Amazon DocumentDB resources. For more information, see Managing access permissions to your Amazon DocumentDB resources.

This topic provides examples of identity-based policies in which an account administrator can attach permissions policies to IAM identities (that is, users, groups, and roles).

The following is an example of an IAM policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateDBInstanceOnly",
            "Effect": "Allow",
            "Action": [
                "rds:CreateDBInstance"
            ],
            "Resource": [
                "arn:aws:rds:*:123456789012:db:test*",
                "arn:aws:rds:*:123456789012:pg:cluster-pg:default*",
                "arn:aws:rds:*:123456789012:subgrp:default"
```

```
            ]
        }
    ]
}
```

The policy includes a single statement that specifies the following permissions for the IAM user:

- The policy allows the IAM user to create an instance using the CreateDBInstance action (this also applies to the create-db-instance AWS CLI operation and the AWS Management Console).
- The `Resource` element specifies that the user can perform actions on or with resources. You specify resources using an Amazon Resource Name (ARN). This ARN includes the name of the service that the resource belongs to (`rds`), the AWS Region (`*` indicates any Region in this example), the user account number (`123456789012` is the user ID in this example), and the type of resource.

  The `Resource` element in the example specifies the following policy constraints on resources for the user:

  - The instance identifier for the new instance must begin with `test` (for example, `testCustomerData1`, `test-region2-data`).
  - The cluster parameter group for the new instance must begin with `default`.
  - The subnet group for the new instance must be the `default` subnet group.

The policy doesn't specify the `Principal` element because in an identity-based policy you don't specify the principal who gets the permission. When you attach policy to a user, the user is the implicit principal. When you attach a permissions policy to an IAM role, the principal identified in the role's trust policy gets the permissions.

For a table showing all of the Amazon DocumentDB API operations and the resources that they apply to, see Amazon DocumentDB API permissions: actions, resources, and conditions reference.

Permissions required to use the Amazon DocumentDB console

For a user to work with the Amazon DocumentDB console, that user must have a minimum set of permissions. These permissions allow the user to describe the Amazon DocumentDB resources for their AWS account and to provide other related information, including Amazon EC2 security and network information.

If you create an IAM policy that is more restrictive than the minimum required permissions, the console won't function as intended for users with that IAM policy. To ensure that those users can still use the Amazon DocumentDB console, also attach the `AmazonDocDBConsoleFullAccess` managed policy to the user, as described in AWS managed policies for Amazon DocumentDB.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the Amazon DocumentDB API.

Customer managed policy examples

In this section, you can find example user policies that grant permissions for various Amazon DocumentDB actions. These policies work when you are using Amazon DocumentDB API actions, AWS SDKs, or the AWS CLI. When you are using the console, you need to grant additional permissions specific to the console, which is discussed in Permissions required to use the Amazon DocumentDB console.

For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS) and Amazon Neptune.

> **Note**
>
> All examples use the US East (N. Virginia) Region (`us-east-1`) and contain fictitious account IDs.

Examples

- Example 1: Allow a user to perform any describe action on any Amazon DocumentDB resource
- Example 2: Prevent a user from deleting an instance
- Example 3: Prevent a user from creating a cluster unless storage encryption is enabled

Example 1: Allow a user to perform any describe action on any Amazon DocumentDB resource

The following permissions policy grants permissions to a user to run all of the actions that begin with `Describe`. These actions show information about an Amazon DocumentDB resource, such as an instance. The wildcard character (*) in the `Resource` element indicates that the actions are allowed for all Amazon DocumentDB resources that are owned by the account.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"AllowRDSDescribe",
         "Effect":"Allow",
         "Action":"rds:Describe*",
         "Resource":"*"
      }
   ]
}
```

Example 2: Prevent a user from deleting an instance

The following permissions policy grants permissions to prevent a user from deleting a specific instance. For example, you might want to deny the ability to delete your production instances to any user that is not an administrator.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"DenyDelete1",
         "Effect":"Deny",
         "Action":"rds:DeleteDBInstance",
         "Resource":"arn:aws:rds:us-east-1:123456789012:db:my-db-instance"
      }
   ]
}
```

Example 3: Prevent a user from creating a cluster unless storage encryption is enabled

The following permissions policy denies permissions to a user from creating an Amazon DocumentDB cluster unless storage encryption is enabled.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Sid": "PreventUnencryptedDocumentDB",
         "Effect": "Deny",
         "Action": "RDS:CreateDBCluster",
         "Condition": {
         "Bool": {
         "rds:StorageEncrypted": "false"
      },
         "StringEquals": {
         "rds:DatabaseEngine": "docdb"
         }
      },
      "Resource": "*"
      }
   ]
}
```

AWS managed policies for Amazon DocumentDB

To add permissions to users, groups, and roles, it is easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the `ViewOnlyAccess` AWS managed policy provides read-only access to many AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

The following AWS managed policies, which you can attach to users in your account, are specific to Amazon DocumentDB:

- AmazonDocDBFullAccess – Grants full access to all Amazon DocumentDB resources for the root AWS account.
- AmazonDocDBReadOnlyAccess – Grants read-only access to all Amazon DocumentDB resources for the root AWS account.
- AmazonDocDBConsoleFullAccess – Grants full access to manage Amazon DocumentDB and Amazon DocumentDB elastic cluster resources using the AWS Management Console.
- AmazonDocDBElasticReadOnlyAccess – Grants read-only access to all Amazon DocumentDB elastic cluster resources for the root AWS account.
- AmazonDocDBElasticFullAccess – Grants full access to all Amazon DocumentDB elastic cluster resources for the root AWS account.

AmazonDocDBFullAccess

This policy grants administrative permissions that allow a principal full access to all Amazon DocumentDB actions. The permissions in this policy are grouped as follows:

- The Amazon DocumentDB permissions allow all Amazon DocumentDB actions.
- Some of the Amazon EC2 permissions in this policy are required to validate the passed resources in an API request. This is to make sure Amazon DocumentDB is able to successfully use the resources with a cluster. The rest of the Amazon EC2 permissions in this policy allow Amazon DocumentDB to create AWS resources that are needed to make it possible for you to connect to your clusters.
- The Amazon DocumentDB permissions are used during API calls to validate the passed resources in a request. They are required for Amazon DocumentDB to be able to use the passed key with the Amazon DocumentDB cluster.
- The CloudWatch Logs are required for Amazon DocumentDB to be able to ensure that the log delivery destinations are reachable, and that they are valid for broker log use.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "rds:AddRoleToDBCluster",
                "rds:AddSourceIdentifierToSubscription",
                "rds:AddTagsToResource",
                "rds:ApplyPendingMaintenanceAction",
                "rds:CopyDBClusterParameterGroup",
                "rds:CopyDBClusterSnapshot",
                "rds:CopyDBParameterGroup",
                "rds:CreateDBCluster",
                "rds:CreateDBClusterParameterGroup",
                "rds:CreateDBClusterSnapshot",
                "rds:CreateDBInstance",
                "rds:CreateDBParameterGroup",
                "rds:CreateDBSubnetGroup",
                "rds:CreateEventSubscription",
                "rds:DeleteDBCluster",
                "rds:DeleteDBClusterParameterGroup",
                "rds:DeleteDBClusterSnapshot",
                "rds:DeleteDBInstance",
                "rds:DeleteDBParameterGroup",
                "rds:DeleteDBSubnetGroup",
                "rds:DeleteEventSubscription",
                "rds:DescribeAccountAttributes",
                "rds:DescribeCertificates",
                "rds:DescribeDBClusterParameterGroups",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:DescribeDBClusterSnapshots",
                "rds:DescribeDBClusters",
                "rds:DescribeDBEngineVersions",
                "rds:DescribeDBInstances",
                "rds:DescribeDBLogFiles",
                "rds:DescribeDBParameterGroups",
                "rds:DescribeDBParameters",
                "rds:DescribeDBSecurityGroups",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribeEngineDefaultClusterParameters",
```

```
                "rds:DescribeEngineDefaultParameters",
                "rds:DescribeEventCategories",
                "rds:DescribeEventSubscriptions",
                "rds:DescribeEvents",
                "rds:DescribeOptionGroups",
                "rds:DescribeOrderableDBInstanceOptions",
                "rds:DescribePendingMaintenanceActions",
                "rds:DescribeValidDBInstanceModifications",
                "rds:DownloadDBLogFilePortion",
                "rds:FailoverDBCluster",
                "rds:ListTagsForResource",
                "rds:ModifyDBCluster",
                "rds:ModifyDBClusterParameterGroup",
                "rds:ModifyDBClusterSnapshotAttribute",
                "rds:ModifyDBInstance",
                "rds:ModifyDBParameterGroup",
                "rds:ModifyDBSubnetGroup",
                "rds:ModifyEventSubscription",
                "rds:PromoteReadReplicaDBCluster",
                "rds:RebootDBInstance",
                "rds:RemoveRoleFromDBCluster",
                "rds:RemoveSourceIdentifierFromSubscription",
                "rds:RemoveTagsFromResource",
                "rds:ResetDBClusterParameterGroup",
                "rds:ResetDBParameterGroup",
                "rds:RestoreDBClusterFromSnapshot",
                "rds:RestoreDBClusterToPointInTime"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        },
        {
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics",
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcs",
                "kms:ListAliases",
```

```
                "kms:ListKeyPolicies",
                "kms:ListKeys",
                "kms:ListRetirableGrants",
                "logs:DescribeLogStreams",
                "logs:GetLogEvents",
                "sns:ListSubscriptions",
                "sns:ListTopics",
                "sns:Publish"
            ],
            "Effect": "Allow",
            "Resource": [
                "*"
            ]
        },
        {
            "Action": "iam:CreateServiceLinkedRole",
            "Effect": "Allow",
            "Resource": "arn:aws:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "rds.amazonaws.com"
                }
            }
        }
    ]
}
```

AmazonDocDBReadOnlyAccess

This policy grants read-only permissions that allow users to view information in Amazon DocumentDB. Principals with this policy attached can't make any updates or delete exiting resources, nor can they create new Amazon DocumentDB resources. For example, principals with these permissions can view the list of clusters and configurations associated with their account, but cannot change the configuration or settings of any clusters. The permissions in this policy are grouped as follows:

- Amazon DocumentDB permissions allow you to list Amazon DocumentDB resources, describe them, and get information about them.
- Amazon EC2 permissions are used to describe the Amazon VPC, subnets, security groups, and ENIs that are associated with a cluster.

- An Amazon DocumentDB permission is used to describe the key that is associated with the cluster.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "rds:DescribeAccountAttributes",
                "rds:DescribeCertificates",
                "rds:DescribeDBClusterParameterGroups",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:DescribeDBClusterSnapshots",
                "rds:DescribeDBClusters",
                "rds:DescribeDBEngineVersions",
                "rds:DescribeDBInstances",
                "rds:DescribeDBLogFiles",
                "rds:DescribeDBParameterGroups",
                "rds:DescribeDBParameters",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribeEventCategories",
                "rds:DescribeEventSubscriptions",
                "rds:DescribeEvents",
                "rds:DescribeOrderableDBInstanceOptions",
                "rds:DescribePendingMaintenanceActions",
                "rds:DownloadDBLogFilePortion",
                "rds:ListTagsForResource"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Action": [
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
```

```
        {
            "Action": [
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeInternetGateways",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcs"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Action": [
                "kms:ListKeys",
                "kms:ListRetirableGrants",
                "kms:ListAliases",
                "kms:ListKeyPolicies"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Action": [
                "logs:DescribeLogStreams",
                "logs:GetLogEvents"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:logs:*:*:log-group:/aws/rds/*:log-stream:*",
                "arn:aws:logs:*:*:log-group:/aws/docdb/*:log-stream:*"
            ]
        }
    ]
}
```

AmazonDocDBConsoleFullAccess

Grants full access to manage Amazon DocumentDB resources using the AWS Management Console for following:

- The Amazon DocumentDB permissions to allow all Amazon DocumentDB and Amazon DocumentDB cluster actions.
- Some of the Amazon EC2 permissions in this policy are required to validate the passed resources in an API request. This is to make sure Amazon DocumentDB is able to successfully use the resources to provision and maintain the cluster. The rest of the Amazon EC2 permissions in this policy allow Amazon DocumentDB to create AWS resources that are needed to make it possible for you to connect to your clusters like VPCEndpoint.
- AWS KMS permissions are used during API calls to AWS KMS to validate the passed resources in a request. They are required for Amazon DocumentDB to be able to use the passed key to encrypt and decrypt the data at rest with the Amazon DocumentDB elastic cluster.
- The CloudWatch Logs are required for Amazon DocumentDB to be able to ensure that the log delivery destinations are reachable, and that they are valid for auditing and profiling log use.
- Secrets Manager permissions are required to validate given secret and use it setup the admin user for Amazon DocumentDB elastic clusters.
- Amazon RDS permissions are required for Amazon DocumentDB cluster management actions. For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon RDS.
- SNS permissions allows principals to Amazon Simple Notification Service (Amazon SNS) subscriptions and topics, and to publish Amazon SNS messages.
- IAM permissions are required for creating the service linked roles required for metrics and logs publishing.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DocdbSids",
            "Effect": "Allow",
            "Action": [
                "docdb-elastic:CreateCluster",
                "docdb-elastic:UpdateCluster",
                "docdb-elastic:GetCluster",
                "docdb-elastic:DeleteCluster",
                "docdb-elastic:ListClusters",
```

```
                "docdb-elastic:CreateClusterSnapshot",
                "docdb-elastic:GetClusterSnapshot",
                "docdb-elastic:DeleteClusterSnapshot",
                "docdb-elastic:ListClusterSnapshots",
                "docdb-elastic:RestoreClusterFromSnapshot",
                "docdb-elastic:TagResource",
                "docdb-elastic:UntagResource",
                "docdb-elastic:ListTagsForResource",
                "docdb-elastic:CopyClusterSnapshot",
                "docdb-elastic:StartCluster",
                "docdb-elastic:StopCluster",
                "docdb-elastic:GetPendingMaintenanceAction",
                "docdb-elastic:ListPendingMaintenanceActions",
                "docdb-elastic:ApplyPendingMaintenanceAction",
                "rds:AddRoleToDBCluster",
                "rds:AddSourceIdentifierToSubscription",
                "rds:AddTagsToResource",
                "rds:ApplyPendingMaintenanceAction",
                "rds:CopyDBClusterParameterGroup",
                "rds:CopyDBClusterSnapshot",
                "rds:CopyDBParameterGroup",
                "rds:CreateDBCluster",
                "rds:CreateDBClusterParameterGroup",
                "rds:CreateDBClusterSnapshot",
                "rds:CreateDBInstance",
                "rds:CreateDBParameterGroup",
                "rds:CreateDBSubnetGroup",
                "rds:CreateEventSubscription",
                "rds:CreateGlobalCluster",
                "rds:DeleteDBCluster",
                "rds:DeleteDBClusterParameterGroup",
                "rds:DeleteDBClusterSnapshot",
                "rds:DeleteDBInstance",
                "rds:DeleteDBParameterGroup",
                "rds:DeleteDBSubnetGroup",
                "rds:DeleteEventSubscription",
                "rds:DeleteGlobalCluster",
                "rds:DescribeAccountAttributes",
                "rds:DescribeCertificates",
                "rds:DescribeDBClusterParameterGroups",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBClusterSnapshotAttributes",
                "rds:DescribeDBClusterSnapshots",
                "rds:DescribeDBClusters",
```

```
                "rds:DescribeDBEngineVersions",
                "rds:DescribeDBInstances",
                "rds:DescribeDBLogFiles",
                "rds:DescribeDBParameterGroups",
                "rds:DescribeDBParameters",
                "rds:DescribeDBSecurityGroups",
                "rds:DescribeDBSubnetGroups",
                "rds:DescribeEngineDefaultClusterParameters",
                "rds:DescribeEngineDefaultParameters",
                "rds:DescribeEventCategories",
                "rds:DescribeEventSubscriptions",
                "rds:DescribeEvents",
                "rds:DescribeGlobalClusters",
                "rds:DescribeOptionGroups",
                "rds:DescribeOrderableDBInstanceOptions",
                "rds:DescribePendingMaintenanceActions",
                "rds:DescribeValidDBInstanceModifications",
                "rds:DownloadDBLogFilePortion",
                "rds:FailoverDBCluster",
                "rds:ListTagsForResource",
                "rds:ModifyDBCluster",
                "rds:ModifyDBClusterParameterGroup",
                "rds:ModifyDBClusterSnapshotAttribute",
                "rds:ModifyDBInstance",
                "rds:ModifyDBParameterGroup",
                "rds:ModifyDBSubnetGroup",
                "rds:ModifyEventSubscription",
                "rds:ModifyGlobalCluster",
                "rds:PromoteReadReplicaDBCluster",
                "rds:RebootDBInstance",
                "rds:RemoveFromGlobalCluster",
                "rds:RemoveRoleFromDBCluster",
                "rds:RemoveSourceIdentifierFromSubscription",
                "rds:RemoveTagsFromResource",
                "rds:ResetDBClusterParameterGroup",
                "rds:ResetDBParameterGroup",
                "rds:RestoreDBClusterFromSnapshot",
                "rds:RestoreDBClusterToPointInTime"
            ],
            "Resource": [
                "*"
            ]
        },
        {
```

```
            "Sid": "DependencySids",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "cloudwatch:GetMetricData",
                "cloudwatch:GetMetricStatistics",
                "cloudwatch:ListMetrics",
                "ec2:AllocateAddress",
                "ec2:AssignIpv6Addresses",
                "ec2:AssignPrivateIpAddresses",
                "ec2:AssociateAddress",
                "ec2:AssociateRouteTable",
                "ec2:AssociateSubnetCidrBlock",
                "ec2:AssociateVpcCidrBlock",
                "ec2:AttachInternetGateway",
                "ec2:AttachNetworkInterface",
                "ec2:CreateCustomerGateway",
                "ec2:CreateDefaultSubnet",
                "ec2:CreateDefaultVpc",
                "ec2:CreateInternetGateway",
                "ec2:CreateNatGateway",
                "ec2:CreateNetworkInterface",
                "ec2:CreateRoute",
                "ec2:CreateRouteTable",
                "ec2:CreateSecurityGroup",
                "ec2:CreateSubnet",
                "ec2:CreateVpc",
                "ec2:CreateVpcEndpoint",
                "ec2:DescribeAccountAttributes",
                "ec2:DescribeAddresses",
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeCustomerGateways",
                "ec2:DescribeInstances",
                "ec2:DescribeNatGateways",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribePrefixLists",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroupReferences",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeVpcEndpoints",
                "ec2:DescribeVpcs",
                "ec2:ModifyNetworkInterfaceAttribute",
```

```
                "ec2:ModifySubnetAttribute",
                "ec2:ModifyVpcAttribute",
                "ec2:ModifyVpcEndpoint",
                "kms:DescribeKey",
                "kms:ListAliases",
                "kms:ListKeyPolicies",
                "kms:ListKeys",
                "kms:ListRetirableGrants",
                "logs:DescribeLogStreams",
                "logs:GetLogEvents",
                "sns:ListSubscriptions",
                "sns:ListTopics",
                "sns:Publish"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "DocdbSLRSid",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/rds.amazonaws.com/
AWSServiceRoleForRDS",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "rds.amazonaws.com"
                }
            }
        },
        {
            "Sid": "DocdbElasticSLRSid",
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/docdb-
elastic.amazonaws.com/AWSServiceRoleForDocDB-Elastic",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "docdb-elastic.amazonaws.com"
                }
            }
        }
    ]
```

```
}
```

AmazonDocDBElasticReadOnlyAccess

This policy grants read-only permissions that allow users to view elastic cluster information in Amazon DocumentDB. Principals with this policy attached can't make any updates or delete exiting resources, nor can they create new Amazon DocumentDB resources. For example, principals with these permissions can view the list of clusters and configurations associated with their account, but cannot change the configuration or settings of any clusters. The permissions in this policy are grouped as follows:

- Amazon DocumentDB elastic cluster permissions allow you to list Amazon DocumentDB elastic cluster resources, describe them, and get information about them.
- CloudWatch permissions are used to verify service metrics.

JSON

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "docdb-elastic:ListClusters",
            "docdb-elastic:GetCluster",
            "docdb-elastic:ListClusterSnapshots",
            "docdb-elastic:GetClusterSnapshot",
            "docdb-elastic:ListTagsForResource"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Allow",
         "Action": [
            "cloudwatch:GetMetricData",
            "cloudwatch:ListMetrics",
            "cloudwatch:GetMetricStatistics"
         ],
         "Resource": "*"
```

```
        }
    ]
}
```

AmazonDocDBElasticFullAccess

This policy grants administrative permissions that allow a principal full access to all Amazon DocumentDB actions for Amazon DocumentDB elastic cluster.

This policy uses AWS tags (https://docs.aws.amazon.com/tag-editor/latest/userguide/tagging.html) within conditions to scope access to resources. If you are using a secret, it must be tagged with tag key `DocDBElasticFullAccess` and a tag value. If you are using a customer managed key, it must be tagged with tag key `DocDBElasticFullAccess` and a tag value.

The permissions in this policy are grouped as follows:

- Amazon DocumentDB elastic cluster permissions allow all Amazon DocumentDB actions.
- Some of the Amazon EC2 permissions in this policy are required to validate the passed resources in an API request. This is to make sure Amazon DocumentDB is able to successfully use the resources to provision and maintain the cluster. The rest of the Amazon EC2 permissions in this policy allow Amazon DocumentDB to create AWS resources that are needed to make it possible for you to connect to your clusters like a VPC endpoint.
- AWS KMS permissions are required for Amazon DocumentDB to be able to use the passed key to encrypt and decrypt the data at rest within the Amazon DocumentDB elastic cluster.

> **Note**
>
> The customer managed key must have a tag with key `DocDBElasticFullAccess` and a tag value.

- SecretsManager permissions are required to validate given secret and use it setup the admin user for Amazon DocumentDB elastic clusters.

> **Note**
>
> The secret used must have a tag with key `DocDBElasticFullAccess` and a tag value.

- IAM permissions are required for creating the service linked roles required for metrics and logs publishing.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DocdbElasticSid",
            "Effect": "Allow",
            "Action": [
                "docdb-elastic:CreateCluster",
                "docdb-elastic:UpdateCluster",
                "docdb-elastic:GetCluster",
                "docdb-elastic:DeleteCluster",
                "docdb-elastic:ListClusters",
                "docdb-elastic:CreateClusterSnapshot",
                "docdb-elastic:GetClusterSnapshot",
                "docdb-elastic:DeleteClusterSnapshot",
                "docdb-elastic:ListClusterSnapshots",
                "docdb-elastic:RestoreClusterFromSnapshot",
                "docdb-elastic:TagResource",
                "docdb-elastic:UntagResource",
                "docdb-elastic:ListTagsForResource",
                "docdb-elastic:CopyClusterSnapshot",
                "docdb-elastic:StartCluster",
                "docdb-elastic:StopCluster",
                "docdb-elastic:GetPendingMaintenanceAction",
                "docdb-elastic:ListPendingMaintenanceActions",
                "docdb-elastic:ApplyPendingMaintenanceAction"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "EC2Sid",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateVpcEndpoint",
                "ec2:DescribeVpcEndpoints",
```

```
                "ec2:DeleteVpcEndpoints",
                "ec2:ModifyVpcEndpoint",
                "ec2:DescribeVpcAttribute",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeSubnets",
                "ec2:DescribeVpcs",
                "ec2:DescribeAvailabilityZones",
                "secretsmanager:ListSecrets"
            ],
            "Resource": [
                "*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:CalledViaFirst": "docdb-elastic.amazonaws.com"
                }
            }
        },
        {
            "Sid": "KMSSid",
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "kms:ViaService": [
                        "docdb-elastic.*.amazonaws.com"
                    ],
                    "aws:ResourceTag/DocDBElasticFullAccess": "*"
                }
            }
        },
        {
            "Sid": "KMSGrantSid",
            "Effect": "Allow",
            "Action": [
                "kms:CreateGrant"
            ],
            "Resource": "*",
            "Condition": {
```

```
                "StringLike": {
                    "aws:ResourceTag/DocDBElasticFullAccess": "*",
                    "kms:ViaService": [
                        "docdb-elastic.*.amazonaws.com"
                    ]
                },
                "Bool": {
                    "kms:GrantIsForAWSResource": true
                }
            }
        },
        {
            "Sid": "SecretManagerSid",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:ListSecretVersionIds",
                "secretsmanager:DescribeSecret",
                "secretsmanager:GetSecretValue",
                "secretsmanager:GetResourcePolicy"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "secretsmanager:ResourceTag/DocDBElasticFullAccess": "*"
                },
                "StringEquals": {
                    "aws:CalledViaFirst": "docdb-elastic.amazonaws.com"
                }
            }
        },
        {
            "Sid": "CloudwatchSid",
            "Effect": "Allow",
            "Action": [
                "cloudwatch:GetMetricData",
                "cloudwatch:ListMetrics",
                "cloudwatch:GetMetricStatistics"
            ],
            "Resource": [
                "*"
            ]
        },
        {
            "Sid": "SLRSid",
```

```
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "arn:aws:iam::*:role/aws-service-role/docdb-
elastic.amazonaws.com/AWSServiceRoleForDocDB-Elastic",
            "Condition": {
                "StringLike": {
                    "iam:AWSServiceName": "docdb-elastic.amazonaws.com"
                }
            }
        }
    ]
}
```

AmazonDocDB-ElasticServiceRolePolicy

You can't attach `AmazonDocDBElasticServiceRolePolicy` to your AWS Identity and Access Management entities. This policy is attached to a service-linked role that allows Amazon DocumentDB to perform actions on your behalf. For more information, see Service-linked roles in elastic clusters.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "cloudwatch:namespace": [
                        "AWS/DocDB-Elastic"
                    ]
                }
            }
        }
    ]
```

```
}
```

Amazon DocumentDB updates to AWS managed policies

Change	Description	Date
AmazonDocDBElasticFullAccess, AmazonDocDBConsoleFullAccess - Change	Policies updated to add pending maintenance actions.	2/11/2025
AmazonDocDBElasticFullAccess, AmazonDocDBConsoleFullAccess - Change	Policies updated to add start/stop cluster and copy cluster snapshot actions.	2/21/2024
AmazonDocDBElasticReadOnlyAccess, AmazonDocDBElasticFullAccess - Change	Policies updated to add `cloudwatch:GetMetricData` action.	6/21/2023
AmazonDocDBElasticReadOnlyAccess - New policy	New managed policy for Amazon DocumentDB elastic clusters.	6/8/2023
AmazonDocDBElasticFullAccess - New policy	New managed policy for Amazon DocumentDB elastic clusters.	6/5/2023
AmazonDocDB-ElasticServiceRolePolicy – New policy	Amazon DocumentDB creates a new AWSServiceRoleForDocDB-Elastic service linked role for Amazon DocumentDB elastic clusters.	11/30/2022
AmazonDocDBConsoleFullAccess - Change	Policy updated to add Amazon DocumentDB global	11/30/2022

Change	Description	Date
	and elastic cluster permissio ns.	
AmazonDocDBConsole FullAccess, AmazonDoc DBFullAccess, AmazonDoc DBReadOnlyAccess - New Policy	Service launch.	1/19/2017

Amazon DocumentDB API permissions: actions, resources, and conditions reference

Use the following sections as a reference when you set up Using identity-based policies (IAM policies) for Amazon DocumentDB and write permissions policies that you can attach to an IAM identity (identity-based policies).

The following lists each Amazon DocumentDB API operation. Included in the list are the corresponding actions for which you can grant permissions to perform the action, the AWS resource that you can grant the permissions for, and condition keys that you can include for fine-grained access control. You specify the actions in the policy's `Action` field, the resource value in the policy's `Resource` field, and conditions in the policy's `Condition` field. For more information about conditions, see Specifying conditions in a policy.

You can use AWS-wide condition keys in your Amazon DocumentDB policies to express conditions. For a complete list of AWS-wide keys, see Available Keys in the *IAM User Guide*.

You can test IAM policies with the IAM policy simulator. It automatically provides a list of resources and parameters required for each AWS action, including Amazon DocumentDB actions. The IAM policy simulator determines the permissions that are required for each of the actions that you specify. For information about the IAM policy simulator, see Testing IAM Policies with the IAM Policy Simulator in the *IAM User Guide*.

> **Note**
>
> To specify an action, use the `rds:` prefix followed by the API operation name (for example, `rds:CreateDBInstance`).

The following lists Amazon RDS API operations and their related actions, resources, and condition keys.

Topics

- Amazon DocumentDB actions that support resource-level permissions
- Amazon DocumentDB actions that don't support resource-level permissions

Amazon DocumentDB actions that support resource-level permissions

Resource-level permissions provide the ability to specify the resources on which users are allowed to perform actions. Amazon DocumentDB has partial support for resource-level permissions. This means that for certain Amazon DocumentDB actions, you can control when users are allowed to use those actions based on conditions that have to be fulfilled, or specific resources that users are allowed to use. For example, you can grant users permission to modify only specific instances.

The following lists Amazon DocumentDB API operations and their related actions, resources, and condition keys.

> **Note**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon RDS. For more Amazon DocumentDB actions and permissions, refer to Actions, resources, and condition keys for Amazon RDS in the *Service Authorization Reference*.

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
AddTagsToResource	Instance	`rds:db-tag`

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
rds:AddTagsToResource	arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	
	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
ApplyPendingMaintenanceAction rds:ApplyPendingMaintenanceAction	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:db-tag
CopyDBClusterSnapshot rds:CopyDBClusterSnapshot	Cluster snapshot arn:aws:rds: *region*:*account-id* :cluster-snapshot: *cluster-snapshot-name*	rds:cluster-snapshot-tag
CreateDBCluster rds:CreateDBCluster	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-name*	rds:cluster-tag
	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
CreateDBClusterParameterGroup rds:CreateDBClusterParameterGroup	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag
CreateDBClusterSnapshot rds:CreateDBClusterSnapshot	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-name*	rds:cluster-tag
	Cluster snapshot arn:aws:rds: *region*:*account-id* :cluster-snapshot: *cluster-snapshot-name*	rds:cluster-snapshot-tag
CreateDBInstance rds:CreateDBInstance	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:DatabaseClass rds:db-tag
	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-name*	rds:cluster-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
CreateDBSubnetGroup rds:CreateDBSubnetGroup	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
DeleteDBInstance rds:DeleteDBInstance	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:db-tag
DeleteDBSubnetGroup rds:DeleteDBSubnetGroup	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
DescribeDBClusterParameterGroups rds:DescribeDBClusterParameterGroups	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag
DescribeDBClusterParameters rds:DescribeDBClusterParameters	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
DescribeDBClusters rds:DescribeDBClusters	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-instance-name*	rds:cluster-tag
DescribeDBClusterSnapshotAttributes rds:DescribeDBClusterSnapshotAttributes	Cluster snapshot arn:aws:rds: *region*:*account-id* :cluster-snapshot: *cluster-snapshot-name*	rds:cluster-snapshot-tag
DescribeDBSubnetGroups rds:DescribeDBSubnetGroups	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
DescribePendingMaintenanceActions rds:DescribePendingMaintenanceActions	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:DatabaseClass rds:db-tag
FailoverDBCluster rds:FailoverDBCluster	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-instance-name*	rds:cluster-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
ListTagsForResource rds:ListTagsForResource	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:db-tag
	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
ModifyDBCluster rds:ModifyDBCluster	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-name*	rds:cluster-tag
	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag
ModifyDBClusterParameterGroup rds:ModifyDBClusterParameterGroup	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
ModifyDBClusterSnapshotAttribute rds:ModifyDBClusterSnapshotAttribute	Cluster snapshot arn:aws:rds: *region*:*account-id* :cluster-snapshot: *cluster-snapshot-name*	rds:cluster-snapshot-tag
ModifyDBInstance rds:ModifyDBInstance	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:DatabaseClass rds:db-tag
RebootDBInstance rds:RebootDBInstance	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:db-tag
RemoveTagsFromResource rds:RemoveTagsFromResource	Instance arn:aws:rds: *region*:*account-id* :db:*db-instance-name*	rds:db-tag
	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag
ResetDBClusterParameterGroup rds:ResetDBClusterParameterGroup	Cluster parameter group arn:aws:rds: *region*:*account-id* :cluster-pg: *cluster-parameter-group-name*	rds:cluster-pg-tag

Amazon DocumentDB API Operations and Actions	Resources	Condition Keys
RestoreDBClusterFromSnapshot rds:RestoreDBClusterFromSnapshot	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-instance-name*	rds:cluster-tag
	Cluster snapshot arn:aws:rds: *region*:*account-id* :cluster-snapshot: *cluster-snapshot-name*	rds:cluster-snapshot-tag
RestoreDBClusterToPointInTime rds:RestoreDBClusterToPointInTime	Cluster arn:aws:rds: *region*:*account-id* :cluster: *db-cluster-instance-name*	rds:cluster-tag
	Subnet group arn:aws:rds: *region*:*account-id* :subgrp:*subnet-group-name*	rds:subgrp-tag

Amazon DocumentDB actions that don't support resource-level permissions

You can use all Amazon DocumentDB actions in an IAM policy to either grant or deny users permission to use that action. However, not all Amazon DocumentDB actions support resource-level permissions, which enable you to specify the resources on which an action can be performed. The following Amazon DocumentDB API actions currently don't support resource-level permissions. Therefore, to use these actions in an IAM policy, you must grant users permission to use all resources for the action by using a * wildcard for the Resource element in your statement.

- rds:DescribeDBClusterSnapshots

- `rds:DescribeDBInstances`

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

Use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon DocumentDB gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:`*`servicename`*`:*:`*`123456789012`*`:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of `aws:SourceArn` must be cluster.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Amazon DocumentDB to prevent the confused deputy problem.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Sid": "ConfusedDeputyPreventionExamplePolicy",
    "Effect": "Allow",
    "Principal": {
```

```
      "Service": "servicename.amazonaws.com"
    },
    "Action": "servicename:ActionName",
    "Resource": [
      "arn:aws:servicename:::ResourceName/*"
    ],
    "Condition": {
      "ArnLike": {
        "aws:SourceArn": "arn:aws:servicename:*:123456789012:*"
      },
      "StringEquals": {
        "aws:SourceAccount": "123456789012"
      }
    }
  }
}
```

Authentication using IAM identity

Amazon DocumentDB users and applications can use IAM users and roles to authenticate into an Amazon DocumentDB cluster. Amazon DocumentDB IAM authentication is a password-less authentication method. Also, client applications do not send the password secrets to the Amazon DocumentDB cluster when using IAM roles/users. Instead, client connections are authenticated by AWS STS using temporary security tokens. Non-administrative users and applications can now use the same IAM identity ARN when connecting to different Amazon DocumentDB clusters and other AWS services.

You can also choose to use both password-based and IAM authentication to authenticate users and applications to an Amazon DocumentDB cluster. IAM authentication is available only in Amazon DocumentDB instance-based cluster version 5.0. IAM authentication using IAM identity ARNs is not supported for the Amazon DocumentDB primary user.

> **Note**
>
> The primary user can only be authenticated using existing password-based authentication.

Topics

- Getting started with authentication using IAM users and roles

- Configuring AWS compute types to authenticate to Amazon DocumentDB using AWS IAM
- Monitoring IAM authentication requests
- Using IAM authentication
- Drivers supporting IAM
- IAM identity authentication FAQ

Getting started with authentication using IAM users and roles

Amazon DocumentDB users and roles with IAM identities are created and managed in an `$external` database.

Creating a user

Connect as the primary user, then create an IAM user and role:

```
use $external;
db.createUser(
    {
        user: "arn:aws:iam::123456789123:user/iamuser",
        mechanisms: ["MONGODB-AWS"],
        roles: [ { role: "readWrite", db: "readWriteDB" } ]
    }
);
```

Alternatively, add an Amazon DocumentDB user using an IAM role:

```
use $external;
db.createUser(
    {
        user: "arn:aws:iam::123456789123:role/iamrole",
        mechanisms: ["MONGODB-AWS"],
        roles: [ { role: "readWrite", db: "readWriteDB" } ]
    }
);
```

Modifying an IAM user or role

Modify an existing IAM user:

```
use $external;
```

```
db.updateUser(
  "arn:aws:iam::123456789123:user/iamuser",
  {
    roles: [ { role: "read", db: "readDB" } ]
  }
);
```

Modify an existing IAM role:

```
use $external;
db.updateUser(
  "arn:aws:iam::123456789123:role/iamrole",
  {
    roles: [ { role: "read", db: "readDB" } ]
  }
);
```

To grant or revoke roles from an IAM user:

```
use $external;
db.grantRolesToUser(
  "arn:aws:iam::123456789123:user/iamuser",
  [ { db: "admin", role: "readWriteAnyDatabase" } ]
);
```

```
use $external;
db.revokeRolesFromUser(
  "arn:aws:iam::123456789123:user/iamuser",
  [ { db: "admin", role: "readWriteAnyDatabase" } ]
);
```

To grant or revoke roles from an IAM role:

```
use $external;
db.grantRolesToUser(
  "arn:aws:iam::123456789123:user/iamrole",
  [ { db: "admin", role: "readWriteAnyDatabase" } ]
);
```

```
use $external;
```

```
db.revokeRolesFromUser(
  "arn:aws:iam::123456789123:user/iamrole",
  [ { db: "admin", role: "readWriteAnyDatabase" } ]
);
```

Dropping an IAM user or role

To drop an existing IAM user:

```
use $external;
db.dropUser("arn:aws:iam::123456789123:user/iamuser");
```

To drop an existing IAM role:

```
use $external;
db.dropUser("arn:aws:iam::123456789123:role/iamrole");
```

Configure a connection URI to authenticate using AWS IAM

To authenticate using AWS IAM, use the following URI parameters: `authSource` as `$external` and `authMechanism` as `MONGODB-AWS`. If you are using an IAM user, the username and password fields are replaced by an Access Key and Secret Key respectively. If you are assuming an IAM role, attached to the environment you are in (for example, AWS Lambda function, Amazon EC2 instance). You don't need to specifically pass any credential when authenticating using the `MONGODB-AWS` mechanism. If you are using MongoDB drivers that support the `MONGODB-AWS` authentication mechanism, the drivers also have the ability to retrieve IAM role credentials from the compute instance (for example, Amazon EC2, Lambda function, and others). The following example uses a mongo shell to authenticate using `MONGODB-AWS` by passing an Access Key and Secret Key (of an IAM user) manually to demonstrate authentication against Amazon DocumentDB.

The following example uses Python code to authenticate using `MONGODB-AWS` without explicitly passing any credentials (using an IAM Role attached to the environment) to demonstrate authentication against Amazon DocumentDB.

```
##Create a MongoDB client, open a connection to Amazon DocumentDB using an IAM role
    client = pymongo.MongoClient('mongodb://<DocDBEndpoint>:27017/?
tls=true&tlsCAFile=global-
bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false&authSource=
%24external&authMechanism=MONGODB-AWS')
```

The following example uses a mongo shell to authenticate using MONGODB-AWS mechanism by passing an Access Key and Secret Key (of an IAM user) manually to demonstrate authentication against Amazon DocumentDB.

```
$ mongo 'mongodb://<access_key>:<secret_key>@<cluster_endpoint>:<db_port>/test?
authSource=%24external&authMechanism=MONGODB-AWS'
```

The following example uses a mongo shell to authenticate using MONGODB-AWS without explicitly passing any credentials (using IAM Role attached to the environment) to demonstrate authentication against Amazon DocumentDB.

```
$ mongo 'mongodb://<cluster_endpoint>:<db_port>/test?authSource=
%24external&authMechanism=MONGODB-AWS'
```

Configuring AWS compute types to authenticate to Amazon DocumentDB using AWS IAM

Using Amazon EC2/AWS Lambda/AWS Fargate

Amazon EC2 uses the following environment variables. If you have a IAM role attached to the EC2 instance or an execution IAM role associated with a Lambda function or an Amazon ECS task, then these variables are automatically populated and the driver can fetch these values from environment:

```
AWS_ACCESS_KEY_ID
AWS_SECRET_ACCESS_KEY
AWS_SESSION_TOKEN
```

For more information about environment variables, see Using Lambda environment variables in the *AWS Lambda Developer Guide*.

Using Amazon EKS

Assigning a role to your Amazon Elastic Kubernetes Service (Amazon EKS) pods will automatically setup the following two environment variables:

```
AWS_WEB_IDENTITY_TOKEN_FILE - path of web identity token file
AWS_ROLE_ARN - Name of IAM role to connect with
```

With the help of these variables, manually assume the role from your code using the AWS SDK call for `AssumeRoleWithWebIdentity`:

- Omit the `ProviderID` parameter.
- Find the value of the `WebIdentityToken` parameter in the file described in the `AWS_WEB_IDENTITY_TOKEN_FILE` environment variable.

For more info on Amazon EKS, see What is Amazon EKS in the *Amazon EKS User Guide*.

Monitoring IAM authentication requests

Using Amazon DocumentDB auditing

Go to the audit log folder in Amazon CloudWatch, and use different search patterns to get the logs for IAM authentication. For example, use `{ $.param.mechanism = "MONGODB-AWS" }` as the search pattern for “Search all log streams”.

For more info on supported events in auditing, see Auditing Amazon DocumentDB events.

Using Amazon CloudWatch metrics

`StsGetCallerIdentityCalls`: This metric shows how many `GetCallerIdentity` calls an Amazon DocumentDB instance is making to the regionalized AWS Security Token Service (AWS STS) endpoint. Refer to the `MONGODB-AWS` authentication specification on why database instances need to make STS `GetCallerIdentity` calls.

Using IAM authentication

When you don’t want to manage the username and password in your own database, you can use IAM authentication. IAM authentication is available only in Amazon DocumentDB instance-based cluster version 5.0.

IAM authentication has a dependency on AWS Security Token Service (AWS STS). If you get an AWS STS throttling exception when using IAM authentication, lower your connection rate.

For IAM quotas, see IAM and AWS STS quotas in the *IAM User Guide*.

Drivers supporting IAM

Drivers that support Amazon DocumentDB 5.0 and the `MONGODB-AWS` authentication mechanism should work with the IAM authentication implementation in Amazon DocumentDB.

Important

There is a known limitation with Node.js drivers older than version 6.13.1, which are currently not supported by IAM identity authentication for Amazon DocumentDB. Node.js drivers and tools that use Node.js driver (for example, mongosh) must be upgraded to use Node.js driver version 6.13.1 or above.

IAM identity authentication FAQ

Are there any samples I can refer to?

See these pages for sample use cases and configurations:

- How human users can authenticate to Amazon DocumentDB using IAM Users and IAM Roles
- Password-less authentication to Amazon DocumentDB using IAM Roles

I am getting an error while using my Python driver: "pymongo.errors.ConfigurationError: MONGODB-AWS authentication requires pymongo-auth-aws". How can I resolve this?

Make sure you use the following statement while installing the Python driver with IAM authentication:

```
pip install 'pymongo[aws]'
```

This will install the additional AWS dependencies required for IAM authentication to work.

Will my connection drop when my IAM role temporary credentials expire?

No, the temporary IAM credentials are only used for establishing connection and authentication. Then all further authorization happens in the Amazon DocumentDB cluster. Even if IAM credentials rotate/expire, the connection will not drop or get stale.

Authentication using SCRAM-SHA-256

Amazon DocumentDB (with MongoDB compatibility) supports the Salted Challenge Response Authentication Mechanism (SCRAM) for password-based authentication. With SCRAM, the server verifies that a client knows a user's password without transmitting the password to the server.

Amazon DocumentDB supports two SCRAM mechanisms: SCRAM-SHA-1 and SCRAM-SHA-256. SCRAM-SHA-256 uses the stronger SHA-256 hash function and works with FIPS mode.

You assign one or both SCRAM mechanisms to a user when you create or update that user. Users that were created with SCRAM-SHA-1 continue to work without any changes. SCRAM-SHA-256 authentication is available in Amazon DocumentDB instance-based cluster minor versions 5.0.1+ and 8.0.1+.

SCRAM and IAM authentication are mutually exclusive

A user can have either a SCRAM mechanism or the MONGODB-AWS mechanism, but not both. For IAM identity authentication, see Authentication using IAM identity.

Topics

- Creating users with SCRAM mechanisms
- Updating existing users
- Authenticating with SCRAM-SHA-256

Creating users with SCRAM mechanisms

When you create a user, you can specify the SCRAM mechanisms to enable with the mechanisms field. The following examples create users with SCRAM-SHA-1 only, SCRAM-SHA-256 only, and both mechanisms:

```
db.runCommand({
    createUser: 'sha1user',
    pwd: passwordPrompt(),
    roles: ['readWrite'],
    mechanisms: ['SCRAM-SHA-1']
});
db.runCommand({
    createUser: 'sha256user',
    pwd: passwordPrompt(),
    roles: ['readWrite'],
    mechanisms: ['SCRAM-SHA-256']
});
db.runCommand({
```

```
    createUser: 'bothuser',
    pwd: passwordPrompt(),
    roles: ['readWrite'],
    mechanisms: ['SCRAM-SHA-1','SCRAM-SHA-256']
});
```

Default mechanisms

If you don't specify the `mechanisms` field, Amazon DocumentDB enables both `SCRAM-SHA-1` and `SCRAM-SHA-256` for the user. The following two commands are equivalent:

```
db.runCommand({
    createUser: 'user1',
    pwd: passwordPrompt(),
    roles: ['readWrite']
});
// is equivalent to:
db.runCommand({
    createUser: 'user1',
    pwd: passwordPrompt(),
    roles: ['readWrite'],
    mechanisms: ['SCRAM-SHA-1','SCRAM-SHA-256']
});
```

Enabling both mechanisms by default maintains compatibility with older MongoDB drivers that support only `SCRAM-SHA-1`. Newer drivers and FIPS mode prefer `SCRAM-SHA-256`. If you don't want any user to authenticate with `SCRAM-SHA-1`, specify only `SCRAM-SHA-256` in the `mechanisms` field for all of your users.

Updating existing users

Users created with `SCRAM-SHA-1` continue to be supported with no changes. To switch an existing user from `SCRAM-SHA-1` to `SCRAM-SHA-256`, update the user with a new password and the `SCRAM-SHA-256` mechanism:

```
db.runCommand({
    updateUser: 'existing',
    pwd: passwordPrompt(),
    mechanisms: ['SCRAM-SHA-256']
});
```

Authenticating with SCRAM-SHA-256

Your driver automatically authenticates using the connection string parameters before it sends the first command on each connection. To select `SCRAM-SHA-256` in a connection URI, set the `authMechanism` parameter to `SCRAM-SHA-256`:

```
mongodb://username:password@hostname:port/authenticationDatabase?authMechanism=SCRAM-SHA-256
```

You can also send an explicit authentication request from mongosh:

```
db.auth({user: 'sha256user', pwd: passwordPrompt(), mechanism: 'SCRAM-SHA-256'})
Enter password:
1
```

If you don't specify an authentication mechanism in your connection options, the driver automatically selects a mechanism that the target database user supports.

Managing Amazon DocumentDB users

In Amazon DocumentDB, users authenticate to a cluster in conjunction with a password. Each cluster has primary sign-in credentials that are established during cluster creation.

> **Note**
>
> All new users created before **March 26, 2020** have been granted the `dbAdminAnyDatabase`, `readWriteAnyDatabase`, and `clusterAdmin` roles. It is recommended that you reevaluate all users and modify the roles as necessary to enforce least privilege for all users in your clusters.
> For more information, see Database access using Role-Based Access Control.

Primary and `serviceadmin` user

A newly created Amazon DocumentDB cluster has two users: the primary user and the `serviceadmin` user.

The *primary user* is a single, privileged user that can perform administrative tasks and create additional users with roles. When you connect to an Amazon DocumentDB cluster for the first

time, you must authenticate using the primary sign-in credentials. The primary user receives these administrative permissions for an Amazon DocumentDB cluster when that cluster is created, and is granted the role of `root`.

The `serviceadmin` user is created implicitly when the cluster is created. Every Amazon DocumentDB cluster has a `serviceadmin` user that provides AWS the ability to manage your cluster. You cannot log in as, drop, rename, change the password, or change the permissions for `serviceadmin`. Any attempt to do so results in an error.

> **Note**
>
> The primary and `serviceadmin` users for an Amazon DocumentDB cluster cannot be deleted and the role of `root` for the primary user cannot be revoked.
> If you forget your primary user password, you can reset it using the AWS Management Console or the AWS CLI.

Creating additional users

After you connect as the primary user (or any user that has the role `createUser`), you can create a new user, as shown below.

```
db.createUser(
    {
        user: "sample-user-1",
        pwd: "password123",
        roles:
            [{"db":"admin", "role":"dbAdminAnyDatabase" }]
    }
)
```

To view user details, you can use the `show users` command as follows. You can additionally remove users with the `dropUser` command. For more information, see Common commands.

```
show users
{
    "_id" : "serviceadmin",
    "user" : "serviceadmin",
```

```
    "db" : "admin",
    "roles" : [
     {
            "role" : "root",
            "db" : "admin"
        }
    ]
},

{
    "_id" : "myPrimaryUser",
    "user" : "myPrimaryUser",
    "db" : "admin",
    "roles" : [
     {
            "role" : "root",
            "db" : "admin"
        }
    ]
},

{
    "_id" : "sample-user-1",
    "user" : "sample-user-1",
    "db" : "admin",
    "roles" : [
     {
            "role" : "dbAdminAnyDatabase",
            "db" : "admin"
     }
    ]
}
```

In the example above, the new user `sample-user-1` is attributed to the `admin` database. This is always the case for a new user. Amazon DocumentDB does not have the concept of an `authenticationDatabase` and thus all authentication is performed in the context of the `admin` database.

When creating users, if you omit the db field when specifying the role, Amazon DocumentDB will implicitly attribute the role to the database in which the connection is being issued against. For example, if your connection is issued against the database `sample-database` and you run the

following command, the user `sample-user-2` will be created in the `admin` database and will have `readWrite` permissions to the database `sample-database`.

```
db.createUser(
    {
        user: "sample-user-2",
        pwd: "password123",
        roles:
            ["readWrite"]
    }
)
```

Creating users with roles that are scoped across all databases (for example, `readInAnyDatabase`) require that you are either in the context of the `admin` database when creating the user or you explicitly state the database for the role when creating the user.

To switch the context of your database, you can use the following command.

```
use admin
```

To learn more about Role Based Access Control and enforcing least privilege amongst the users in your cluster, see Database access using Role-Based Access Control.

Automatically rotating passwords for Amazon DocumentDB

With AWS Secrets Manager, you can replace hardcoded credentials in your code (including passwords) with an API call to Secrets Manager to retrieve the secret programmatically. This helps ensure that the secret can't be compromised by someone examining your code, because the secret simply isn't there. Also, you can configure Secrets Manager to automatically rotate the secret for you according to a schedule that you specify. This enables you to replace long-term secrets with short-term ones, which helps to significantly reduce the risk of compromise.

Using Secrets Manager, you can automatically rotate your Amazon DocumentDB passwords (that is, *secrets*) using an AWS Lambda function that Secrets Manager provides.

For more information about AWS Secrets Manager and native integration with Amazon DocumentDB, see the following:

- Blog: How to rotate Amazon DocumentDB and Amazon Redshift credentials in AWS Secrets Manager

- What is AWS Secrets Manager?
- Rotate AWS Secrets Manager secrets
- Amazon DocumentDB credentials in Secrets Manager

Database access using Role-Based Access Control

You can restrict access to the actions that users can perform on databases using *role-based access control* (RBAC) in Amazon DocumentDB (with MongoDB compatibility). RBAC works by granting one or more roles to a user. These roles determine the operations that a user can perform on database resources. Amazon DocumentDB currently supports both built-in roles that are scoped at the database level, such as `read`, `readWrite`, `readAnyDatabase`, `clusterAdmin`, and user-defined roles that can be scoped to specific actions and granular resources such as collections based on your requirements.

Common use cases for RBAC include enforcing least privileges by creating users with read-only access to the databases or collections in a cluster, and multi-tenant application designs that enable a single user to access a given database or collection in a cluster.

> **Note**
>
> All new users created before **March 26, 2020** have been granted the `dbAdminAnyDatabase`, `readWriteAnyDatabase`, and `clusterAdmin` roles. It is recommended that you reevaluate all existing users and modify the roles as necessary to enforce least privileges for your clusters.

Topics

- RBAC concepts
- Getting started with RBAC built-in roles
- Getting started with RBAC user-defined roles
- Connecting to Amazon DocumentDB as a User
- Common commands
- Functional differences
- Limits
- Database access using Role-Based Access Control

RBAC concepts

The following are important terms and concepts related to role-based access control. For more information on Amazon DocumentDB users, see Managing Amazon DocumentDB users.

- **User** — An individual entity that can authenticate to the database and perform operations.
- **Password** — A secret that is used to authenticate the user.
- **Role** — Authorizes a user to perform actions on one or more databases.
- **Admin Database** — The database in which users are stored and authorized against.
- **Database (db)** — The namespace within clusters that contains collections for storing documents.

The following command creates a user named `sample-user`.

```
db.createUser({user: "sample-user", pwd: "abc123", roles: [{role: "read", db: "sample-database"}]})
```

In this example:

- `user: "sample-user"` — Indicates the user name.
- `pwd: "abc123"` — Indicates the user password.
- `role: "read", "db: "sample-database"` — Indicates that the user `sample-user` will have read permissions in `sample-database`.



The following example shows the output after you get the user `sample-user` with `db.getUser(sample-user)`. In this example, the user `sample-user` resides in the `admin` database but has the read role for the database `sample-database`.



When creating users, if you omit the db field when specifying the role, Amazon DocumentDB will implicitly attribute the role to the database in which the connection is being issued against. For example, if your connection is issued against the database `sample-database` and you run the following command, the user `sample-user` will be created in the `admin` database and will have `readWrite` permissions to the database `sample-database`.

```
db.createUser({user: "sample-user", pwd: "abc123", roles: ["readWrite"]})
```

Output from this operation looks something like the following.

```
{
   "user":"sample-user",
   "roles":[
      {
         "db":"sample-database",
         "role":"readWrite"
      }
   ]
}
```

Creating users with roles that are scoped across all databases (for example, `readAnyDatabase`) require that you either be in the context of the `admin` database when creating the user, or you explicitly state the database for the role when creating the user. To issue commands against the `admin` database, you can use the command `use admin`. For more information, see Common commands.

Getting started with RBAC built-in roles

To help you get started with role-based access control, this section walks you through an example scenario of enforcing least privilege by creating roles for three users with different job functions.

- `user1` is a new manager that needs to be able to view and access all databases in a cluster.
- `user2` is a new employee that needs access to only one database, `sample-database-1`, in that same cluster.
- `user3` is an existing employee that needs to view and access a different database, `sample-database-2` that they didn't have access to before, in the same cluster.

At a point later, both `user1` and `user2` leave the company and so their access must be revoked.

To create users and grant roles, the user that you authenticate to the cluster with must have an associated role that can perform actions for `createUser` and `grantRole`. For example, the roles `admin` and `userAdminAnyDatabase` can both grant such abilities, for example. For actions per role, see Database access using Role-Based Access Control.

> **Note**
>
> In Amazon DocumentDB, all user and role operations (for example, `create`, `get`, `drop`, `grant`, `revoke`, etc.) are implicitly performed in the `admin` database whether or not you are issuing commands against the `admin` database.

First, to understand what the current users and roles are in the cluster, you can run the `show users` command, as in the following example. You will see two users, `serviceadmin` and the primary user for the cluster. These two users always exist and cannot be deleted. For more information, see Managing Amazon DocumentDB users.

```
show users
```

For `user1`, create a role with read and write access to all databases in the entire cluster with the following command.

```
db.createUser({user: "user1", pwd: "abc123", roles: [{role: "readWriteAnyDatabase", db:
 "admin"}]})
```

Output from this operation looks something like the following.

```
{
   "user":"user1",
   "roles":[
      {
         "role":"readWriteAnyDatabase",
         "db":"admin"
      }
   ]
}
```

For `user2`, create a role with read-only access to the database `sample-database-1` with the following command.

```
db.createUser({user: "user2", pwd: "abc123", roles: [{role: "read", db: "sample-
database-1"}]})
```

Output from this operation looks something like the following.

```
{
   "user":"user2",
   "roles":[
      {
         "role":"read",
         "db":"sample-database-1"
      }
   ]
}
```

To simulate the scenario that `user3` is an existing user, first create the user `user3`, and then assign a new role to `user3`.

```
db.createUser({user: "user3", pwd: "abc123", roles: [{role: "readWrite", db: "sample-
database-1"}]})
```

Output from this operation looks something like the following.

```
{
   "user":"user3",
   "roles":[
```

```
      {
         "role":"readWrite",
         "db":"sample-database-1"
      }
   ]
}
```

Now that the user `user3` has been created, assign `user3` the role `read` to `sample-database-2`.

```
db.grantRolesToUser("user3", [{role: "read", db: "sample-database-2"}])
```

Lastly, both `user1` and `user2` leave the company and need their access to the cluster revoked. You can do this by dropping the users, as follows.

```
db.dropUser("user1")
db.dropUser("user2")
```

To ensure that all users have the appropriate roles, you can list all users with the following command.

```
show users
```

Output from this operation looks something like the following.

```
{
   "_id":"serviceadmin",
   "user":"serviceadmin",
   "db":"admin",
   "roles":[
      {
         "db":"admin",
         "role":"root"
      }
   ]
}
{
   "_id":"master-user",
   "user":"master-user",
   "db":"admin",
   "roles":[
      {
```

```
         "db":"admin",
         "role":"root"
      }
   ]
}
{
   "_id":"user3",
   "user":"user3",
   "db":"admin",
   "roles":[
      {
         "db":"sample-database-2",
         "role":"read"
      },
      {
         "db":"sample-database-1",
         "role":"readWrite"
      }
   ]
}
```

Getting started with RBAC user-defined roles

To help you get started with user-defined roles, this section walks you through an example scenario of enforcing least privilege by creating roles for three users with different job functions.

In this example, the following applies:

- `user1` is a new manager that needs to be able to view and access all databases in a cluster.
- `user2` is a new employee that needs only the 'find' action to only one database, `sample-database-1`, in that same cluster.
- `user3` is an existing employee that needs to view and access a specific collection, col2 in a different database, `sample-database-2` that they didn't have access to before, in the same cluster.
- For `user1`, create a role with read and write access to all databases in the entire cluster with the following command.

```
db.createUser(
 {
     user: "user1", pwd: "abc123",
```

```
     roles: [{role: "readWriteAnyDatabase", db: "admin"}]
 }
 )
```

Output from this operation looks something like the following.

```
{
   "user":"user1",
   "roles":[
      {
         "role":"readWriteAnyDatabase",
         "db":"admin"
      }
   ]
}
```

For `user2`, create a role with 'find' privileges to all collections in the database `sample-database-1` with the following command. Note that this role would ensure that any associated users can only run find queries.

```
db.createRole(
{
    role: "findRole",
    privileges: [
    {
        resource: {db: "sample-database-1", collection: ""}, actions: ["find"]
    }],
    roles: []
}
)
```

Output from this operation looks something like the following.

```
{
   "role":"findRole",
   "privileges":[
      {
         "resource":{
            "db":"sample-database-1",
            "collection":""
         },
         "actions":[
```

```
            "find"
         ]
      }
   ],
   "roles":[

   ]
}
```

Next, create the user (`user2`) and attach the recently created role `findRole` to the user.

```
db.createUser(
{
    user: "user2",
    pwd: "abc123",
    roles: []
})

 db.grantRolesToUser("user2",["findRole"])
```

To simulate the scenario that `user3` is an existing user, first create the user `user3`, and then create a new role called collectionRole which we will in the next step assing to `user3`.

Now you can assign a new role to `user3`. This new role will allow `user3` to be able to insert, update, delete and find access to one specific collection col2 in `sample-database-2`.

```
db.createUser(
{
    user: "user3",
    pwd: "abc123",
    roles: []
})

db.createRole(
{
    role: "collectionRole",
    privileges: [
    {
        resource: {db: "sample-database-2", collection: "col2"}, actions: ["find",
 "update", "insert", "remove"]
    }],
    roles: []
```

```
}
)
```

Output from this operation looks something like the following.

```
{
   "role":"collectionRole",
   "privileges":[
      {
         "resource":{
            "db":"sample-database-2",
            "collection":"col2"
         },
         "actions":[
            "find",
            "update",
            "insert",
            "remove"
         ]
      }
   ],
   "roles":[

   ]
}
```

Now that the user `user3` has been created, you can grant `user3` the role `collectionFind`.

```
db.grantRolesToUser("user3",["collectionRole"])
```

Lastly, both `user1` and `user2` leave the company and need their access to the cluster revoked. You can do this by dropping the users, as follows.

```
db.dropUser("user1")
db.dropUser("user2")
```

To ensure that all users have the appropriate roles, you can list all users with the following command.

```
show users
```

Output from this operation looks something like the following.

```
{
   "_id":"serviceadmin",
   "user":"serviceadmin",
   "db":"admin",
   "roles":[
      {
         "db":"admin",
         "role":"root"
      }
   ]
}
{
   "_id":"master-user",
   "user":"master-user",
   "db":"admin",
   "roles":[
      {
         "db":"admin",
         "role":"root"
      }
   ]
}
{
   "_id":"user3",
   "user":"user3",
   "db":"admin",
   "roles":[
      {
         "db":"admin",
         "role":"collectionRole"
      }
   ]
}
```

Connecting to Amazon DocumentDB as a User

When connecting to an Amazon DocumentDB cluster, you connect in the context of a particular database. By default, if you don't specify a database in your connection string, you are automatically connected to the cluster in the context of the `test` database. All collection level commands like `insert` and `find` are issued against collections in the `test` database.

To see the database you are in the context of or — in other words — issuing commands against, use the db command in the mongo shell, as follows.

Query:

```
db
```

Output:

```
test
```

Although the default connection might be in the context of the `test` database, that does not necessarily mean that the user associated with the connection is authorized to perform actions on the `test` database. In the preceding example scenario, if you authenticate as the user `user3`, which has the `readWrite` role for the `sample-database-1` database, the default context of your connection is the `test` database. However, if you try to insert a document into a collection on the `test` database, you will receive an *Authorization failure* error message. This is because that user is not authorized to perform that command on that database, as shown below.

Query:

```
db
```

Output:

```
test
```

Query:

```
db.col.insert({x:1})
```

Output:

```
WriteCommandError({ "ok" : 0, "code" : 13, "errmsg" : "Authorization failure" })
```

If you change the context of your connection to the `sample-database-1` database, you can write to the collection for which the user has the authorization to do so.

Query:

```
use sample-database-1
```

Output:

```
switched to db sample-database-1
```

Query:

```
db.col.insert({x:1})
```

Output:

```
WriteResult({ "nInserted" : 1})
```

When you authenticate to a cluster with a particular user, you can also specify the database in the connection string. Doing so removes the necessity to perform the `use` command after the user has been authenticated to the `admin` database.

The following connection string authenticates the user against the `admin` database, but the context of the connection will be against the `sample-database-1` database.

```
mongo "mongodb://user3:abc123@sample-cluster.node.us-east-1.docdb.amazonaws.com:27017/
sample-database-2"
```

Common commands

This section provides examples of common commands using role-based access control in Amazon DocumentDB. You must be in the context of the `admin` database to create and modify users and roles. You can use the `use admin` command to switch to the `admin` database.

> **Note**
>
> Modifications to the users and roles will implicitly occur in the `admin` database. Creating users with roles that are scoped across all databases (for example, `readAnyDatabase`) requires that you are either in the context of the `admin` database (that is, `use admin`) when creating the user, or you explicitly state the database for the role when creating the user (as shown in Example 2 in this section).

Example 1: Create a user with `read` role for the database `foo`.

```
db.createUser({user: "readInFooBar", pwd: "abc123", roles: [{role: "read", db:
 "foo"}]})
```

Output from this operation looks something like the following.

```
{
   "user":"readInFooBar",
   "roles":[
      {
         "role":"read",
         "db":"foo"
      }
   ]
}
```

Example 2: Create a user with read access on all databases.

```
db.createUser({user: "readAllDBs", pwd: "abc123", roles: [{role: "readAnyDatabase", db:
 "admin"}]})
```

Output from this operation looks something like the following.

```
{
   "user":"readAllDBs",
   "roles":[
      {
         "role":"readAnyDatabase",
         "db":"admin"
      }
   ]
}
```

Example 3: Grant `read` role to an existing user on a new database.

```
db.grantRolesToUser("readInFooBar", [{role: "read", db: "bar"}])
```

Example 4: Update a user's role.

```
db.updateUser("readInFooBar", {roles: [{role: "read", db: "foo"}, {role: "read", db:
 "baz"}]})
```

Example 5: Revoke access to a database for a user.

```
db.revokeRolesFromUser("readInFooBar", [{role: "read", db: "baz"}])
```

Example 6: Describe a built-in role.

```
db.getRole("read", {showPrivileges:true})
```

Output from this operation looks something like the following.

```
{
   "role":"read",
   "db":"sample-database-1",
   "isBuiltin":true,
   "roles":[

   ],
   "inheritedRoles":[

   ],
   "privileges":[
      {
         "resource":{
            "db":"sample-database-1",
            "collection":""
         },
         "actions":[
            "changeStream",
            "collStats",
            "dbStats",
            "find",
            "killCursors",
            "listCollections",
            "listIndexes"
         ]
      }
   ],
   "inheritedPrivileges":[
      {
```

```
         "resource":{
            "db":"sample-database-1",
            "collection":""
         },
         "actions":[
            "changeStream",
            "collStats",
            "dbStats",
            "find",
            "killCursors",
            "listCollections",
            "listIndexes"
         ]
      }
   }
```

Example 7: Drop a user from the cluster.

```
db.dropUser("readInFooBar")
```

Output from this operation looks something like the following.

```
true
```

Example 8: Create a role with read and write access to a specific collection

```
db.createRole(
{
    role: "collectionRole",
    privileges: [
    {
        resource: {db: "sample-database-2", collection: "col2"}, actions: ["find",
 "update", "insert", "remove"]
    }],
    roles: []
}
)
```

Output from this operation looks something like the following.

```
{
```

```
   "role":"collectionRole",
   "privileges":[
      {
         "resource":{
            "db":"sample-database-2",
            "collection":"col2"
         },
         "actions":[
            "find",
            "update",
            "insert",
            "remove"
         ]
      }
   ],
   "roles":[

   ]
}
```

Example 9: Create a user and assign a user defined role

```
db.createUser(
{
    user: "user3",
    pwd: "abc123",
    roles: []
})

db.grantRolesToUser("user3",["collectionRole"])
```

Example 10: Grant additional privileges to a user defined role

```
db.grantPrivilegesToRole(
  "collectionRole",
  [
    {
      resource: { db: "sample-database-1", collection: "col1" },
      actions: ["find", "update", "insert", "remove"]
    }
  ]
)
```

Example 11: Remove privileges from a user defined role

```
db.revokePrivilegesFromRole(
  "collectionRole",
  [
    {
      resource: { db: "sample-database-1", collection: "col2" },
      actions: ["find", "update", "insert", "remove"]
    }
  ]
)
```

Example 12: Update an existing user defined role

```
db.updateRole(
  "collectionRole",
  {
    privileges: [
    {
        resource: {db: "sample-database-3", collection: "sample-collection-3"},
 actions: ["find", "update", "insert", "remove"]
    }],
    roles: []
  }
)
```

Functional differences

In Amazon DocumentDB, user and role definitions are stored in the `admin` database and users are authenticated against the `admin` database. This functionality differs from the MongoDB Community Edition, but is consistent with MongoDB Atlas.

Amazon DocumentDB also supports change streams, which provide a time-ordered sequence of change events that occur within your cluster's collections. The `listChangeStreams` action is applied at the cluster level (that is, across all databases), and the `modifyChangeStreams` action can be applied at the database level and cluster level.

Limits

The following table contains the limits for Role-Based Access Control in Amazon DocumentDB.

Description	Limit
Number of users per cluster	1000
Number of roles associated with a user	1000
Number of user-defined roles	100
Number of resources associated with a privilege	100

Database access using Role-Based Access Control

With role-based access control, you can create a user and grant it one or more roles to determine what operations that user can perform in a database or cluster.

The following is a list of built-in roles that are currently supported in Amazon DocumentDB.

> **Note**
>
> In Amazon DocumentDB 4.0 and 5.0, the `ListCollection` and `ListDatabase` commands can optionally use the `authorizedCollections` and `authorizedDatabases` parameters to list the collections and databases that the user has permission to access with requiring the `listCollections` and `listDatabase` roles, respectively. Also, users now have the ability to kill their own cursors without requiring the `KillCursor` role.

Database user

Role name	Description	Actions

`read`	Grants a user read access to the specified database.	`changeStreams`
		`collStats`
		`dbStats`
		`find`

Role name	Description	Actions
		killCursors listIndexes listCollections
readWrite	Grants the user read and write access to the specified database.	All actions from read permissions. createCollection dropCollection createIndex dropIndex insert killCursors listIndexes listCollections remove update

Cluster user

Role name	Description	Actions
readAnyDatabase	Grants a user read access to all databases in the cluster.	All actions from read permissions.

Role name	Description	Actions
		`listChangeStreams` `listDatabases`

`readWriteAnyDatabase`	Grants a user read and write access to all databases in the cluster.	All actions from `readWrite` permissions.
		`listChangeStreams`
		`listDatabases`

`userAdminAnyDatabase`	Grants a user the ability to assign and modify the roles or privileges any user has to the specified database.	`changeCustomData`
		`changePassword`
		`createUser`
		`dropRole`
		`dropUser`
		`grantRole`
		`listDatabases`
		`revokeRole`
		`viewRole`
		`viewUser`

Role name	Description	Actions
dbAdminAnyDatabase	Grants a user the ability to perform database administration roles on any specified database.	All actions from dbAdmin permissions. dropCollection listDatabases listChangeStreams modifyChangeStreams

Superuser

Role name	Description	Actions
root	Grants a user access to the resources and operations of all the following roles combined: readWriteAnyDatabase , dbAdminAnyDatabase , userAdminAnyDatabase , clusterAdmin , restore, and backup.	All actions from readWriteAnyDatabase , dbAdminAnyDatabase , userAdminAnyDatabase , clusterAdmin , restore, and backup.

Database administrator

Role name	Description	Actions
dbAdmin	Grants a user the ability to perform administrative tasks on the specified database.	bypassDocumentValidation collMod collStats createCollection createIndex dropCollection dropDatabase dropIndex dbStats find killCursors listIndexes listCollections modifyChangeStreams
dbOwner	Grants a user the ability to perform any administrative tasks on the specified database by combining the roles dbAdmin and readWrite .	All actions from dbAdmin and readWrite .

Cluster administrator

role Name	Description	Actions

`clusterAdmin`	Grants a user the greatest cluster management access by combining the `clusterManager` , `clusterMonitor` , and `hostManager` roles.	All actions from `clusterManager` , `clusterMonitor` , and `hostManager` .
		`listChangeStreams`
		`dropDatabase`
		`modifyChangeStreams`

`clusterManager`	Grants a user the ability to take management and monitoring actions on the specified cluster.	`listChangeStreams`
		`listSessions`
		`modifyChangeStreams`
		`replSetGetConfig`

`clusterMonitor`	Grants a user the ability to have read-only access to monitoring tools.	`collStats`
		`dbStats`
		`find`
		`getParameter`
		`hostInfo`
		`indexStats`
		`killCursors`

role Name	Description	Actions

		`listChangeStreams`
		`listCollections`
		`listDatabases`
		`listIndexes`
		`listSessions`
		`replSetGetConfig`
		`serverStatus`
		`top`

`hostManager`	Grants a user the ability to monitor and manage servers.	`auditConfigure`
		`killCursors`
		`killAnyCursor`
		`killAnySession`
		`killop`

Backup administrator

Role name	Description	Actions

`backup`	Grants a user the access needed to back up data.	`getParameter`
		`insert`
		`find`

Role name	Description	Actions

		`listChangeStreams`
		`listCollections`
		`listDatabases`
		`listIndexes`
		`update`

Role name	Description	Actions
restore	Grants a user the access needed to restore data.	bypassDocumentValidation changeCustomData changePassword collMod createCollection createIndex createUser dropCollection dropRole dropUser getParameter grantRole find insert listCollections modifyChangeStreams revokeRole

Role name	Description	Actions

		`remove`
		`viewRole`
		`viewUser`
		`update`

Updating your Amazon DocumentDB TLS certificates

Topics

- Updating your application and Amazon DocumentDB cluster
- Automatic server certificate rotation
- Troubleshooting
- Frequently Asked Questions

The certificate authority (CA) certificate for Amazon DocumentDB clusters was updated in August of 2024. If you are using Amazon DocumentDB clusters with Transport Layer Security (TLS) enabled (the default setting) and you have not rotated your client application and server certificates, the following steps are required to mitigate connectivity issues between your application and your Amazon DocumentDB clusters.

- Step 1: Download the new CA certificate and update your application
- Step 2: Update the server certificate

The CA and server certificates were updated as part of standard maintenance and security best practices for Amazon DocumentDB. Client applications must add the new CA certificates to their trust stores, and existing Amazon DocumentDB instances must be updated to use the new CA certificates before this expiration date.

Updating your application and Amazon DocumentDB cluster

Follow the steps in this section to update your application's CA certificate bundle (Step 1) and your cluster's server certificates (Step 2). Before you apply the changes to your production

environments, we strongly recommend testing these steps in a development or staging environment.

> **Note**
>
> You must complete Steps 1 and 2 in each AWS Region in which you have Amazon DocumentDB clusters.

Step 1: Download the new CA certificate and update your application

Download the new CA certificate and update your application to use the new CA certificate to create TLS connections to Amazon DocumentDB. Download the new CA certificate bundle from https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem. This operation downloads a file named `global-bundle.pem`.

> **Note**
>
> If you are accessing the keystore that contains both the old CA certificate (`rds-ca-2019-root.pem`) and the new CA certificates (`rds-ca-rsa2048-g1`, `rds-ca-rsa4096-g1`, `rds-ca-ecc384-g1`), verify that the keystore selects `global-bundle`.

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

Next, update your applications to use the new certificate bundle. The new CA bundle contains both the old CA certificate (rds-ca-2019) and the new CA certificates (rds-ca-rsa2048-g1, rds-ca-rsa4096-g1, rds-ca-ecc384-g1). By having both CA certificates in the new CA bundle, you can update your application and cluster in two steps.

For Java applications, you must create a new trust store with the new CA certificate. For instructions, see the Java tab in the Connecting with TLS enabled topic.

To verify that your application is using the latest CA certificate bundle, see How can I be sure that I'm using the newest CA bundle? If you're already using the latest CA certificate bundle in your application, you can skip to Step 2.

For examples of using a CA bundle with your application, see Encrypting data in transit and Connecting with TLS enabled.

> **Note**
>
> Currently, the MongoDB Go Driver 1.2.1 only accepts one CA server certificate in `sslcertificateauthorityfile`. See Connecting with TLS enabled for connecting to Amazon DocumentDB using Go when TLS is enabled.

Step 2: Update the server certificate

After the application has been updated to use the new CA bundle, the next step is to update the server certificate by modifying each instance in an Amazon DocumentDB cluster. To modify instances to use the new server certificate, see the following instructions.

Amazon DocumentDB provides the following CAs to sign the DB server certificate for a DB instance:

- **rds-ca-ecc384-g1**—Uses a certificate authority with ECC 384 private key algorithm and SHA384 signing algorithm. This CA supports automatic server certificate rotation. This is only supported on Amazon DocumentDB 4.0 and 5.0.
- **rds-ca-rsa2048-g1**—Uses a certificate authority with RSA 2048 private key algorithm and SHA256 signing algorithm in most AWS Regions. This CA supports automatic server certificate rotation.
- **rds-ca-rsa4096-g1**—Uses a certificate authority with RSA 4096 private key algorithm and SHA384 signing algorithm. This CA supports automatic server certificate rotation.

> **Note**
>
> If you are using the AWS CLI, you can see the validities of the certificate authorities listed in the preceding section by using describe-certificates.

These CA certificates are included in the regional and global certificate bundle. When you use the rds-ca-rsa2048-g1, rds-ca-rsa4096-g1, or rds-ca-ecc384-g1 CA with a database, Amazon DocumentDB manages the DB server certificate on the database. Amazon DocumentDB rotates the DB server certificate automatically before it expires.

Note

Amazon DocumentDB does not require a reboot for certificate rotation if your cluster is running on the following engine patch versions:

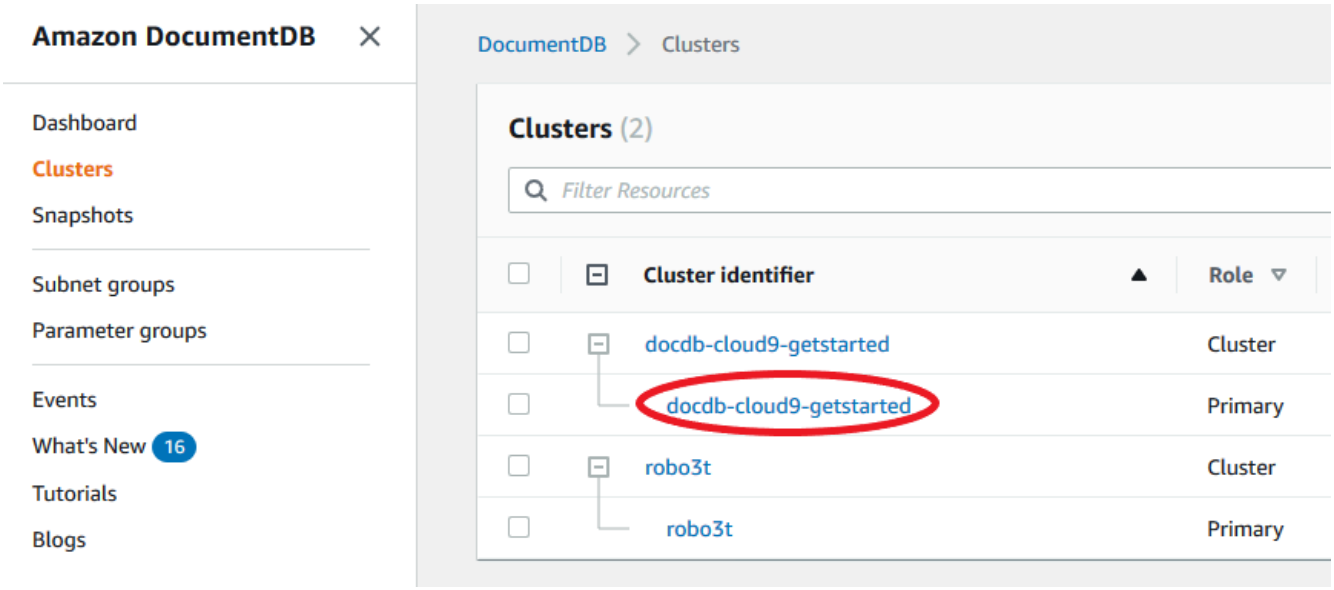
- Amazon DocumentDB 3.6: 1.0.208662 or greater
- Amazon DocumentDB 4.0: 2.0.10179 or greater
- Amazon DocumentDB 5.0: 3.0.4780 or greater

You can determine the current Amazon DocumentDB engine patch version by running the following command: `db.runCommand({getEngineVersion: 1})`.
Before updating the server certificate, ensure that you have completed Step 1.

Using the AWS Management Console

Complete the following steps to identify and rotate the old server certificate for your existing Amazon DocumentDB instances using the AWS Management Console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the list of Regions in the upper-right corner of the screen, choose the AWS Region in which your clusters reside.
3. In the navigation pane on the left side of the console, choose **Clusters**.
4. You may need to identify which instances are still on the old server certificate (`rds-ca-2019`). You can do this in the **Certificate authority** column which is located on the far right of the **Clusters** table.
5. In the **Clusters** table, you'll see the column **Cluster identifier** on the far left. Your instances are listed under clusters, similar to the following screenshot.



6. Check the box to the left of the instance you are interested in.
7. Choose **Actions** and then choose **Modify**.
8. Under **Certificate authority**, select the new server certificate (`rds-ca-rsa2048-g1`) for this instance.
9. You can see a summary of the changes on the next page. Note that there is an extra alert to remind you to ensure that your application is using the latest certificate CA bundle before modifying the instance to avoid causing an interruption in connectivity.
10. You can choose to apply the modification during your next maintenance window or apply immediately. If your intention is to modify the server certificate immediately, use the **Apply Immediately** option.
11. Choose **Modify instance** to complete the update.

Using the AWS CLI

Complete the following steps to identify and rotate the old server certificate for your existing Amazon DocumentDB instances using the AWS CLI.

1. To modify the instances immediately, execute the following command for each instance in the cluster.

```
aws docdb modify-db-instance --db-instance-identifier <yourInstanceIdentifier>
 --ca-certificate-identifier rds-ca-rsa2048-g1 --apply-immediately
```

2. To modify the instances in your clusters to use the new CA certificate during your cluster's next maintenance window, execute the following command for each instance in the cluster.

```
aws docdb modify-db-instance --db-instance-identifier <yourInstanceIdentifier>
 --ca-certificate-identifier rds-ca-rsa2048-g1 --no-apply-immediately
```

Automatic server certificate rotation

Amazon DocumentDB supports automatic server certificate rotation. The server certificate is the leaf certificate issued to each cluster instance. As opposed to the root CA certificates, the server certificates have short (12 months) validity and Amazon DocumentDB automatically handles their rotation without any action from you. Amazon DocumentDB uses the same root CA for this automatic rotation, so you don't need to download a new CA bundle.

> **Important**
>
> When connecting to your Amazon DocumentDB cluster, trust the root CA bundle instead of trusting each server certificate directly. This will prevent connection errors after the server certificate is rotated. See Connecting with TLS enabled.

Amazon DocumentDB attempts to rotate your server certificate in your preferred maintenance window at the server certificate half life. The new server certificate is valid for 12 months.

Use the `describe-db-engine-versions` command and inspect the `SupportsCertificateRotationWithoutRestart` flag to identify whether the engine version supports rotating the certificate without restart.

> **Note**
>
> Amazon DocumentDB supports server certificate rotation without restarts if your cluster is running on the following engine patch versions:
>
> - Amazon DocumentDB 3.6: 1.0.208662 or greater
> - Amazon DocumentDB 4.0: 2.0.10179 or greater
> - Amazon DocumentDB 5.0: 3.0.4780 or greater

You can determine the current Amazon DocumentDB engine patch version by running this command: `db.runCommand({getEngineVersion: 1})`.
If you are using an older engine patch version, Amazon DocumentDB will rotate the server certificate and schedule a database restart event on your preferred maintenance window.

Troubleshooting

If you are having issues connecting to your cluster as part of the certificate rotation, try the following:

- **Verify that your clients are using the latest certificate bundle.** See How can I be sure that I'm using the newest CA bundle?.
- **Verify that your instances are using the latest certificate.** See How do I know which of my Amazon DocumentDB instances are using the old/new server certificate?.
- **Verify that the latest certificate CA is being utilized by your application.** Some drivers, like Java and Go, require extra code to import multiple certificates from a certificate bundle to the trust store. For more information on connecting to Amazon DocumentDB with TLS, see Connecting programmatically to Amazon DocumentDB.
- **Contact support.** If you have questions or issues, contact Support.

Frequently Asked Questions

The following are answers to some common questions about TLS certificates.

What if I have questions or issues?

If you have questions or issues, contact Support.

How do I know whether I'm using TLS to connect to my Amazon DocumentDB cluster?

You can determine whether your cluster is using TLS by examining the `tls` parameter for your cluster's cluster parameter group. If the `tls` parameter is set to `enabled`, you are using the TLS certificate to connect to your cluster. For more information, see Managing Amazon DocumentDB cluster parameter groups.

Why are you updating the CA and server certificates?

The Amazon DocumentDB CA and server certificates are being updated as part of standard maintenance and security best practices for Amazon DocumentDB.

What happens if I don't take any action by the expiration date?

If you are using TLS to connect to your Amazon DocumentDB cluster with an expired CA certificate, your applications that connect via TLS will no longer be able to communicate with the Amazon DocumentDB cluster.

Amazon DocumentDB will not rotate your database certificates automatically before expiration. You must update your applications and clusters to use the new CA certificates before or after the expiration date.

How do I know which of my Amazon DocumentDB instances are using the old/new server certificate?

To identify the Amazon DocumentDB instances that still use the old server certificate, you can use either the Amazon DocumentDB AWS Management Console or the AWS CLI.

Using the AWS Management Console

To identify the instances in your clusters that are using the older certificate

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the list of Regions in the upper-right corner of the screen, choose the AWS Region in which your instances reside.
3. In the navigation pane on the left side of the console, choose **Clusters**.
4. The **Certificate authority** column (near the far right of the table) shows which instances are still on the old server certificate (`rds-ca-2019`) and the new server certificate (`rds-ca-rsa2048-g1`).

Using the AWS CLI

To identify the instances in your clusters that are using the older server certificate, use the `describe-db-clusters` command with the following .

```
aws docdb describe-db-instances \
    --filters Name=engine,Values=docdb \
    --query 'DBInstances[*].
{CertificateVersion:CACertificateIdentifier,InstanceID:DBInstanceIdentifier}'
```

How do I modify individual instances in my Amazon DocumentDB cluster to update the server certificate?

Update server certificates for all instances in a given cluster at the same time. To modify the instances in your cluster, you can use either the console or the AWS CLI.

> **Note**
>
> Before updating the server certificate, ensure that you have completed Step 1.

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the list of Regions in the upper-right corner of the screen, choose the AWS Region in which your clusters reside.
3. In the navigation pane on the left side of the console, choose **Clusters**.
4. The **Certificate authority** column (near the far right of the table) shows which instances are still on the old server certificate (`rds-ca-2019`).
5. In the **Clusters** table, under **Cluster identifier**, select an instance to modify.
6. Choose **Actions** and then choose **Modify**.
7. Under **Certificate authority**, select the new server certificate (`rds-ca-rsa2048-g1`) for this instance.
8. You can see a summary of the changes on the next page. Note that there is an extra alert to remind you to ensure that your application is using the latest certificate CA bundle before modifying the instance to avoid causing an interruption in connectivity.
9. You can choose to apply the modification during your next maintenance window or apply immediately.

10. Choose **Modify instance** to complete the update.

Using the AWS CLI

Complete the following steps to identify and rotate the old server certificate for your existing Amazon DocumentDB instances using the AWS CLI.

1. To modify the instances immediately, execute the following command for each instance in the cluster.

```
aws docdb modify-db-instance --db-instance-identifier <yourInstanceIdentifier> --
ca-certificate-identifier rds-ca-rsa2048-g1 --apply-immediately
```

2. To modify the instances in your clusters to use the new CA certificate during your cluster's next maintenance window, execute the following command for each instance in the cluster.

```
aws docdb modify-db-instance --db-instance-identifier <yourInstanceIdentifier> --
ca-certificate-identifier rds-ca-rsa2048-g1 --no-apply-immediately
```

What happens if I add a new instance to an existing cluster?

All new instances that are created use the old server certificate and require TLS connections using the old CA certificate. Any new Amazon DocumentDB instances created after January 25, 2024 will default to using the new certificate rds-ca-rsa2048-g1.

What happens if there is an instance replacement or failover on my cluster?

If there is an instance replacement in your cluster, the new instance that is created continues to use the same server certificate that the instance was previously using. We recommend that you update server certificates for all instances at the same time. If a failover occurs in the cluster, the server certificate on the new primary is used.

If I'm not using TLS to connect to my cluster, do I still need to update each of my instances?

We highly recommend enabling TLS. In the event that you do not enable TLS, we still recommend rotating the certificates on your Amazon DocumentDB instances in the event you plan to use TLS to connect to your clusters in the future. If you never plan to use TLS to connect to your Amazon DocumentDB clusters, no action is needed.

If I'm not using TLS to connect to my cluster but I plan to in the future, what should I do?

If you created a cluster before January, 2024, follow Step 1 and Step 2 in the previous section to ensure that your application is using the updated CA bundle, and that each Amazon DocumentDB instance is using the latest server certificate. If you create a cluster after January 25, 2024, your cluster will already have the latest server certificate (rds-ca-rsa2048-g1). To verify that your application is using the latest CA bundle, see If I'm not using TLS to connect to my cluster, do I still need to update each of my instances?

Can the deadline be extended beyond August, 2024?

If your applications are connecting via TLS, the deadline cannot be extended.

How can I be sure that I'm using the newest CA bundle?

To verify that you have the newest bundle, use the following command. To run this command, you must have java installed and the java tools need to be in the PATH variable of your shell. For more information, see Using Java

macOS and Amazon Linux

```
keytool -printcert -v -file global-bundle.pem
```

Windows

```
keytool -printcert -v -file global-bundle.p7b
```

Why do I see "RDS" in the name of the CA bundle?

For certain management features, such as certificate management, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS).

When will the new certificate expire?

The new server certificate will expire (generally) as follows:

- **rds-ca-rsa2048-g1**—Expires 2061

- **rds-ca-rsa4096-g1**—Expires 2121
- **rds-ca-ecc384-g1**—Expires 2121

What kind of errors will I see if I don't take action before the certificate expires?

Error messages will vary depending on your driver. In general, you'll see certificate validation errors that contain the string "certificate has expired".

If I applied the new server certificate, can I revert it back to the old server certificate?

If you need to revert an instance to the old server certificate, do so for all instances in the cluster. You can revert the server certificate for each instance in a cluster by using the AWS Management Console or the AWS CLI.

Using the AWS Management Console

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the list of Regions in the upper-right corner of the screen, choose the AWS Region in which your clusters reside.
3. In the navigation pane on the left side of the console, choose **Clusters**.
4. In the **Clusters** table, under **Cluster identifier**, select an instance to modify. Choose **Actions**, and then choose **Modify**.
5. Under **Certificate authority**, you can select the old server certificate (`rds-ca-2019`).
6. Choose **Continue** to view a summary of your modifications.
7. In this resulting page, you can choose to schedule your modifications to be applied in the next maintenance window or apply your modifications immediately. Make your selection, and choose **Modify instance**.

> **Note**
>
> If you choose to apply your modifications immediately, any changes in the pending modifications queue are also applied. If any of the pending modifications require downtime, choosing this option can cause unexpected downtime.

Using the AWS CLI

```
aws docdb modify-db-instance --db-instance-identifier <db_instance_name> ca-
certificate-identifier rds-ca-2019 <--apply-immediately | --no-apply-immediately>
```

If you choose `--no-apply-immediately`, the changes will be applied during the cluster's next maintenance window.

If I restore from a snapshot or a point in time restore, will it have the new server certificate?

If you restore a snapshot or perform a point-in-time restore after August, 2024, the new cluster that is created will use the new CA certificate.

What if I'm having issues connecting directly to my Amazon DocumentDB cluster from any Mac OS?

Mac OS has updated the requirements for trusted certificates. Trusted certificates must now be valid for 397 days or fewer (see https://support.apple.com/en-us/HT211025).

> **Note**
>
> This restriction is observed in newer versions of Mac OS.

Amazon DocumentDB instance certificates are valid for over four years, longer than the Mac OS maximum. In order to connect directly to an Amazon DocumentDB cluster from a computer running Mac OS, you must allow invalid certificates when creating the TLS connection. In this case, invalid certificates mean that the validity period is longer than 397 days. You should understand the risks before allowing invalid certificates when connecting to your Amazon DocumentDB cluster.

To connect to an Amazon DocumentDB cluster from Mac OS using the AWS CLI, use the `tlsAllowInvalidCertificates` parameter.

```
mongo --tls --host <hostname> --username <username> --password <password> --port 27017
 --tlsAllowInvalidCertificates
```

Compliance validation in Amazon DocumentDB

The security and compliance of Amazon DocumentDB is assessed by third-party auditors as part of multiple AWS compliance programs, including the following:

- System and Organization Controls (SOC) 1, 2, and 3. For more information, see SOC.
- Federal Risk and Authorization Management Program (FedRAMP). For more information, see AWS Services in Scope by Compliance Program.
- Payment Card Industry Data Security Standard (PCI DSS). For more information, see PCI DSS.
- ISO 9001, 27001, 27017, and 27018. For more information, see ISO Certified.
- Health Insurance Portability and Accountability Act Business Associate Agreement (HIPAA BAA). For more information, see HIPAA Compliance

AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using Amazon DocumentDB is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of Amazon DocumentDB is subject to compliance with standards like HIPAA or PCI, AWS provides resources to help:

- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
- Architecting for HIPAA Security and Compliance Whitepaper – A whitepaper that describes how companies can use AWS to create HIPAA-compliant applications.

Resilience in Amazon DocumentDB

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

An Amazon DocumentDB cluster can only be created in an Amazon VPC that has at least two subnets in at least two Availability Zones. By distributing your cluster instances across at least two Availability Zones, Amazon DocumentDB helps ensure that there are instances available in your cluster in the unlikely event of an Availability Zone failure. The cluster volume for your Amazon DocumentDB cluster always spans three Availability Zones to provide durable storage with less possibility of data loss.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, Amazon DocumentDB offers several features to help support your data resiliency and backup needs.

Fault-tolerant and self-healing storage

Each 10 GB portion of your storage volume is replicated six ways, across three Availability Zones. Amazon DocumentDB uses fault-tolerant storage that transparently handles the loss of up to two copies of data without affecting database write availability, and up to three copies without affecting read availability. Amazon DocumentDB storage is also self-healing; data blocks and disks are continuously scanned for errors and replaced automatically.

Manual backups and restore

Amazon DocumentDB provides the capability to create full backups of your cluster for long-term retention and recovery. For more information, see Backing up and restoring in Amazon DocumentDB.

Point-in-time recovery

Point-in-time recovery helps protect your Amazon DocumentDB clusters from accidental write or delete operations. With point-in-time recovery, you don't have to worry about creating, maintaining, or scheduling on-demand backups. For more information, see Restoring to a point in time.

Infrastructure security in Amazon DocumentDB

As a managed service, Amazon DocumentDB is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Amazon DocumentDB through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

You can call these API operations from any network location. You can use Amazon DocumentDB policies to control access from specific Amazon Virtual Private Cloud (Amazon VPC) endpoints or specific VPCs. Effectively, this isolates network access to a given Amazon DocumentDB resource from only the specific VPC within the AWS network.

> **Note**
>
> Amazon DocumentDB does not support resource-based access policies.

Amazon DocumentDB API and interface VPC endpoints (AWS PrivateLink)

> **Note**
>
> Amazon DocumentDB elastic clusters do not support AWS PrivateLink VPC endpoints.

You can establish a private connection between your VPC and Amazon DocumentDB API endpoints by creating an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink.

While Amazon DocumentDB instance-based clusters do not require an interface VPC endpoint connection, AWS PrivateLink enables you to privately access Amazon DocumentDB API operations without an internet gateway, NAT device, VPN connection, or Direct Connect connection. Amazon DocumentDB instances in your VPC don't need public IP addresses to communicate with Amazon DocumentDB API endpoints to launch, modify, or terminate database instances and database clusters. Your Amazon DocumentDB instances also don't need public IP addresses to use any of the available Amazon DocumentDB API operations. Traffic between your VPC and Amazon DocumentDB doesn't leave the Amazon network.

Each interface endpoint is represented by one or more elastic network interfaces in your subnets. For more information, see Elastic network interfaces in the *Amazon EC2 User Guide.*

For more information about VPC endpoints, see Access an AWS service using an interface VPC endpoint in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*. For more information about Amazon DocumentDB operations, see the Amazon DocumentDB API Reference.

Topics

- Considerations for VPC endpoints
- Region availability
- Creating an interface VPC endpoint for Amazon DocumentDB API
- Creating a VPC endpoint policy for the Amazon DocumentDB API

Considerations for VPC endpoints

Before you set up an interface VPC endpoint for Amazon DocumentDB API endpoints, ensure that you review the interface endpoint prerequisites in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*.

All Amazon DocumentDB API operations relevant to managing Amazon DocumentDB resources are available from your VPC using AWS PrivateLink.

VPC endpoint policies are supported for Amazon DocumentDB API endpoints. By default, full access to Amazon DocumentDB API operations is allowed through the endpoint. For more information, see Control access to VPC endpoints using endpoint policies in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*.

Region availability

Amazon DocumentDB API currently supports VPC endpoints in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)
- Africa (Cape Town)
- Asia Pacific (Hong Kong)
- Asia Pacific (Mumbai)
- Asia Pacific (Hyderabad)
- Asia Pacific (Osaka)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- Canada (Central)
- China (Beijing)
- China (Ningxia)
- Europe (Frankfurt)
- Europe (Ireland)
- Europe (London)
- Europe (Paris)
- Europe (Spain)
- Europe (Milan)
- Middle East (UAE)
- South America (São Paulo)
- AWS GovCloud (US-East)
- AWS GovCloud (US-West)

Creating an interface VPC endpoint for Amazon DocumentDB API

You can create a VPC endpoint for the Amazon DocumentDB API using either the Amazon VPC Console or the AWS Command Line Interface (AWS CLI). For more information, see Access an AWS service using an interface VPC endpoint in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*.

Create a VPC endpoint for the Amazon DocumentDB API using the service name `com.amazonaws.region.rds`.

Excluding AWS Regions in China, if you enable private DNS for the endpoint, you can make API requests to Amazon DocumentDB with the VPC endpoint using its default DNS name for the AWS Regions, for example rds.us-east-1.amazonaws.com. For the China (Beijing) and China (Ningxia) AWS Regions, you can make API requests with the VPC endpoint using rds-api.cn-north-1.amazonaws.com.cn and rds-api.cn-northwest-1.amazonaws.com.cn, respectively.

For more information, see Access an AWS service using an interface VPC endpoint in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*.

Creating a VPC endpoint policy for the Amazon DocumentDB API

You can attach an endpoint policy to your VPC endpoint that controls access to the Amazon DocumentDB API. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see Control access to VPC endpoints using endpoint policies in the *Amazon Virtual Private Cloud (AWS PrivateLink) User Guide*.

Example: VPC endpoint policy for Amazon DocumentDB API actions

The following is an example of an endpoint policy for the Amazon DocumentDB API. When attached to an endpoint, this policy grants access to the listed Amazon DocumentDB API actions for all principals on all resources.

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Principal": "*",
      "Effect": "Allow",
      "Action": [
        "rds:CreateDBInstance",
        "rds:ModifyDBInstance",
        "rds:CreateDBSnapshot"
      ],
      "Resource": "*"
    }
  ]
}
```

Example: VPC endpoint policy that denies all access from a specified AWS account

The following VPC endpoint policy denies AWS account 123456789012 all access to resources using the endpoint. The policy allows all actions from other AWS accounts.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Resource": "*",
      "Principal": "*"
    },
    {
      "Action": "*",
      "Effect": "Deny",
      "Resource": "*",
      "Principal": { "AWS": [ "123456789012" ] }
    }
  ]
}
```

Security best practices for Amazon DocumentDB

Amazon DocumentDB provides security features including Amazon VPC isolation, encryption at rest using AWS KMS, encryption in transit using TLS, and fine-grained access control through IAM. Use the following best practices to secure your Amazon DocumentDB clusters, instances, and data.

Use AWS Identity and Access Management (IAM) roles to control access to Amazon DocumentDB API operations. This includes operations that create, modify, or delete Amazon DocumentDB resources such as clusters, security groups, and parameter groups. When applications or other AWS services need to access Amazon DocumentDB resources, use an IAM role to provide temporary credentials rather than storing long-term credentials in your application. When creating IAM roles, apply the principle of least privilege. For more information, see IAM roles and Common scenarios for roles in the *IAM User Guide*.

- Enforce least privilege with role-based access control.
- Don't use the AWS account root user to manage Amazon DocumentDB resources. Instead, use AWS Identity and Access Management Identity Center to create human users with temporary credentials. For more information, see IAM Identity Center users in the *IAM User Guide*.
- Grant each identity the minimum set of permissions required to perform their duties.
- Use IAM policies to manage permissions and follow the principle of least privilege. For more information, see Security best practices in IAM in the *IAM User Guide*.
- If you use IAM users with long-term credentials, rotate those credentials regularly.

> **Warning**
>
> This scenario requires IAM users with programmatic access and long-term credentials, which presents a security risk. To help mitigate this risk, we recommend that you provide these users with only the permissions they require to perform the task and that you remove these users when they are no longer needed. Access keys can be updated if necessary. For more information, see Update access keys in the *IAM User Guide*.

- Configure Secrets Manager to automatically rotate the secrets for Amazon DocumentDB. For more information, see Rotating your secrets and Rotating secrets for Amazon DocumentDB in the *AWS Secrets Manager User Guide*.
- Use Transport Layer Security (TLS) and encryption at rest to encrypt your data.

Auditing Amazon DocumentDB events

With Amazon DocumentDB (with MongoDB compatibility), you can audit events that were performed in your cluster. Examples of logged events include successful and failed authentication attempts, dropping a collection in a database, or creating an index. By default, auditing is disabled on Amazon DocumentDB and requires that you opt in to use this feature.

When auditing is enabled, Amazon DocumentDB records Data Definition Language (DDL), Data Manipulation Language (DML), authentication, authorization, and user management events to Amazon CloudWatch Logs. When auditing is enabled, Amazon DocumentDB exports your cluster's auditing records (JSON documents) to Amazon CloudWatch Logs. You can use Amazon CloudWatch Logs to analyze, monitor, and archive your Amazon DocumentDB auditing events.

Although Amazon DocumentDB does not charge an additional cost to enable auditing, you are charged standard rates for the usage of CloudWatch Logs. For information about CloudWatch Logs pricing, see Amazon CloudWatch pricing.

The Amazon DocumentDB auditing feature is distinctly different from the service resource usage that is monitored with AWS CloudTrail. CloudTrail records operations that are performed with the AWS Command Line Interface (AWS CLI) or AWS Management Console on resources like clusters, instances, parameter groups, and snapshots. Auditing of resources with CloudTrail is on by default and cannot be disabled. The Amazon DocumentDB auditing feature is an opt-in feature. It records operations that take place within your cluster on objects, such as databases, collections, indexes, and users.

Topics

- Supported events
- Enabling auditing
- Disabling auditing
- Accessing your audit events
- Filtering DML audit events

Supported events

Amazon DocumentDB auditing supports the following event categories:

- **Data Definition Language (DDL)** - includes database management operations, connections, user management, and authorization.
- **Data Manipulation Language read events (DML reads)** - includes `find()` and the various aggregation operators, arithmetic operators, boolean operators, and other read query operators.
- **Data Manipulation Language write events (DML writes)** - includes `insert(), update(), delete(),` and `bulkWrite()` operators

The event types are as follows.

Event Type	Category	Description
`authCheck`	Authorization	Result code 0: Success
		Result code 13: Unauthorized attempts to perform an operation.
`authenticate`	Connection	Successful or failed authentication attempts on a new connection.
`auditConfigure`	DDL	Audit filter configuration.
`createDatabase`	DDL	Creation of a new database.
`createCollection`	DDL	Creation of a new collection within a database.
`createIndex`	DDL	Creation of a new index within a collection.
`dropCollection`	DDL	Dropping of a collection within a database.
`dropDatabase`	DDL	Dropping of a database.

Event Type	Category	Description
dropIndex	DDL	Dropping of an index within a collection.
modifyChangeStreams	DDL	Change stream was created.
renameCollection	DDL	Renaming of a collection within a database.
createRole	Role Management	Creating a role.
dropAllRolesFromDatabase	Role Management	Dropping all roles within a database.
dropRole	Role Management	Dropping a role.
grantPrivilegesToRole	Role Management	Granting privileges to a role.
grantRolesToRole	Role Management	Granting roles to a user-defined role.
revokePrivilegesFromRole	Role Management	Revoking privileges from a role.
revokeRolesFromRole	Role Management	Revoking roles from a user-defined role.
updateRole	Role Management	Updating a role.
createUser	User Management	Creation of a new user.
dropAllUsersFromDatabase	User Management	Dropping of all users within a database.

Event Type	Category	Description
`dropUser`	User Management	Dropping of an existing user.
`grantRolesToUser`	User Management	Granting roles to a user.
`revokeRolesFromUser`	User Management	Revoking roles from a user.
`updateUser`	UserManagement	Updating of an existing user.
`insert`	DML write	Inserts a document or documents into a collection.
`delete`	DML write	Deletes a document or documents from a collection.
`update`	DML write	Modifies an existing document or documents in a collection.
`bulkWrite`	DML write	Performs multiple write operations with controls for order of execution.
`setAuditConfig`	DML write	Set a new filter for DML auditing.

Event Type	Category	Description
`count`	DML read	Returns the count of documents that would match a find() query for the collection or view.
`countDocuments`	DML read	Returns the count of documents that match the query for a collection or view.
`find`	DML read	Selects documents in a collection or view and returns a cursor to the selected documents.
`getAuditConfig`	DML read	Retrieve the current filter for DML auditing.
`findAndModify`	DML read and DML write	Modifies and returns a single document.
`findOneAndDelete`	DML read and DML write	Deletes a single document based on the filter and sort criteria, returning the deleted document.
`findOneAndReplace`	DML read and DML write	Replaces a single document based on the specified filter.

Event Type	Category	Description
`findOneAndUpdate`	DML read and DML write	Updates a single document based on the filter and sort criteria.
`aggregate`	DML read and DML write	Supports APIs in the aggregation pipeline.
`distinct`	DML read	Finds the distinct values for a specified field across a single collection or view and returns the results in an array.

Note

Values in the DML event document parameter field have a 1KB size limit. Amazon DocumentDB truncates the value if it exceeds 1KB.

Note

TTL delete events are not audited at this time.

Enabling auditing

Enabling auditing on a cluster is a two-step process. Ensure that both steps are completed, or audit logs will not be sent to CloudWatch Logs.

Step 1. Enable the audit_logs cluster parameter

To enable auditing, you need to modify the `audit_logs` parameter in the parameter group. `audit_logs` is a comma-delimited list of events to log. Events must be specified in lowercase and there should be no white space between the list elements.

You can set the following values for the parameter group:

Value	Description	
`ddl`	Setting this will enable auditing for DDL events such as createDatabase, dropDatabase, createCollection, dropCollection, createIndex, dropIndex, authCheck, authenticate, createUser, dropUser, grantRolesToUser, revokeRolesFromUser, updateUser, and dropAllUsersFromDatabase	
`dml_read`	Setting this will enable auditing for DML read events such as find, sort count, distinct, group, projecta, unwind, geoNear, geoIntersects, geoWithin and other MongoDB read query operators.	

Value	Description
`dml_write`	Setting this will enable auditing for DML write events such as insert(), update(), delete(), and bulkWrite()
`all`	Setting this will enable auditing for your database events, such as read queries, write queries, database actions and administrator actions.
none	Setting this will disable auditing

Value	Description
`enabled` (legacy)	This is a legacy parameter setting that is equivalent to 'ddl'. Setting this will enable auditing for DDL events such as createDatabase, dropDatabase, createCollection, dropCollection, createIndex, dropIndex, authCheck, authenticate, createUser, dropUser, grantRolesToUser, revokeRolesFromUser, updateUser, and dropAllUsersFromDatabase. We do not recommend using this setting because it is a legacy setting.
`disabled` (legacy)	This is a legacy parameter setting that is equivalent to 'none'. We do not recommend using this setting because it is a legacy setting.

> **Note**
>
> The default value for the audit_logs cluster parameter is none (legacy "`disabled`").

You can also use the above mentioned values in combinations.

Value	Description	
`ddl, dml_read`	Setting this will enable auditing for DDL events and DML read events.	
`ddl, dml_write`	Setting this will enable auditing for DDL events and DML write	
`dml_read, dml_write`	Setting this will enable auditing for all DML events	

> **Note**
>
> You cannot modify a default parameter group.

For more information, see the following:

- Creating Amazon DocumentDB cluster parameter groups

  After creating a custom parameter group, modify it by changing the `audit_logs` parameter value to `all`.
- Modifying Amazon DocumentDB cluster parameter groups

Step 2. Enable Amazon CloudWatch Logs export

When the value of the `audit_logs` cluster parameter is `enabled`, `ddl`, `dml_read`, or `dml_write`, you must also enable Amazon DocumentDB to export logs to Amazon CloudWatch. If you omit either of these steps, audit logs will not be sent to CloudWatch.

When creating a cluster, performing a point-in-time-restore, or restoring a snapshot, you can enable CloudWatch Logs by following these steps.

Using the AWS Management Console

To enable Amazon DocumentDB exporting logs to CloudWatch using the console, see the following topics:

- **When creating a cluster** — In Creating a cluster and primary instance using the AWS Management Console, see **Create a Cluster: Additional Configurations** (step 5, **Log exports**)
- **When modifying an existing cluster** — Modifying an Amazon DocumentDB cluster
- **When performing a cluster snapshot restore** — Restoring from a cluster snapshot
- **When performing a point-in-time restore** — Restoring to a point in time

Using the AWS CLI

To enable audit logs when creating a new cluster

The following code creates the cluster `sample-cluster` and enables CloudWatch audit logs.

Example

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster \
    --db-cluster-identifier sample-cluster \
    --port 27017 \
    --engine docdb \
    --master-username master-username \
    --master-user-password password \
    --db-subnet-group-name default \
    --enable-cloudwatch-logs-exports audit
```

For Windows:

```
aws docdb create-db-cluster ^
    --db-cluster-identifier sample-cluster ^
    --port 27017 ^
    --engine docdb ^
    --master-username master-username ^
    --master-user-password password ^
    --db-subnet-group-name default ^
    --enable-cloudwatch-logs-exports audit
```

To enable audit logs when modifying an existing cluster

The following code modifies the cluster `sample-cluster` and enables CloudWatch audit logs.

Example

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
   --db-cluster-identifier sample-cluster \
   --cloudwatch-logs-export-configuration '{"EnableLogTypes":["audit"]}'
```

For Windows:

```
aws docdb modify-db-cluster ^
   --db-cluster-identifier sample-cluster ^
   --cloudwatch-logs-export-configuration '{"EnableLogTypes":["audit"]}'
```

Output from these operations looks something like the following (JSON format).

```
{
    "DBCluster": {
        "HostedZoneId": "ZNKXH85TT8WVW",
        "StorageEncrypted": false,
        "DBClusterParameterGroup": "default.docdb4.0",
        "MasterUsername": "<user-name>",
        "BackupRetentionPeriod": 1,
        "Port": 27017,
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-77186e0d"
            }
```

```
        ],
        "DBClusterArn": "arn:aws:rds:us-east-1:900083794985:cluster:sample-cluster",
        "Status": "creating",
        "Engine": "docdb",
        "EngineVersion": "4.0.0",
        "MultiAZ": false,
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1c",
            "us-east-1f"
        ],
        "DBSubnetGroup": "default",
        "DBClusterMembers": [],
        "ReaderEndpoint": "sample-cluster.cluster-ro-corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
        "EnabledCloudwatchLogsExports": [
            "audit"
        ],
        "PreferredMaintenanceWindow": "wed:03:08-wed:03:38",
        "AssociatedRoles": [],
        "ClusterCreateTime": "2019-02-13T16:35:04.756Z",
        "DbClusterResourceId": "cluster-YOS52CUXGDTNKDQ7DH72I4LED4",
        "Endpoint": "sample-cluster.cluster-corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
        "PreferredBackupWindow": "07:16-07:46",
        "DBClusterIdentifier": "sample-cluster"
    }
}
```

Disabling auditing

You can disable auditing by disabling CloudWatch Logs export and disabling the `audit_logs` parameter.

Disabling CloudWatch Logs export

You can disable exporting audit logs using either the AWS Management Console or the AWS CLI.

Using the AWS Management Console

The following procedure uses the AWS Management Console to disable Amazon DocumentDB exporting logs to CloudWatch.

To disable audit logs

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**. Then choose the button to the left of the name of the cluster for which you want to disable exporting logs.
3. Choose **Actions**, and then choose **Modify**.
4. Scroll down to the **Log exports** section and choose **Disabled**.
5. Choose **Continue**.
6. Review your changes, and then choose when you want this change applied to your cluster.
   - **Apply during the next scheduled maintenance window**
   - **Apply immediately**
7. Choose **Modify cluster**.

Using the AWS CLI

The following code modifies the cluster `sample-cluster` and disables CloudWatch audit logs.

Example

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
   --db-cluster-identifier sample-cluster \
   --cloudwatch-logs-export-configuration '{"DisableLogTypes":["audit"]}'
```

For Windows:

```
aws docdb modify-db-cluster ^
   --db-cluster-identifier sample-cluster ^
   --cloudwatch-logs-export-configuration '{"DisableLogTypes":["audit"]}'
```

Output from this operation looks something like the following (JSON format).

```
{
    "DBCluster": {
        "DBClusterParameterGroup": "default.docdb4.0",
        "HostedZoneId": "ZNKXH85TT8WVW",
```

```
        "MasterUsername": "<user-name>",
        "Status": "available",
        "Engine": "docdb",
        "Port": 27017,
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1c",
            "us-east-1f"
        ],
        "EarliestRestorableTime": "2019-02-13T16:35:50.387Z",
        "DBSubnetGroup": "default",
        "LatestRestorableTime": "2019-02-13T16:35:50.387Z",
        "DBClusterArn": "arn:aws:rds:us-east-1:900083794985:cluster:sample-
cluster2",
        "Endpoint": "sample-cluster2.cluster-corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
        "ReaderEndpoint": "sample-cluster2.cluster-ro-corcjozrlsfc.us-
east-1.docdb.amazonaws.com",
        "BackupRetentionPeriod": 1,
        "EngineVersion": "4.0.0",
        "MultiAZ": false,
        "ClusterCreateTime": "2019-02-13T16:35:04.756Z",
        "DBClusterIdentifier": "sample-cluster2",
        "AssociatedRoles": [],
        "PreferredBackupWindow": "07:16-07:46",
        "DbClusterResourceId": "cluster-YOS52CUXGDTNKDQ7DH72I4LED4",
        "StorageEncrypted": false,
        "PreferredMaintenanceWindow": "wed:03:08-wed:03:38",
        "DBClusterMembers": [],
        "VpcSecurityGroups": [
            {
                "Status": "active",
                "VpcSecurityGroupId": "sg-77186e0d"
            }
        ]
    }
}
```

Disabling the audit_logs parameter

To disable the `audit_logs` parameter for your cluster, you can modify the cluster so that it uses a parameter group where the `audit_logs` parameter value is `disabled`. Or you can modify the `audit_logs` parameter value in the cluster's parameter group so that it is `disabled`.

For more information, see the following topics:

- Modifying an Amazon DocumentDB cluster
- Modifying Amazon DocumentDB cluster parameter groups

Accessing your audit events

Use following steps to access your audit events on Amazon CloudWatch.

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Make sure that you are in the same Region as your Amazon DocumentDB cluster.
3. In the navigation pane, choose **Logs**.
4. To find the audit logs for your cluster, from the list locate and choose **`/aws/docdb/`*`yourClusterName`*`/audit`**.

   The auditing events for each of your instances are available under each of the respective instance names.

Filtering DML audit events

Getting started with DML audit filtering

DML audit events can be filtered before they are written to Amazon CloudWatch. To utilize this feature, audit log and DML logging must be enabled. Amazon DocumentDB supports filtering on `atype`, `command`, `user`, `namespace`, and `auditAuthorizationSuccess`.

> **Note**
>
> DDL events are not filtered.

You can enable audit filtering at anytime by specifying the audit filter using the `setAuditConfig`, `filter`, and `auditAuthorizationSuccess` parameters in the `db.adminCommand( { command } )` operation:

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter:
         {
            //filter conditions
         },
      auditAuthorizationSuccess: true | false
   }
)
```

You can also retrieve the audit filter settings by running the following command:

```
db.admin.runCommand( { getAuditConfig: 1})
```

Security requirements

Only database users/roles with privileged action `auditConfigure` can execute the above commands against `admindb` when setting or listing DML audit filters. You can either use one of the built-in roles from [`clusterAdmin`, `hostManager`, `root`] or create custom roles that have `auditConfigure` privileges. The following is an example of using existing roles with the `auditConfigure` privilege and an example with custom roles.

User with built-in role:

```
use admin
db.createUser(
  {
    user: "myClusterAdmin",
    pwd: "password123",
    roles: [ { role: "clusterAdmin", db: "admin" } ]
  }
)
```

User with custom roles:

```
use admin
```

```
db.createRole(
   {
     role: "myRole",
     privileges: [
       { resource: { cluster: true }, actions: [ "auditConfigure" ] }
     ],
     roles: []
   }
)
db.createUser(
  {
    user: "myUser",
    pwd: "myPassword",
    roles: [ { role: "myRole", db: "admin" } ]
  }
)
```

Filtering use cases

Example: filtering events by commands

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {
        "$and": [
         {
            "param.command":
               {
                  $in: [ "find","count", "insert", "delete", "update",
 "findandmodify" ]
               }
         }
         ]
      },
      auditAuthorizationSuccess: true
   }
)
```

Example: filtering events by user name

In this example, only user "myUser" will be logged:

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {
      "$and": [
         {
            "param.user":
               {
                  $in: [ "myUser" ]
               }
         }
         ]},
      auditAuthorizationSuccess: true})
```

Example: filtering by atype

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {atype: "authCheck"},
      auditAuthorizationSuccess: true
   })
```

> **Note**
>
> All DML logs have authCheck as atype. Only DDL has a different atype. If you put a value other than authCheck in the filter, it will not produce a DML log in CloudWatch.

Example: filtering by using multiple filters joined by operators

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {
      "$and": [
         {
            "param.command":
               {
                  $in: [ "find","count", "insert", "delete", "update",
 "findandmodify" ]
```

```
            }
        }
        ],
       "$nor": [
        {
           "param.command":
            {
               $in: ["count", "insert", "delete", "update", "findandmodify" ]
            }
        }]
       },
      auditAuthorizationSuccess: true})
```

> **Note**
>
> At the top level, only `$and`, `$or`, and `$nor` are supported. Any other operators are not supported and will cause an error.

Example: filtering by events by `auditAuthorizationSuccess`

In this filter, all commands that have successfully passed authorization will not be logged:

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {},
      auditAuthorizationSuccess: false
   }
)
```

Example: filtering with `$in` and `$nin` conditions

When using both in `$in` and `$nin`, the command will not be logged as there will be an implicit "and" between the conditions. In this example, regex will block the `find` command so nothing will be logged:

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {
      "$and": [
```

```
         {
            atype: "authCheck",
            "param.command":
               {
                  $in: [ "find", "insert", "delete", "update", "findandmodify" ],
                  $nin: ["count", "insert", "delete", "update", "findandmodify" ],
                  $not: /^^find.*/
               }
         },
         ],
       "$or": [
         {
            "param.command":
               {
                  $nin: ["count", "insert", "delete", "update", "findandmodify" ]
               }
         }]
       },
      auditAuthorizationSuccess: true})
```

Example: filtering by namespace

```
db.admin.runCommand(
   {
      setAuditConfig: 1,
      filter: {
      "$and": [
         {
            "param.ns":
               {
                  $in: [ "test.foo" ]
               }
         }
         ]},
      auditAuthorizationSuccess: true})
```

Example: resetting to default filter

Resetting to the default value means that every DML audit event will be logged. To reset filtering to the default value, run the following command:

```
db.admin.runCommand(
   {
```

```
      setAuditConfig: 1,
      filter: {},
      auditAuthorizationSuccess: true
   }
)
```

Amazon VPC and Amazon DocumentDB

Amazon Virtual Private Cloud (Amazon VPC) makes it possible for you to launch AWS resources, such as Amazon DocumentDB instances, into a virtual private cloud (VPC).

When you use a VPC, you have control over your virtual networking environment. You can choose your own IP address range, create subnets, and configure routing and access control lists. There is no additional cost to run your cluster in a VPC.

Accounts have a default VPC. All new clusters are created in the default VPC unless you specify otherwise.

Topics

- DocumentDB clusters in a VPC
- Accessing an Amazon DocumentDB cluster in a VPC
- Create an IPv4-only VPC for use with a DocumentDB cluster
- Create a dual-stack VPC for use with a DocumentDB cluster

Following, you can find a discussion about VPC functionality relevant to Amazon DocumentDB clusters. For more information about Amazon VPC, see the Amazon VPC User Guide.

DocumentDB clusters in a VPC

Your Amazon DocumentDB cluster is in a virtual private cloud (VPC). A VPC is a virtual network that is logically isolated from other virtual networks in the AWS Cloud. Amazon VPC makes it possible for you to launch AWS resources, such as an Amazon DocumentDB cluster or Amazon EC2 instance, into a VPC. The VPC can either be a default VPC that comes with your account or one that you create. All VPCs are associated with your AWS account.

Your default VPC has three subnets that you can use to isolate resources inside the VPC. The default VPC also has an internet gateway that can be used to provide access to resources inside the VPC from outside the VPC.

For scenarios involving Amazon DocumentDB clusters in a VPC and outside of a VPC, see Accessing an Amazon DocumentDB cluster in a VPC.

Topics

- Working with clusters in a VPC
- Working with subnet groups
- Amazon DocumentDB IP addressing
- Creating a cluster in a VPC

Working with clusters in a VPC

Here are some tips on working with a cluster in a VPC:

- Your VPC must have at least two subnets. These subnets must be in two different Availability Zones in the AWS Region where you want to deploy your cluster. A subnet is a segment of a VPC's IP address range that you can specify and that you can use to group clusters based on your security and operational needs.
- If you want your cluster in the VPC to be publicly accessible, make sure to turn on the VPC attributes `DNS hostnames` and `DNS resolution`.
- Your VPC must have a subnet group that you create. You create a subnet group by specifying the subnets you created. Amazon DocumentDB chooses a subnet and an IP address within that subnet group to associate with the primary instance in your cluster. The primary instance uses the Availability Zone that contains the subnet.
- Your VPC must have a VPC security group that allows access to the cluster.

  For more information, see Accessing an Amazon DocumentDB cluster in a VPC.
- The CIDR blocks in each of your subnets must be large enough to accommodate spare IP addresses for Amazon DocumentDB to use during maintenance activities, including failover and compute scaling. For example, a range such as 10.0.0.0/24 and 10.0.1.0/24 is typically large enough.
- A VPC can have an `instance tenancy` attribute of either `default` or `dedicated`. All default VPCs have the `instance tenancy` attribute set to default, and a default VPC can support any instance class.

  If you choose to have your cluster in a dedicated VPC where the `instance tenancy` attribute is set to `dedicated`, the instance class of your cluster must be one of the approved Amazon EC2

dedicated instance types. For example, the `r5.large` EC2 dedicated instance corresponds to the `db.r5.large` instance class. For information about instance tenancy in a VPC, see Amazon EC2 instances in the *Amazon Elastic Compute Cloud User Guide*.

For more information about the instance types that can be in a dedicated instance, see Amazon EC2 dedicated instances on the Amazon EC2 pricing page.

> **Note**
>
> When you set the `instance tenancy` attribute to `dedicated` for a cluster, it doesn't guarantee that the cluster will run on a dedicated host.

Working with subnet groups

Subnets are segments of a VPC's IP address range that you designate to group your resources based on security and operational needs. A subnet group is a collection of subnets (typically private) that you create in a VPC and that you then designate for your clusters. By using a subnet group, you can specify a particular VPC when creating clusters using the AWS CLI or Amazon DocumentDB API. If you use the console, you can choose the VPC and subnet groups you want to use.

Each subnet group should have subnets in at least two Availability Zones in a given AWS Region. When creating a cluster in a VPC, you choose a subnet group for it. From the subnet group, Amazon DocumentDB chooses a subnet and an IP address within that subnet to associate with the primary instance in your cluster. The database uses the Availability Zone that contains the subnet. DocumentDB always assigns an IP address from a subnet that has free IP address space.

The subnets in a subnet group are either public or private, depending on the configuration that you set for their network access control lists (network ACLs) and routing tables. For a cluster to be publicly accessible, all of the subnets in its subnet group must be public. If a subnet that's associated with a publicly accessible cluster changes from public to private, it can affect cluster availability.

To create a subnet group that supports dual-stack mode, make sure that each subnet that you add to the subnet group has an Internet Protocol version 6 (IPv6) CIDR block associated with it. For more information, see Amazon DocumentDB IP addressing and IPv6 support for your VPC in the *Amazon Virtual Private Cloud User Guide*.

When Amazon DocumentDB creates a cluster in a VPC, it assigns a network interface to the cluster by using an IP address from your subnet group. Always use the Domain Name System (DNS) name to connect to your cluster because the underlying IP address changes during failover.

> **Note**
>
> For each cluster that you run in a VPC, make sure to reserve at least one address in each subnet in the subnet group for use by Amazon DocumentDB for recovery actions.

Shared subnets

You can create a cluster in a shared VPC.

Some considerations to keep in mind while using shared VPCs:

- You can move a cluster from a shared VPC subnet to a non-shared VPC subnet and vice-versa.
- Participants in a shared VPC must create a security group in the VPC to allow them to create a cluster.
- Owners and participants in a shared VPC can access the database by using DocumentDB queries. However, only the creator of a resource can make any API calls on the resource.

Amazon DocumentDB IP addressing

IP addresses enable resources in your VPC to communicate with each other, and with resources over the internet. Amazon DocumentDB supports both IPv4 and IPv6 addressing protocols. By default, Amazon DocumentDB and Amazon VPC use the IPv4 addressing protocol. You can't turn off this behavior. When you create a VPC, make sure to specify an IPv4 CIDR block (a range of private IPv4 addresses). You can optionally assign an IPv6 CIDR block to your VPC and subnets, and assign IPv6 addresses from that block to clusters in your subnet.

> **Note**
>
> Dual-stack mode (IPv6 addressing) is only supported in Amazon DocumentDB versions 4.0 and 5.0.

Support for the IPv6 protocol expands the number of supported IP addresses. By using the IPv6 protocol, you ensure that you have sufficient available addresses for the future growth of the

internet. New and existing DocumentDB resources can use IPv4 and IPv6 addresses within your VPC. Configuring, securing, and translating network traffic between the two protocols used in different parts of an application can cause operational overhead. You can standardize on the IPv6 protocol for Amazon DocumentDB resources to simplify your network configuration.

Topics

- IPv4 addresses
- IPv6 addresses
- Dual-stack mode

IPv4 addresses

When you create a VPC, you must specify a range of IPv4 addresses for the VPC in the form of a CIDR block, such as `10.0.0.0/16`. A subnet group defines the range of IP addresses in this CIDR block that a cluster can use. These IP addresses can be private or public.

A private IPv4 address is an IP address that's not reachable over the internet. You can use private IPv4 addresses for communication between your cluster and other resources, such as Amazon EC2 instances, in the same VPC. Each cluster has a private IP address for communication in the VPC.

A public IP address is an IPv4 address that's reachable from the internet. Public IP addressing is not allowed for your DocumentDB cluster. Any public IP address should be resolved by the Internet gateway and by EC2 in the public subnet.

To see how to create a VPC with only private IPv4 addresses that you can use for a common Amazon DocumentDB scenario, see Create an IPv4-only VPC for use with a DocumentDB cluster.

IPv6 addresses

You can optionally associate an IPv6 CIDR block with your VPC and subnets, and assign IPv6 addresses from that block to the resources in your VPC. Each IPv6 address is globally unique.

The IPv6 CIDR block for your VPC is automatically assigned from Amazon's pool of IPv6 addresses. You can't choose the range yourself.

When connecting to an IPv6 address, make sure that the following conditions are met:

- The client is configured so that client to database traffic over IPv6 is allowed.
- DocumentDB security groups used by the cluster are configured correctly so that client to database traffic over IPv6 is allowed.

- The client operating system stack allows traffic on the IPv6 address, and operating system drivers and libraries are configured to choose the correct default cluster endpoint (either IPv4 or IPv6).

For more information about IPv6, see IP Addressing in the *Amazon Virtual Private Cloud User Guide*.

Dual-stack mode

When a cluster can communicate over both the IPv4 and IPv6 addressing protocols, it's running in dual-stack mode. So, resources can communicate with the cluster over IPv4, IPv6, or both. DocumentDB disables Internet Gateway access for IPv6 endpoints of private dual-stack mode clusters. DocumentDB does this to ensure that your IPv6 endpoints are private and can only be accessed from within your VPC.

Topics

- Dual-stack mode and subnet groups
- Working with dual-stack mode clusters
- Modifying IPv4-only clusters to use dual-stack mode
- Dual-stack mode Region and version availability
- Limitations for dual-stack network clusters

Dual-stack mode and subnet groups

To use dual-stack mode, make sure that each subnet in the subnet group that you associate with the cluster has an IPv6 CIDR block associated with it. You can create a new subnet group or modify an existing subnet group to meet this requirement. After a cluster is in dual-stack mode, clients can connect to it normally. Make sure that client security firewalls and DocumentDB cluster security groups are accurately configured to allow traffic over IPv6. To connect, clients use the cluster's endpoint. Client applications can specify which protocol is preferred when connecting to a database. In dual-stack mode, the cluster detects the client's preferred network protocol, either IPv4 or IPv6, and uses that protocol for the connection.

If a subnet group stops supporting dual-stack mode because of subnet deletion or CIDR disassociation, there's a risk of an incompatible network state for clusters that are associated with the subnet group. Also, you can't use the subnet group when you create a new dual-stack mode cluster.

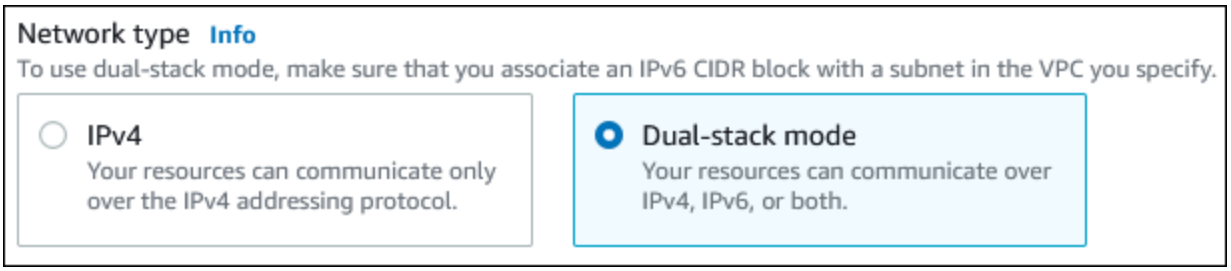
To determine whether a subnet group supports dual-stack mode by using the AWS Management Console, view the **Network type** on the details page of the subnet group. To determine whether a subnet group supports dual-stack mode by using the AWS CLI, run the `describe-db-subnet-groups` command and view `SupportedNetworkTypes` in the output.

Read replicas are treated as independent clusters and can have a network type that's different from the primary cluster. If you change the network type of a read replica's primary cluster, the read replica isn't affected. When you are restoring a cluster, you can restore it to any network type that's supported.

Working with dual-stack mode clusters

When you create or modify a cluster, you can specify dual-stack mode to allow your resources to communicate with your cluster over IPv4, IPv6, or both.

When you use the AWS Management Console to create or modify a cluster, you can specify dual-stack mode in the **Network type** section. The following image shows the **Network type** section in the console:



When you use the AWS CLI to create or modify a cluster, set the `--network-type` option to `DUAL` to use dual-stack mode. When you use the DocumentDB API to create or modify a cluster, set the `NetworkType` parameter to `DUAL` to use dual-stack mode. If dual-stack mode isn't supported by the specified DocumentDB engine version or subnet group, the `NetworkTypeNotSupported` error is returned.

For more information about creating a cluster, see Creating an Amazon DocumentDB cluster. For more information about modifying a cluster, see Modifying an Amazon DocumentDB cluster.

To determine whether a cluster is in dual-stack mode by using the console, view the **Network type** on the **Connectivity & security** tab for the cluster.

Modifying IPv4-only clusters to use dual-stack mode

You can modify an IPv4-only cluster to use dual-stack mode. To do so, change the network type of the cluster.

It is recommended that you change the network type of your Amazon DocumentDB cluster during a maintenance window. You can set network type manually by using the `modify-db-cluster` command.

Before modifying a cluster to use dual-stack mode, make sure that its subnet group supports dual-stack mode. If the subnet group associated with the cluster doesn't support dual-stack mode, specify a different subnet group that supports it when you modify the cluster. Modifying the subnet group of a cluster can cause downtime.

If you modify the subnet group of a cluster before you change the cluster to use dual-stack mode, make sure that the subnet group is valid for the cluster before and after the change.

Run the `ModifyDBCluster` call with only the `NetworkType` parameter set to `DUAL` to change the network to dual-stack mode. Adding other parameters along with `NetworkType` in the same API call could result in downtime. To modify multiple parameters, ensure that the network type modification is successfully completed before sending another `ModifyDBCluster` request with other parameters.

If you can't connect to the cluster after the change, make sure that the client and database security firewalls and route tables are accurately configured to allow traffic to the database on the selected network (either IPv4 or IPv6). You might also need to modify operating system parameter, libraries, or drivers to connect using an IPv6 address.

To modify an IPv4-only cluster to use dual-stack mode

1. Modify a subnet group to support dual-stack mode, or create a subnet group that supports dual-stack mode:

    a. Associate an IPv6 CIDR block with your VPC.

    For instructions, see Add or remove a CIDR block to your VPC in the *Amazon VPC User Guide*.

    b. Attach the IPv6 CIDR block to all of the subnets in your subnet group.

    For instructions, see Add or remove an IPv6 CIDR block to your subnet in the *Amazon Virtual Private Cloud User Guide*.

    c. Confirm that the subnet group supports dual-stack mode.

    If you are using the AWS Management Console, select the subnet group, and make sure that the **Supported network types** value is **Dual**.

If you are using the AWS CLI, run the `describe-db-subnet-groups` command, and make sure that the `SupportedNetworkType` value for the cluster is `Dual`.

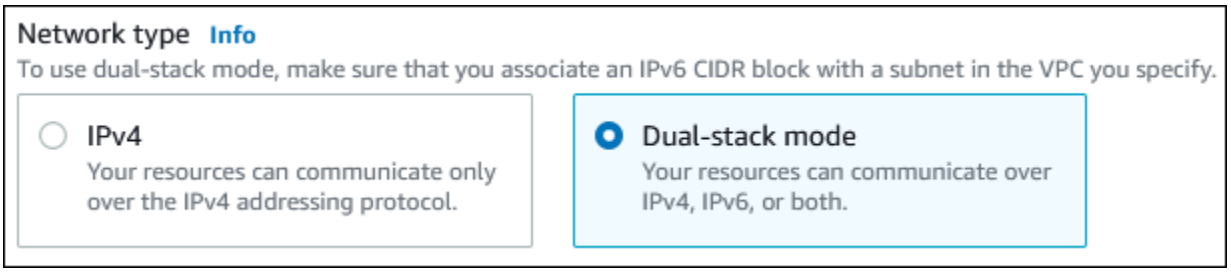
2. Modify the security group associated with the cluster to allow IPv6 connections to the database, or create a new security group that allows IPv6 connections.

   For instructions, see Security group rules in the *Amazon Virtual Private Cloud User Guide*.

3. Modify the cluster to support dual-stack mode. To do so, set the **Network type** to **Dual-stack mode**.

   If you are using the console, make sure that the following settings are correct:

   - **Network type** — **Dual-stack mode**



   - **Subnet group** — The subnet group that you configured in a previous step
   - **Security group** — The security that you configured in a previous step

   If you are using the AWS CLI, make sure that the following settings are correct:

   - `--network-type` — `dual`
   - `--db-subnet-group-name` — The subnet group that you configured in a previous step
   - `--vpc-security-group-ids` — The VPC security group that you configured in a previous step

   For example:

```
aws docdb modify-db-cluster --db-cluster-identifier <cluster-name> --network-type
 "DUAL"
```

4. Confirm that the cluster supports dual-stack mode.

If you are using the console, choose the **Connectivity & security** tab for the cluster and make sure that the **Network type** value is **Dual-stack mode**.

If you are using the AWS CLI, run the `describe-db-cluster` command, and make sure that the `NetworkType` value for the cluster is `dual`.

Run the `dig` command on the cluster endpoint to identify the IPv6 address associated with it:

```
dig <db-cluster-endpoint> AAAA
```

Use the cluster endpoint, not the IPv6 address, to connect to the cluster.

Dual-stack mode Region and version availability

Feature availability and support varies across AWS Regions.

Region support

Dual stack mode for Amazon DocumentDB is currently available in all commercial AWS Regions.

Version support

Dual-stack mode is supported on Amazon DocumentDB version 4.0, 5.0 and 8.0. If you are unable to access dual-stack mode on any of these versions, make sure that you are running the latest engine patch version on your cluster.

Limitations for dual-stack network clusters

The following limitations apply to dual-stack network clusters:

- Clusters can't use the IPv6 protocol exclusively. They can use IPv4 exclusively, or they can use the IPv4 and IPv6 protocol (dual-stack mode).
- Amazon DocumentDB doesn't support native IPv6 subnets.
- Clusters that use dual-stack mode must be private. They can't be publicly accessible.

Creating a cluster in a VPC

The following procedures help you create a cluster in a VPC. To use the default VPC, you can begin with step 2, and use the VPC and subnet group that have already been created for you. You can also create additional VPCs, if needed.

> **Note**
>
> If you want your cluster in the VPC to be publicly accessible, you must update the DNS information for the VPC by enabling the VPC attributes `DNS hostnames` and `DNS resolution`. For information about updating the DNS for a VPC instance, see View and update DNS attributes for your VPC in the *Amazon Virtual Private Cloud User Guide*.

Follow these steps to create a cluster in a VPC:

- Step 1: Create a VPC
- Step 2: Create a subnet group
- Step 3: Create a VPC security group
- Step 4: Create a cluster in the VPC

Step 1: Create a VPC

Create a VPC with subnets in at least two Availability Zones. You use these subnets when you create a subnet group. If you have a default VPC, a subnet is automatically created for you in each Availability Zone in the AWS Region.

For more information, see Create an IPv4-only VPC for use with a DocumentDB cluster, or see Create a VPC in the *Amazon Virtual Private Cloud User Guide*.

Step 2: Create a subnet group

A subnet group is a collection of subnets (typically private) that you create for a VPC and that you then designate for your clusters. A subnet group allows you to specify a particular VPC when you create clusters using the AWS CLI or DocumentDB API. If you use the AWS Management Console, you can just choose the VPC and subnets you want to use. Each subnet group must have at least one subnet in at least two Availability Zones in the AWS Region. As a best practice, each subnet group should have at least one subnet for every Availability Zone in the AWS Region.

For a cluster to be publicly accessible, the subnets in the subnet group must have an internet gateway. For more information about internet gateways for subnets, see Enable internet access for a VPC using an internet gateway in the *Amazon Virtual Private Cloud User Guide*.

> **Note**
>
> The subnet group for a local zone can have only one subnet.

When you create a cluster in a VPC, you can choose a subnet group. Amazon DocumentDB chooses a subnet and an IP address (within that subnet) to associate with your cluster. If no subnet groups exist, Amazon DocumentDB creates a default subnet group when you create the cluster. DocumentDB creates and associates an Elastic Network Interface to your cluster with that IP address. The cluster uses the Availability Zone that contains the subnet.

In this step, you create a subnet group and add the subnets that you created for your VPC.

To create a subnet group

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Subnet groups**.
3. Choose **Create**.
4. For **Name**, type the name of your subnet group.
5. For **Description**, type a description for your subnet group.
6. In the **Add subnets** section, for **VPC**, choose the default VPC or the VPC that you created. Then choose the Availability Zones that include the subnets from **Availability Zones**, and then choose the subnets from **Subnets**.
7. Choose Create.

   Your new subnet group appears in the **Subnet groups** list on the DocumentDB console. You can choose the subnet group to see details, including all of the subnets associated with the group, in the details pane at the bottom of the window.

Step 3: Create a VPC security group

Before you create your cluster, create a VPC security group to associate with it. If you don't create a VPC security group, you can use the default security group when you create the cluster. For

instructions on how to create a security group for your cluster, see Create an IPv4-only VPC for use with a DocumentDB cluster, or see Control traffic to your AWS resources using security groups in the *Amazon Virtual Private Cloud User Guide*.

Step 4: Create a cluster in the VPC

In this step, you create a cluster using the VPC name, the subnet group, and the VPC security group you created in the previous steps.

> **Note**
>
> If you want your cluster in the VPC to be publicly accessible, you must enable the VPC attributes `DNS hostnames` and `DNS resolution`. For more information, see View and update DNS attributes for your VPC in the *Amazon Virtual Private Cloud User Guide*.

For details on how to create a cluster, see Creating an Amazon DocumentDB cluster.

When prompted in the **Connectivity** section, enter the VPC name, the subnet group, and the VPC security group.

> **Note**
>
> Updating VPCs isn't currently supported for DocumentDBDocumentDB clusters.

Accessing an Amazon DocumentDB cluster in a VPC

Amazon DocumentDB supports the following scenarios for accessing a cluster in a VPC:

Topics

- A cluster in a VPC accessed by an Amazon EC2 instance in the same VPC
- A cluster in a VPC accessed by an Amazon EC2 instance in a different VPC

A cluster in a VPC accessed by an Amazon EC2 instance in the same VPC

A common use of a cluster in a VPC is to share data with an application server that is running in an Amazon EC2 instance in the same VPC.

The simplest way to manage access between EC2 instances and clusters in the same VPC is to do the following:

- Create a VPC security group for your clusters to be in. This security group can be used to restrict access to the clusters. For example, you can create a custom rule for this security group. This might allow TCP access using the port that you assigned to the cluster when you created it and an IP address you use to access the cluster for development or other purposes.
- Create a VPC security group for your EC2 instances (web servers and clients) to be in. This security group can, if needed, allow access to the EC2 instance from the internet by using the VPC's routing table. For example, you can set rules on this security group to allow TCP access to the EC2 instance over port 22.
- Create custom rules in the security group for your clusters that allow connections from the security group you created for your EC2 instances. These rules might allow any member of the security group to access the clusters.

There is an additional public and private subnet in a separate Availability Zone. An DocumentDB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easy to switch to a multi-AZ cluster deployment in the future.

For instructions on how to create a VPC with both public and private subnets for this scenario, see Create an IPv4-only VPC for use with a DocumentDB cluster.

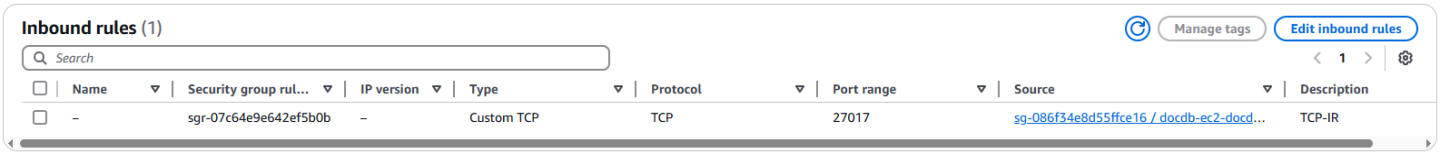
> **Tip**
>
> You can set up network connectivity between an Amazon EC2 instance and a DocumentDB cluster automatically when you create the cluster. For more information, see Connect Amazon EC2 automatically.

To create a rule in a VPC security group that allows connections from another security group, do the following:

1. Sign in to the AWS Management Console and open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. In the navigation pane, locate and choose **Security groups**.

3. Choose or create a security group for which you want to allow access to members of another security group. This is the security group that you use for your clusters. Choose the **Inbound rules** tab, and then choose **Edit inbound rules**.
4. On the **Edit inbound rules** page, choose **Add rule**.
5. For **Type**, choose the entry that corresponds to the port you used when you created your cluster, such as **Custom TCP**.
6. In the **Source** field, start typing the ID of the security group, which lists the matching security groups. Choose the security group with members that you want to have access to the resources protected by this security group. In the scenario preceding, this is the security group that you use for your EC2 instance.
7. If required, repeat the steps for the TCP protocol by creating a rule with **All TCP** as the **Type** and your security group in the **Source** field. If you intend to use the UDP protocol, create a rule with **All UDP** as the **Type** and your security group in **Source**.
8. Choose **Save rules**.

The following screen shows an inbound rule with a security group for its source.



For more information about connecting to a cluster from your EC2 instance, see Connect Amazon EC2 automatically.

A cluster in a VPC accessed by an Amazon EC2 instance in a different VPC

When your clusters is in a different VPC from the EC2 instance you are using to access it, you can use VPC peering to access the cluster.

A VPC peering connection is a networking connection between two VPCs that enables you to route traffic between them using private IP addresses. Resources in either VPC can communicate with each other as if they are within the same network. You can create a VPC peering connection between your own VPCs, with a VPC in another AWS account, or with a VPC in a different AWS Region. To learn more about VPC peering, see VPC peering in the *Amazon Virtual Private Cloud User Guide*.

Create an IPv4-only VPC for use with a DocumentDB cluster

A common scenario includes a cluster in a virtual private cloud (VPC) based on the Amazon VPC service. For example, this VPC could share data with a service or application that is running in the same VPC. In this topic, you create the VPC for this scenario.

Topics

- Step 1: Create a VPC with private and public subnets
- Step 2: Create a VPC security group for a public application
- Step 3: Create a VPC security group for a private cluster
- Step 4: Create a subnet group
- Deleting a VPC

Your cluster needs to be available only to your application, and not to the public internet. Thus, you create a VPC with both public and private subnets. The application is hosted in the public subnet, so that it can reach the public internet. The cluster is hosted in a private subnet. The application can connect to the cluster because it is hosted within the same VPC. But the cluster isn't available to the public internet, providing greater security.

The procedure in this topic configures an additional public and private subnet in a separate Availability Zone. These subnets aren't used by the procedure. A DocumentDB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easier to configure more than one DocumentDB instance.

This topic describes configuring a VPC for Amazon DocumentDB clusters. For more information about Amazon VPC, see the *Amazon VPC User Guide*.

> **Tip**
>
> You can set up network connectivity between an Amazon EC2 instance and a DocumentDB cluster automatically when you create the cluster. The network configuration is similar to the one described in this scenario. For more information, see Connect Amazon EC2 automatically.

Step 1: Create a VPC with private and public subnets

Use the following procedure to create a VPC with both public and private subnets.

To create a VPC and subnets

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. In the top-right corner of the AWS Management Console, choose the Region to create your VPC in. This example uses the US West (Oregon) Region.
3. In the upper-left corner, choose **VPC dashboard**. To begin creating a VPC, choose **Create VPC**.
4. For **Resources to create** under **VPC settings**, choose **VPC and more**.
5. For the **VPC settings**, set these values:
   - **Name tag auto-generation** — `example`
   - **IPv4 CIDR block** — `10.0.0.0/16`
   - **IPv6 CIDR block** — **No IPv6 CIDR block**
   - **Tenancy** — **Default**
   - **Number of Availability Zones (AZs)** — **2**
   - **Customize AZs** — Keep the default values
   - **Number of public subnets** — **2**
   - **Number of private subnets** — **2**
   - **Customize subnets CIDR blocks** — Keep the default values
   - **NAT gateways ($)** — **None**
   - **VPC endpoints** — **None**
   - **DNS options** — Keep the default values
6. Choose **Create VPC**.

Step 2: Create a VPC security group for a public application

Next, create a security group for public access. To connect to public EC2 instances in your VPC, you add inbound rules to your VPC security group. These allow traffic to connect from the internet.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:

   - **Security group name** — **`example-securitygroup`**
   - **Description** — **`Application security group`**
   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example**.
4. Add inbound rules to the security group.

   a. Determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com. An example of an IP address is `203.0.113.25/32`.

   In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, find the range of IP addresses used by client computers.

   > **Warning**
   >
   > If you use `0.0.0.0/0` for SSH access, you make it possible for all IP addresses to access your public instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your instances using SSH.

   b. In the **Inbound rules** section, choose **Add rule**.

   c. Set the following values for your new inbound rule to allow SSH access to your Amazon EC2 instance. After you do this, you can connect to your EC2 instance to install the application and other utilities. You also connect to your EC2 instance to upload content for your application.

      - **Type** — **SSH**
      - **Source** — The IP address or range you created from Step a, for example: **`203.0.113.25/32`**

   d. Choose **Add rule**.

   e. Set the following values for your new inbound rule to allow HTTP access to your application:

- **Type** — **`HTTP`**
- **Source** — **`0.0.0.0/0`**

5. Choose **Create security group** to create the security group.

   Note the security group ID because you need it later in another procedure.

Step 3: Create a VPC security group for a private cluster

To keep your cluster private, create a second security group for private access. To connect to private clusters in your VPC, you add inbound rules to your VPC security group that allow traffic from your application only.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:
   - **Security group name** — **`example-securitygroup`**
   - **Description** — **`Instance security group`**
   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example**
4. Add inbound rules to the security group.
   a. In the **Inbound rules** section, choose **Add rule**.
   b. Set the following values for your new inbound rule to allow DocumentDB traffic on port 27017 from your Amazon EC2 instance. After you do this, you can connect from your application to your cluster. By doing so, you can store and retrieve data from your application to your database.
      - **Type** — **`Custom TCP`**
      - **Source** — The identifier of the application security group that you created previously in this topic, for example: **sg-9edd5cfb**.
   c. Choose **Add rule**.
   d. Set the following values for your new inbound rule to allow HTTP access to your application:

- **Type** — **`HTTP`**
- **Source** — **`0.0.0.0/0`**

5. Choose **Create security group** to create the security group.

Step 4: Create a subnet group

A subnet group is a collection of subnets that you create in a VPC and that you then designate for your clusters. A subnet group makes it possible for you to specify a particular VPC when creating clusters.

To create a subnet group

1. Identify the private subnets for your database in the VPC.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   b. Choose **VPC Dashboard**, and then choose **Subnets**.

   c. Note the subnet IDs of the subnets you created in Step 1 named, for example: **example-subnet-private1-us-west-2a** and **example-subnet-private2-us-west-2b**. You need the subnet IDs when you create your subnet group.

2. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

   Make sure that you connect to the Amazon DocumentDB console, not to the Amazon VPC console.

3. In the navigation pane, choose **Subnet groups**.
4. Choose **Create**.
5. On the **Create subnet group** page, set these values in the **Subnet group details** section:

   - **Name** — **`example-db-subnet-group`**
   - **Description** — **`Instance security group`**

6. In the **Add subnets** section, set these values:

   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example**
   - **Availability Zones** — Select both Availability Zones created in Step 1. Example: **us-west-2a** and **us-west-2b**

- **Subnets** — Choose the private subnets you created in Step 1.

7. Choose **Create**.

Your new subnet group appears in the subnet groups list on the DocumentDB console. You can choose the subnet group to see details in the details pane. These details include all of the subnets associated with the group.

> **Note**
>
> If you created this VPC to associated it with a DocumentDB cluster, create the cluster by following the instructions in Creating an Amazon DocumentDB cluster.

Deleting a VPC

You can delete a VPC and the other resources that are used within it, if they are no longer needed.

> **Note**
>
> If you added resources in the VPC that you created in this topic, you might need to delete these before you can delete the VPC. For example, these resources might include Amazon EC2 instances or DocumentDB clusters. For more information, see Delete your VPC in the *Amazon VPC User Guide*.

To delete a VPC and related resources

1. Delete the subnet group:

   a. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

   b. In the navigation pane, choose **Subnet groups**.

   c. Select the subnet group you want to delete, such as **example-db-subnet-group**.

   d. Choose **Delete**, and then choose **Delete** in the confirmation window.

2. Note the VPC ID:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

b. Choose **VPC Dashboard**, and then choose **Your VPCs**.

c. In the list, identify the VPC that you created, such as **vpc-example**.

d. Note the **VPC ID** of the VPC that you created. You need the VPC ID in later steps.

3. Delete the security groups:

a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

b. Choose **VPC Dashboard**, and then choose **Security Groups**.

c. Select the security group for the Amazon DocumentDB cluster, such as **example-securitygroup**.

d. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation dialog.

e. Back on the **Security Groups** page, select the security group for the Amazon EC2 instance, such as **example-securitygroup**.

f. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation dialog.

4. Delete the VPC:

a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

b. Choose **VPC Dashboard**, and then choose **Your VPCs**.

c. Select the VPC you want to delete, such as **vpc-example**.

d. For **Actions**, choose Delete VPC.

The confirmation page shows other resources that are associated with the VPC that will also be deleted, including the subnets associated with it.

e. On the confirmation dialog, enter **`delete`**, and then choose **Delete**.

Create a dual-stack VPC for use with a DocumentDB cluster

A common scenario includes a cluster in a virtual private cloud (VPC) based on the Amazon VPC service. This VPC shares data with a public Amazon EC2 instance that is running in the same VPC. In this topic, you create the VPC for this scenario.

Topics

- Step 1: Create a VPC with private and public subnets

- Step 2: Create a VPC security group for a public Amazon EC2 instance
- Step 3: Create a VPC security group for a private cluster
- Step 4: Create a subnet group
- Step 5: Create an Amazon EC2 instance in dual-stack mode
- Step 6: Create a cluster in dual-stack mode
- Step 7: Connect to your Amazon EC2 instance and DB cluster
- Delete the VPC

In this procedure, you create the VPC for this scenario that works with a database running in dual-stack mode. Dual-stack mode enables connection over the IPv6 addressing protocol. For more information about IP addressing, see Amazon DocumentDB IP addressing.

Dual-stack network clusters are supported in most Regions. For more information see Dual-stack mode Region and version availability. To see the limitations of dual-stack mode, see Limitations for dual-stack network clusters.

This topic and the IPv4-only topic create the public and private subnets in the same VPC. For information about creating the Amazon DocumentDB cluster in one VPC and the Amazon EC2 instance in a different VPC, see Accessing an Amazon DocumentDB cluster in a VPC.

Your DocumentDB cluster needs to be available only to your Amazon EC2 instance, and not to the public internet. Thus, you create a VPC with both public and private subnets. The EC2 instance is hosted in the public subnet, so that it can reach the public internet. The cluster is hosted in a private subnet. The EC2 instance can connect to the cluster because it's hosted within the same VPC. However, the cluster is not available to the public internet, providing greater security.

The procedures in this topic configures an additional public and private subnet in a separate Availability Zone. These subnets aren't used by the procedure. An DocumentDB subnet group requires a subnet in at least two Availability Zones. The additional subnet makes it easy to configure more than one DocumentDB instance.

To create a cluster that uses dual-stack mode, specify **Dual-stack mod**e for the **Network type** setting. You can also modify a cluster with the same setting. For more information about creating a cluster, see Creating an Amazon DocumentDB cluster. For more information about modifying a DB cluster, see Modifying an Amazon DocumentDB cluster.

This topic describes configuring a VPC for Amazon DocumentDB clusters. For more information about Amazon VPC, see the *Amazon VPC User Guide*.

Step 1: Create a VPC with private and public subnets

Use the following procedure to create a VPC with both public and private subnets.

To create a VPC and subnets

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. In the top-right corner of the AWS Management Console, choose the Region to create your VPC in. This example uses the US West (Oregon) Region.
3. In the upper-left corner, choose **VPC dashboard**. To begin creating a VPC, choose **Create VPC**.
4. For **Resources to create** under **VPC settings**, choose **VPC and more**.
5. For the **VPC settings**, set these values:
   - **Name tag auto-generation** — `example-dual-stack`
   - **IPv4 CIDR block** — `10.0.0.0/16`
   - **IPv6 CIDR block** — **Amazon-provided IPv6 CIDR block**
   - **Tenancy** — **Default**
   - **Number of Availability Zones (AZs)** — **2**
   - **Customize AZs** — Keep the default values
   - **Number of public subnets** — **2**
   - **Number of private subnets** — **2**
   - **Customize subnets CIDR blocks** — Keep the default values
   - **NAT gateways ($)** — **None**
   - **Egress only internet gateway** — **No**
   - **VPC endpoints** — **None**
   - **DNS options** — Keep the default values
6. Choose **Create VPC**.

Step 2: Create a VPC security group for a public Amazon EC2 instance

Next, you create a security group for public access. To connect to public EC2 instances in your VPC, add inbound rules to your VPC security group that allow traffic to connect from the internet.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:
   - **Security group name** — **`example-dual-stack-securitygroup`**
   - **Description** — **`Dual-stack security group`**
   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example-dual-stack**.
4. Add inbound rules to the security group.
   a. Determine the IP address to use to connect to EC2 instances in your VPC using Secure Shell (SSH). To determine your public IP address, in a different browser window or tab, you can use the service at https://checkip.amazonaws.com.

      An example of an Internet Protocol version 4 (IPv4) address range is `203.0.113.25/32`. An example of an Internet Protocol version 6 (IPv6) address range is `2001:db8:1234:1a00::/64`.

      In many cases, you might connect through an internet service provider (ISP) or from behind your firewall without a static IP address. If so, find the range of IP addresses used by client computers.

      > **Warning**
      >
      > If you use `0.0.0.0/0` for IPv4 or `::0` for IPv6, you make it possible for all IP addresses to access your public instances using SSH. This approach is acceptable for a short time in a test environment, but it's unsafe for production environments. In production, authorize only a specific IP address or range of addresses to access your instances.

   b. In the **Inbound rules** section, choose **Add rule**.
   c. Set the following values for your new inbound rule to allow SSH access to your Amazon EC2 instance. After you do this, you can connect to your EC2 instance to install applications or other utilities. Specify an IP address so you can access your EC2 instance:
      - **Type** — **SSH**

- **Source** — The IP address or range you created from Step a. An example of an IPv4 address range is **`203.0.113.25/3`**2. An example of an IPv6 address range is **`2001:DB8::/32`**.

d. Choose **Add rule**.

5. Choose **Create security group** to create the security group.

   Note the security group ID because you need it later in another procedure.

Step 3: Create a VPC security group for a private cluster

To keep your cluster private, create a second security group for private access. To connect to private clusters in your VPC, add inbound rules to your VPC security group. These allow traffic from your Amazon EC2 instance only.

To create a VPC security group

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
2. Choose **VPC Dashboard**, choose **Security Groups**, and then choose **Create security group**.
3. On the **Create security group** page, set these values:
   - **Security group name** — **`example-dual-stack-cluster-securitygroup`**
   - **Description** — **`Dual-stack cluster security group`**
   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example-dual-stack**
4. Add inbound rules to the security group.

   a. In the **Inbound rules** section, choose **Add rule**.

   b. Set the following values for your new inbound rule to allow DocumentDB traffic on port 27017 from your Amazon EC2 instance. After you do this, you can connect from your EC2 instance to your cluster. By doing so, you can send data from your EC2 instance to your database.

      - **Type** — **`Custom TCP`**
      - **Source** — The identifier of the EC2 security group that you created previously in this topic, for example: **sg-9edd5cfb**.

5. Choose **Create security group** to create the security group.

Step 4: Create a subnet group

A subnet group is a collection of subnets that you create in a VPC and that you then designate for your clusters. By using a subnet group, you can specify a particular VPC when creating clusters. To create a subnet group that is DUAL compatible, all subnets must be DUAL compatible. To be DUAL compatible, a subnet must have an IPv6 CIDR associated with it.

To create a subnet group

1. Identify the private subnets for your database in the VPC.

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   b. Choose **VPC Dashboard**, and then choose **Subnets**.

   c. Note the subnet IDs of the subnets you created in Step 1 named, for example: **example-dual-stack-subnet-private1-us-west-2a** and **example-dual-stack-subnet-private2-us-west-2b**. You need the subnet IDs when you create your subnet group.

2. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

   Make sure that you connect to the Amazon DocumentDB console, not to the Amazon VPC console.

3. In the navigation pane, choose **Subnet groups**.

4. Choose **Create**.

5. On the **Create subnet group** page, set these values in the **Subnet group details** section:

   - **Name** — **`example-dual-stack-cluster-subnet-group`**
   - **Description** — **`Dual-stack cluster subnet group`**

6. In the **Add subnets** section, set these values:

   - **VPC** — Choose the VPC that you created earlier, for example: **vpc-example-dual-stack**
   - **Availability Zones** — Select both Availability Zones created in Step 1. Example: **us-west-2a** and **us-west-2b**
   - **Subnets** — Choose the private subnets you created in Step 1.

7. Choose **Create**.

Your new subnet group appears in the subnet groups list on the DocumentDB console. You can choose the subnet group to see details in the details pane. These details include all of the subnets associated with the group.

Step 5: Create an Amazon EC2 instance in dual-stack mode

To create an Amazon EC2 instance, follow the instructions in Launch an EC2 instance using the launch instance wizard in the console in the *Amazon Elastic Compute Cloud User Guide*.

On the **Configure Instance Details** page, set these values and keep the other values as their defaults:

- **Network** — Choose an existing VPC with both public and private subnets, such as **vpc-example-dual-stack-vpc** (vpc-*identifier*) created in Step 1: Create a VPC with private and public subnets.
- **Subnet** — Choose an existing public subnet, such as **subnet-*identifier* | example-dual-stack-subnet-public1-us-east-2a | us-east-2a** created in Step 2: Create a VPC security group for a public Amazon EC2 instance.
- **Auto-assign Public IP** — Choose **Enable**.
- **Auto-assign IPv6 IP** — Choose **Enable**.
- **Firewall (security groups)** Choose **Select an existing security group**.—
- **Common security groups** — Choose an existing security group, such as the **example-dual-stack-securitygroup** created in Step 2: Create a VPC security group for a public Amazon EC2 instance. Make sure that the security group that you choose includes inbound rules for Secure Shell (SSH) and HTTP access.

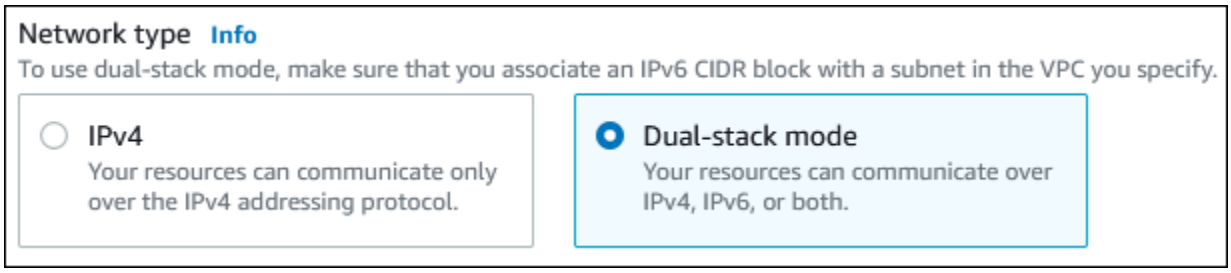
Step 6: Create a cluster in dual-stack mode

In this step, you create a DB cluster that runs in dual-stack mode. **!!! Note: this section needs editing once the IPv6 updates have been made to the console !!!**

To create a cluster in dual-stack mode

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the upper-right corner of the console, choose the AWS Region where you want to create the DocumentDB cluster. This example uses the US East (Ohio) Region.

3. In the navigation pane, choose **Clusters**.
4. On the **Clusters** list page, choose **Create**.
5. On the **Create Amazon DocumentDB cluster** page, make sure that the **Instance-based cluster** option is chosen.
6. In the **Connectivity** section, under **Network type**, choose **Dual-stack mode**.



7. At the bottom of the page, switch on **Show advanced settings**.
8. In the **Network settings** section, set these values:

   - **Virtual private cloud (VPC)** — Choose an existing VPC with both public and private subnets, such as **vpc-example-dual-stack** (vpc-*identifier*) created in Step 1: Create a VPC with private and public subnets.

     The VPC must have subnets in different Availability Zones.
   - **Subnet group** — Choose a subnet group for the VPC, such as **example-dual-stack-cluster-subnet-group** created in Step 4: Create a subnet group.
   - **Public access** — Choose **No**.
   - **VPC security group (firewall)** — Select **Choose existing**.
   - **Existing VPC security groups** — Choose an existing VPC security group that is configured for private access, such as **example-dual-stack-cluster-securitygroup** created in Step 3: Create a VPC security group for a private cluster.

     Remove other security groups, such as the default security group, by choosing the **X** associated with each.
   - **Availability Zone** — Choose the Availability Zone you created in Step 1. Example: **us-west-2a**.

     To avoid cross-AZ traffic, make sure the cluster and the EC2 instance are in the same Availability Zone.
9. For the remaining sections, specify your cluster settings. For information about each setting, see Creating an Amazon DocumentDB cluster.

Step 7: Connect to your Amazon EC2 instance and DB cluster

After you create your Amazon EC2 instance and DocumentDB cluster in dual-stack mode, you can connect to each one using the IPv6 protocol. To connect to an EC2 instance using the IPv6 protocol, follow the instructions in Connect your EC2 instance in the *Amazon Elastic Compute Cloud User Guide*.

To connect to your DocumentDB cluster from the EC2 instance, follow the instructions in Step 5: Install the MongoDB Shell, in the Connect Amazon EC2 manually topic (and continue with the subsequent Step 6 and Step 7 in the same procedure).

Delete the VPC

You can delete a VPC and the other resources that are used within it, if they are no longer needed.

> **Note**
>
> If you added resources in the VPC that you created in this topic, you might need to delete these before you can delete the VPC. For example, these resources might include Amazon EC2 instances or DocumentDB clusters. For more information, see Delete your VPC in the *Amazon VPC User Guide*.

To delete a VPC and related resources

1. Delete the subnet group:
   a. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
   b. In the navigation pane, choose **Subnet groups**.
   c. Select the subnet group you want to delete, such as **example-dual-stack-cluster-subnet-group**.
   d. Choose **Delete**, and then choose **Delete** in the confirmation window.
2. Note the VPC ID:
   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.
   b. Choose **VPC Dashboard**, and then choose **Your VPCs**.
   c. In the list, identify the VPC that you created, such as **vpc-example-dual-stack**.

d. Note the **VPC ID** of the VPC that you created. You need the VPC ID in later steps.

3. Delete the security groups:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   b. Choose **VPC Dashboard**, and then choose **Security Groups**.

   c. Select the security group for the Amazon DocumentDB cluster, such as **example-dual-stack-securitygroup**.

   d. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation dialog.

   e. Back on the **Security Groups** page, select the security group for the Amazon EC2 instance, such as **example-securitygroup**.

   f. For **Actions**, choose **Delete security groups**, and then choose **Delete** on the confirmation dialog.

4. Delete the NAT gateway:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   b. Choose **VPC Dashboard**, and then choose **Security Groups**.

   c. Select the NAT gateway of the VPC that you created. Use the VPC ID to identify the correct NAT gateway.

   d. For **Actions**, choose **Delete NAT gateway**.

   e. On the confirmation dialog, enter **`delete`**, and then choose **Delete**.

5. Delete the VPC:

   a. Open the Amazon VPC console at https://console.aws.amazon.com/vpc.

   b. Choose **VPC Dashboard**, and then choose **Your VPCs**.

   c. Select the VPC you want to delete, such as **vpc-example-dual-stack**.

   d. For **Actions**, choose Delete VPC.

      The confirmation page shows other resources that are associated with the VPC that will also be deleted, including the subnets associated with it.

   e. On the confirmation dialog, enter **`delete`**, and then choose **Delete**.

6. Release the elastic IP address:

   a. Open the EC2 console at https://console.aws.amazon.com/ec2.

b. Choose **EC2 Dashboard**, and then choose **Elastic IPs**.

c. Select the Elastic IP address that you want to release.

d. For **Actions**, choose **Release Elastic IP addresses**.

e. On the confirmation dialog, choose **Release**.

Backing up and restoring in Amazon DocumentDB

Amazon DocumentDB (with MongoDB compatibility) continuously backs up your data to Amazon Simple Storage Service (Amazon S3) for 1–35 days so that you can quickly restore to any point within the backup retention period. Amazon DocumentDB also takes automatic snapshots of your data as part of this continuous backup process.

> **Note**
>
> These are service-managed Amazon S3 buckets and you will not have access to the backup files. To control your own backups, see Dumping, restoring, importing, and exporting data.

You can also retain backup data beyond the backup retention period by creating a manual snapshot of your cluster's data. The backup process does not impact your cluster's performance.

This section discusses the use cases for the backup capabilities in Amazon DocumentDB and shows you how to manage backups for your Amazon DocumentDB clusters.

Topics

- Back up and restore: concepts
- Understanding backup storage usage
- Dumping, restoring, importing, and exporting data
- Cluster snapshot considerations
- Comparing automatic and manual snapshots
- Creating a manual cluster snapshot
- Copying Amazon DocumentDB cluster snapshots
- Sharing Amazon DocumentDB cluster snapshots
- Restoring from a cluster snapshot
- Restoring to a point in time
- Deleting a cluster snapshot

Back up and restore: concepts

Noun	Description	APIs (Verbs)

Backup retention period	A period of time between 1 and 35 days for which you can perform a point-in-time restore.	`create-db-cluster`
		`modify-db-cluster`
		`restore-db-cluster-to-point-in-time`

Amazon DocumentDB storage volume	Highly available and highly durable storage volume that replicates data six ways across three Availability Zones. An Amazon DocumentDB cluster is highly durable	`create-db-cluster` `delete-db-cluster`

Noun	Description	APIs (Verbs)
	regardless of the number of instances in the cluster.	

Backup window	Period of time in the day in which automatic snapshots are taken.	`create-db-cluster`
		`describe-db-cluster`
		`modify-db-cluster`

Automatic snapshot	Daily snapshots that are full backups of cluster and are automatically created by the continuous backup process in Amazon DocumentDB.	`restore-db-cluster-from-snapshot`
		`describe-db-cluster-snapshot-attributes`
		`describe-db-cluster-snapshots`

Noun	Description	APIs (Verbs)

Manual snapshot	Snapshots you create manually to retain full backups of a cluster beyond the backup period.	`create-db-cluster-snapshot`
		`copy-db-cluster-snapshot`
		`delete-db-cluster-snapshot`
		`describe-db-cluster-snapshot-attributes`
		`describe-db-cluster-snapshots`
		`modify-db-cluster-snapshot-attribute`

Understanding backup storage usage

Amazon DocumentDB backup storage consists of continuous backups within the backup retention period and manual snapshots outside the retention period. To control your backup storage usage, you can reduce the backup retention interval, remove old manual snapshots when they are no longer needed, or both. For general information about Amazon DocumentDB backups, see Backing up and restoring in Amazon DocumentDB. For pricing information about Amazon DocumentDB backup storage, see Amazon DocumentDB Pricing.

To control your costs, you can monitor the amount of storage consumed by continuous backups and manual snapshots that persist beyond the retention period. Then you can reduce the backup retention interval and remove manual snapshots when they are no longer needed.

You can use the Amazon CloudWatch metrics `TotalBackupStorageBilled`, `SnapshotStorageUsed`, and `BackupRetentionPeriodStorageUsed` to review and monitor the amount of storage used by your Amazon DocumentDB backups, as follows:

- `BackupRetentionPeriodStorageUsed` represents the amount of backup storage used for storing continuous backups at the current time. This metric value depends on the size of the cluster volume and the number of changes you make during the retention period. However, for billing purposes the metric does not exceed the cumulative cluster volume size during the retention period. For example, if your cluster size is 100 GiB and your retention period is two

days, the maximum value for `BackRetentionPeriodStorageUsed` is 200 GiB (100 GiB + 100 GiB).

- `SnapshotStorageUsed` represents the amount of backup storage used for storing manual snapshots beyond the backup retention period. Manual snapshots taken within the retention period do not count against your backup storage. Similarly, automatic snapshots do not count against your backup storage. The size of each snapshot is the size of the cluster volume at the time you take the snapshot. The `SnapshotStorageUsed` value depends on the number of snapshots you keep and the size of each snapshot. For example, suppose that you have one snapshot outside the retention period and cluster volume size was 100 GiB when that snapshot was taken. The amount of `SnapshotStorageUsed` is 100 GiB.

- `TotalBackupStorageBilled` represents the sum of `BackupRetentionPeriodStorageUsed` and `SnapshotStorageUsed`, minus an amount of free backup storage equal to the size of cluster volume for one day. For example, if your cluster size is 100 GiB, you have one retention day, and you have one snapshot outside the retention period, the `TotalBackupStorageBilled` is 100 GiB (100 GiB + 100 GiB - 100 GiB).

- These metrics are computed independently for each Amazon DocumentDB cluster.

You can monitor your Amazon DocumentDB clusters and build reports using CloudWatch metrics through the CloudWatch console. For more information about how to use CloudWatch metrics, see Monitoring and logging in Amazon DocumentDB.

Dumping, restoring, importing, and exporting data

You can use the `mongodump`, `mongorestore`, `mongoexport`, and `mongoimport` utilities to move data in and out of your Amazon DocumentDB cluster. This section discusses the purpose of each of these tools and configurations to help you achieve better performance.

MongoDB Database Tools version requirement

Use MongoDB Database Tools up to and including version 100.11.0 for Amazon DocumentDB. To download, see MongoDB Database Tools releases on the MongoDB website.

Topics

- mongodump
- mongorestore
- mongoexport
- mongoimport
- Tutorial

mongodump

The mongodump utility creates a binary (BSON) backup of a MongoDB database. The mongodump tool is the preferred method of dumping data from your source MongoDB deployment when looking to restore it into your Amazon DocumentDB cluster due to the size efficiencies achieved by storing the data in a binary format.

Depending on the resources available on the instance or machine you are using to perform the command, you can speed up your mongodump by increasing the number of parallel collections dumped from the default 1 using the `--numParallelCollections` option. A good rule of thumb is to start with one worker per vCPU on your Amazon DocumentDB cluster's primary instance.

Example usage

The following is an example usage of the mongodump utility in the Amazon DocumentDB cluster, `sample-cluster`.

```
mongodump --ssl \
    --host="sample-cluster.node.us-east-1.docdb.amazonaws.com:27017" \
    --collection=sample-collection \
    --db=sample-database \
    --out=sample-output-file \
    --numParallelCollections 4  \
```

```
    --username=sample-user \
    --password=abc0123 \
    --sslCAFile global-bundle.pem
```

mongorestore

The mongorestore utility enables you to restore a binary (BSON) backup of a database that was created with the mongodump utility. You can improve restore performance by increasing the number of workers for each collection during the restore with the --numInsertionWorkersPerCollection option (the default is 1). A good rule of thumb is to start with one worker per vCPU on your Amazon DocumentDB cluster's primary instance.

Example usage

The following is an example usage of the mongorestore utility in the Amazon DocumentDB cluster, sample-cluster.

```
mongorestore --ssl \
    --host="sample-cluster.node.us-east-1.docdb.amazonaws.com:27017" \
    --username=sample-user \
    --password=abc0123 \
    --sslCAFile global-bundle.pem <fileToBeRestored>
```

mongoexport

The mongoexport tool exports data in Amazon DocumentDB to JSON, CSV, or TSV file formats. The mongoexport tool is the preferred method of exporting data that needs to be human or machine readable.

> **Note**
>
> mongoexport does not directly support parallel exports. However, it is possible to increase performance by executing multiple mongoexport jobs concurrently for different collections.

Example usage

The following is an example usage of the mongoexport tool in the Amazon DocumentDB cluster, sample-cluster.

```
mongoexport --ssl \
    --host="sample-cluster.node.us-east-1.docdb.amazonaws.com:27017" \
    --collection=sample-collection \
    --db=sample-database \
    --out=sample-output-file \
    --username=sample-user \
    --password=abc0123 \
    --sslCAFile global-bundle.pem
```

`mongoimport`

The `mongoimport` tool imports the contents of JSON, CSV, or TSV files into an Amazon DocumentDB cluster. You can use the `--numInsertionWorkers` parameter to parallelize and speed up the import (the default is 1).

Example usage

The following is an example usage of the `mongoimport` tool in the Amazon DocumentDB cluster, `sample-cluster`.

```
mongoimport --ssl \
    --host="sample-cluster.node.us-east-1.docdb.amazonaws.com:27017" \
    --collection=sample-collection \
    --db=sample-database \
    --file=<yourFile> \
    --numInsertionWorkers 4 \
    --username=sample-user \
    --password=abc0123 \
    --sslCAFile global-bundle.pem
```

Tutorial

The following tutorial describes how to use the mongodump, `mongorestore`, `mongoexport`, and `mongoimport` utilities to move data in and out of an Amazon DocumentDB cluster.

1. **Prerequisites** — Before you begin, ensure that your Amazon DocumentDB cluster is provisioned and that you have access to an Amazon EC2 instance in the same VPC as your cluster. For more information, see Connect using Amazon EC2.

   To be able to use the mongo utility tools, you must have the mongodb-org-tools package installed in your EC2 instance, as follows.

```
sudo yum install mongodb-org-tools-4.0.18
```

Because Amazon DocumentDB uses Transport Layer Security (TLS) encryption by default, you must also download the Amazon RDS certificate authority (CA) file to use the mongo shell to connect, as follows.

```
wget https://truststore.pki.rds.amazonaws.com/global/global-bundle.pem
```

2. **Download sample data** — For this tutorial, you will download some sample data that contains information about restaurants.

```
wget https://raw.githubusercontent.com/ozlerhakan/mongodb-json-files/master/
datasets/restaurant.json
```

3. **Import the sample data into Amazon DocumentDB** — Since the data is in a logical JSON format, you will use the `mongoimport` utility to import the data into your Amazon DocumentDB cluster.

```
mongoimport --ssl \
    --host="tutorialCluster.amazonaws.com:27017" \
    --collection=restaurants \
    --db=business \
    --file=restaurant.json \
    --numInsertionWorkers 4 \
    --username=<yourUsername> \
    --password=<yourPassword> \
    --sslCAFile global-bundle.pem
```

4. **Dump the data with `mongodump`** — Now that you have data in your Amazon DocumentDB cluster, you can take a binary dump of that data using the `mongodump` utility.

```
mongodump --ssl \
    --host="tutorialCluster.us-east-1.docdb.amazonaws.com:27017"\
    --collection=restaurants \
    --db=business \
    --out=restaurantDump.bson \
    --numParallelCollections 4 \
    --username=<yourUsername> \
    --password=<yourPassword> \
```

```
    --sslCAFile global-bundle.pem
```

5. **Drop the `restaurants` collection** — Before you restore the `restaurants` collection in the `business` database, you have to first drop the collection that already exists in that database, as follows.

```
use business
```

```
db.restaurants.drop()
```

6. **Restore the data with `mongorestore`** — With the binary dump of the data from Step 3, you can now use the `mongorestore` utility to restore your data to your Amazon DocumentDB cluster.

```
mongorestore --ssl \
    --host="tutorialCluster.us-east-1.docdb.amazonaws.com:27017" \
    --numParallelCollections 4 \
    --username=<yourUsername> \
    --password=<yourPassword> \
    --sslCAFile global-bundle.pem restaurantDump.bson
```

7. **Export the data using `mongoexport`** — To complete the tutorial, export the data from your cluster in the format of a JSON file, no different than the file you imported in Step 1.

```
mongoexport --ssl \
    --host="tutorialCluster.node.us-east-1.docdb.amazonaws.com:27017" \
    --collection=restaurants \
    --db=business \
    --out=restaurant2.json \
    --username=<yourUsername> \
    --password=<yourPassword> \
    --sslCAFile global-bundle.pem
```

8. **Validation** — You can validate that the output of Step 5 yields the same result as Step 1 with the following commands.

```
wc -l restaurant.json
```

Output from this command:

```
2548 restaurant.json
```

```
wc -l restaurant2.json
```

Output from this command:

```
2548 restaurant2.json
```

Cluster snapshot considerations

Amazon DocumentDB creates daily automatic snapshots of your cluster during your cluster's backup window. Amazon DocumentDB saves the automatic snapshots of your cluster according to the backup retention period that you specify. If necessary, you can recover your cluster to any point in time during the backup retention period. Automatic snapshots don't occur while a copy operation is executing in the same Region for the same cluster.

Topics

- Backup storage
- Backup window
- Backup retention period
- Copy cluster snapshot encryption

In addition to automatic cluster snapshots, you can also manually create a cluster snapshot. You can copy both automatic and manual snapshots. For more information, see Creating a manual cluster snapshot and Copying Amazon DocumentDB cluster snapshots.

> **Note**
>
> Your cluster must be in the *available* state for an automatic snapshot to be taken.
> You can't share an Amazon DocumentDB automated cluster snapshot. As a workaround, you can create a manual snapshot by copying the automated snapshot, and then share that copy. For more information about copying a snapshot, see Copying Amazon DocumentDB cluster snapshots. For more information about restoring a cluster from a snapshot, see Restoring from a cluster snapshot.

Backup storage

Your Amazon DocumentDB backup storage for each AWS Region is composed of the backup storage needed for your backup retention period, which includes automatic and manual cluster snapshots in that Region. The default backup retention period is 1 day. For more information about backup storage pricing, see Amazon DocumentDB Pricing.

When you delete a cluster, all of its automatic snapshots are deleted and cannot be recovered. However, manual snapshots are not deleted when you delete a cluster. if you choose to have Amazon DocumentDB create a final snapshot (manual snapshot) before your cluster is deleted, you can use the final snapshot to recover your cluster.

For more information on snapshots and storage, see Understanding backup storage usage.

Backup window

Automatic snapshots occur daily during the preferred backup window. If the snapshot requires more time than allotted to the backup window, the backup process continues until it finishes, even though the backup window has ended. The backup window can't overlap with the weekly maintenance window for the cluster.

If you don't specify a preferred backup window when you create the cluster, Amazon DocumentDB assigns a default 30-minute backup window. This window is chosen at random from an 8-hour block of time associated with your cluster's Region. You can change your preferred backup window by modifying the cluster. For more information, see Modifying an Amazon DocumentDB cluster.

Region Name	Region	UTC Time Block
US East (Ohio)	us-east-2	03:00-11:00
US East (N. Virginia)	us-east-1	03:00-11:00
US West (Oregon)	us-west-2	06:00-14:00
Africa (Cape Town)	af-south-1	03:00–11:00
Asia Pacific (Hong Kong)	ap-east-1	06:00-14:00
Asia Pacific (Taipei)	ap-east-2	09:00-17:00

Region Name	Region	UTC Time Block
Asia Pacific (Hyderabad)	ap-south-2	06:30–14:30
Asia Pacific (Malaysia)	ap-southeast-5	13:00-21:00
Asia Pacific (Mumbai)	ap-south-1	06:00-14:00
Asia Pacific (Osaka)	ap-northeast-3	12:00-20:00
Asia Pacific (Seoul)	ap-northeast-2	13:00-21:00
Asia Pacific (Singapore)	ap-southeast-1	14:00-22:00
Asia Pacific (Sydney)	ap-southeast-2	12:00-20:00
Asia Pacific (Jakarta)	ap-southeast-3	08:00-16:00
Asia Pacific (Melbourne)	ap-southeast-4	11:00-19:00
Asia Pacific (New Zealand)	ap-southeast-6	13:00-21:00
Asia Pacific (Thailand)	ap-southeast-7	15:00-23:00
Asia Pacific (Tokyo)	ap-northeast-1	13:00-21:00
Canada (Central)	ca-central-1	03:00-11:00
Canada West (Calgary)	ca-west-1	18:00-02:00
China (Beijing)	cn-north-1	06:00-14:00
China (Ningxia)	cn-northwest-1	06:00-14:00
Europe (Frankfurt)	eu-central-1	21:00-05:00
Europe (Zurich)	eu-central-2	02:00-10:00
Europe (Ireland)	eu-west-1	22:00-06:00
Europe (London)	eu-west-2	22:00-06:00

Region Name	Region	UTC Time Block
Europe (Milan)	eu-south-1	02:00-10:00
Europe (Paris)	eu-west-3	23:59-07:29
Europe (Spain)	eu-south-2	02:00–10:00
Europe (Stockholm)	eu-north-1	04:00–12:00
Mexico (Central)	mx-central-1	03:00-11:00
Middle East (UAE)	me-central-1	05:00–13:00
South America (São Paulo)	sa-east-1	00:00-08:00
Israel (Tel Aviv)	il-central-1	04:00-12:00
AWS GovCloud (US-East)	us-gov-east-1	17:00-01:00
AWS GovCloud (US-West)	us-gov-west-1	06:00-14:00

Backup retention period

The backup retention period is the number of days an automatic backup is retained before being automatically deleted. Amazon DocumentDB supports a backup retention period of 1–35 days.

You can set the backup retention period when you create a cluster. If you don't explicitly set the backup retention period, the default backup retention period of 1 day is assigned to your cluster. After you create a cluster, you can modify the backup retention period by modifying the cluster using either the AWS Management Console or the AWS CLI. For more information, see Modifying an Amazon DocumentDB cluster.

Copy cluster snapshot encryption

Cluster and snapshot encryption is based on a KMS encryption key. The KMS key ID is the Amazon Resource Name (ARN), KMS key identifier, or the KMS key alias for the KMS encryption key.

The following guidelines and limitations apply:

- Encryption is inferred from the cluster when creating a snapshot. If the cluster is encrypted, the snapshot of that cluster is encrypted with the same KMS key. If the cluster is not encrypted, the snapshot is not encrypted.
- If you copy an encrypted cluster snapshot from your Amazon Web Services account, you can specify a value for `KmsKeyId` to encrypt the copy with a new KMS encryption key. If you don't specify a value for `KmsKeyId` , then the copy of the cluster snapshot is encrypted with the same KMS key as the source cluster snapshot.
- If you copy an encrypted cluster snapshot that is shared from another Amazon Web Services account, then you must specify a value for `KmsKeyId`.
- To copy an encrypted cluster snapshot to another Amazon Web Services Region, set `KmsKeyId` to the KMS key ID that you want to use to encrypt the copy of the cluster snapshot in the destination Region. KMS encryption keys are specific to the Amazon Web Services Region that they are created in, and you can't use encryption keys from one Amazon Web Services Region in another Amazon Web Services Region.
- If you copy an unencrypted cluster snapshot and specify a value for the `KmsKeyId` parameter, an error is returned.

Comparing automatic and manual snapshots

The following are key features of Amazon DocumentDB (with MongoDB compatibility) automatic and manual snapshots.

Amazon DocumentDB automatic snapshots have the following key features:

- **Automatic snapshot naming** — Automatic snapshot names follow the pattern `rds:<cluster-name>-yyyy-mm-dd-hh-mm`, with `yyyy-mm-dd-hh-mm` representing the date and time the snapshot was created.
- **Created automatically on a schedule** — When you create or modify a cluster, you can set the *backup retention period* to an integer value from 1 to 35 days. By default, new clusters have a backup retention period of 1 day. The backup retention period defines the number of days that automatic snapshots are kept before being automatically deleted. You can't disable automatic backups on Amazon DocumentDB clusters.

  In addition to setting the backup retention period, you also set the *backup window*, the time of day during which automatic snapshots are created.

- **Deleting automatic snapshots** — Automatic snapshots are deleted when you delete the automatic snapshot's cluster. You can't manually delete an automatic snapshot.
- **Incremental** — During the backup retention period, database updates are recorded so that there is an incremental record of changes.
- **Restoring from an automatic snapshot** — You can restore from an automatic snapshot using the AWS Management Console or the AWS CLI. When you restore from a snapshot using the AWS CLI, you must add instances separately after the cluster is *available*.
- **Sharing** — You can't share an Amazon DocumentDB automated cluster snapshot. As a workaround, you can create a manual snapshot by copying the automated snapshot, and then share that copy. For more information about copying a snapshot, see Copying Amazon DocumentDB cluster snapshots. For more information about restoring a cluster from a snapshot, see Restoring from a cluster snapshot.
- **You can restore from any point within the backup retention period** — Because database updates are incrementally recorded, you can restore your cluster to any point in time within the backup retention period.

  When you restore from an automatic snapshot or from a point-in-time restore using the AWS CLI, you must add instances separately after the cluster is *available*.

Amazon DocumentDB manual snapshots have the following key features:

- **Created on demand** — Amazon DocumentDB manual snapshots are created on demand using the Amazon DocumentDB Management Console or AWS CLI.
- **Deleting a manual snapshot** —A manual snapshot is deleted only when you explicitly delete it using either the Amazon DocumentDB console or AWS CLI. A manual snapshot is not deleted when you delete its cluster.
- **Full backups** — When a manual snapshot is taken, a full backup of your cluster's data is created and stored.
- **Manual snapshot naming** — You specify the manual snapshot name. Amazon DocumentDB does not add a `datetime` stamp to the name, so you must add that information if you want it included in the name.
- **Restoring from a manual snapshot** —You can restore from a manual snapshot using the console or the AWS CLI. When you restore from a snapshot using the AWS CLI, you must add instances separately after the cluster is *available*.
- **Service Quotas** — You are limited to a maximum of 100 manual snapshots per AWS Region.

- **Sharing** — You can share manual cluster snapshots, which can be copied by authorized AWS accounts. You can share encrypted or unencrypted manual snapshots. For more information about copying a snapshot, see Copying Amazon DocumentDB cluster snapshots.
- **You restore to when the manual snapshot was taken** —When you restore from a manual snapshot, you restore to when the manual snapshot was taken.

  When you restore from a snapshot using the AWS CLI, you must add instances separately after the cluster is *available*.

Creating a manual cluster snapshot

You can create a manual snapshot using either the AWS Management Console or AWS CLI. The amount of time it takes to create a snapshot varies with the size of your databases. When you create a snapshot, you must do the following:

1. Identify which cluster to back up.
2. Give your snapshot a name. This allows you to restore from it later.

Using the AWS Management Console

To create a manual snapshot using the AWS Management Console, you can follow either method below.

1. **Method 1:**

   1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
   2. In the navigation pane, choose **Snapshots**.

      > **Tip**
      >
      > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

   3. On the **Snapshots** page, choose **Create**.

4. On the **Create cluster snapshot** page:

   a. **Cluster identifier** — From the drop-down list of clusters, choose the cluster that you want to create a snapshot of.

   b. **Snapshot identifier** — Enter a name for your snapshot.

      Snapshot naming constraints:

      - Length is [1–255] letters, numbers, or hyphens.
      - First character must be a letter.
      - Cannot end with a hyphen or contain two consecutive hyphens.
      - Must be unique for all clusters (across Amazon RDS, Amazon Neptune, and Amazon DocumentDB) per AWS account, per Region.

   c. Choose **Create**.

2. **Method 2:**

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. On the **Clusters** page, choose the button to the left of the cluster that you want to snapshot.
4. From the **Actions** menu, choose **Take snapshot**.
5. On the **Create cluster snapshot** page:

   a. **Snapshot identifier** — Enter a name for your snapshot.

      Snapshot naming constraints:

      - Length is [1–63] letters, numbers, or hyphens.
      - First character must be a letter.
      - Cannot end with a hyphen or contain two consecutive hyphens.

- Must be unique for all clusters (across Amazon RDS, Amazon Neptune, and Amazon DocumentDB) per AWS account, per Region.

b. Choose **Create**.

Using the AWS CLI

To create a cluster snapshot using the AWS CLI, use the `create-db-cluster-snapshot` operation with the following parameters.

Parameters

- **`--db-cluster-identifier`** — Required. The name of the cluster that you are taking a snapshot of. This cluster must exist and be *available*.
- **`--db-cluster-snapshot-identifier`** — Required. The name of the manual snapshot that you are creating.

The following example creates a snapshot named `sample-cluster-snapshot` for a cluster named `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb create-db-cluster-snapshot \
    --db-cluster-identifier sample-cluster \
    --db-cluster-snapshot-identifier sample-cluster-snapshot
```

For Windows:

```
aws docdb create-db-cluster-snapshot ^
    --db-cluster-identifier sample-cluster ^
    --db-cluster-snapshot-identifier sample-cluster-snapshot
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshot": {
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1b",
            "us-east-1c"
```

```
        ],
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot",
        "DBClusterIdentifier": "sample-cluster",
        "SnapshotCreateTime": "2020-04-24T04:59:08.475Z",
        "Engine": "docdb",
        "Status": "creating",
        "Port": 0,
        "VpcId": "vpc-abc0123",
        "ClusterCreateTime": "2020-01-10T22:13:38.261Z",
        "MasterUsername": "master-user",
        "EngineVersion": "4.0.0",
        "SnapshotType": "manual",
        "PercentProgress": 0,
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DBClusterSnapshotArn": "arn:aws:rds:us-east-1:<accountID>:cluster-
snapshot:sample-cluster-snapshot"
    }
}
```

Copying Amazon DocumentDB cluster snapshots

In Amazon DocumentDB, you can copy snapshots within the same AWS Region or to a different AWS Region. You can also copy shared snapshots to your account in the same AWS Region or in a different AWS Region. For more information on sharing snapshots, see Sharing Amazon DocumentDB cluster snapshots.

> **Note**
>
> Amazon DocumentDB bills you based upon the amount of backup and snapshot data you keep and the period of time that you keep it. For more information about the storage associated with Amazon DocumentDB backups and snapshots, see Understanding backup storage usage. For pricing information about Amazon DocumentDB storage, see Amazon DocumentDB Pricing.

Topics

- Copying shared snapshots
- Copying snapshots across AWS Regions

- Limitations
- Handling encryption
- Parameter group considerations
- Copying a cluster snapshot

Copying shared snapshots

You can copy snapshots shared to you by other AWS accounts. If you are copying an encrypted snapshot that has been shared from another AWS account, you must have access to the AWS KMS encryption key that was used to encrypt the snapshot. For more information, see Handling encryption.

Copying snapshots across AWS Regions

When you copy a snapshot to an AWS Region that is different from the source snapshot's AWS Region, each copy is a full snapshot. A full snapshot copy contains all of the data and meta data required to restore the Amazon DocumentDB cluster.

Limitations

The following are some limitations when you copy snapshots:

- If you delete a source snapshot before the target snapshot becomes available, the snapshot copy may fail. Verify that the target snapshot has a status of `AVAILABLE` before you delete a source snapshot.
- You can have up to five snapshot copy requests in progress to a single destination Region per account.
- Depending on the AWS Regions involved and the amount of data to be copied, a cross-Region snapshot copy can take hours to complete. In some cases, there might be a large number of cross-Region snapshot copy requests from a given source AWS Region. In these cases, Amazon DocumentDB might put new cross-Region copy requests from that source AWS Region into a queue until some in-progress copies complete. No progress information is displayed about copy requests while they are in the queue. Progress information is displayed when the copy starts.

Handling encryption

You can copy a snapshot that has been encrypted using an AWS KMS encryption key. If you copy an encrypted snapshot, the copy of the snapshot must also be encrypted. If you copy an encrypted snapshot within the same AWS Region, you can encrypt the copy with the same AWS KMS encryption key as the original snapshot, or you can specify a different AWS KMS encryption key. If you copy an encrypted snapshot across Regions, you can't use the same AWS KMS encryption key for the copy as used for the source snapshot, because AWS KMS keys are Region-specific. Instead, you must specify an AWS KMS key valid in the destination AWS Regionn.

The source snapshot remains encrypted throughout the copy process. For more information, see Data protection in Amazon DocumentDB.

> **Note**
>
> For Amazon DocumentDB cluster snapshots, you can't encrypt an unencrypted cluster snapshot when you copy the snapshot.

Parameter group considerations

When you copy a snapshot across Regions, the copy doesn't include the parameter group used by the original Amazon DocumentDB cluster. When you restore a snapshot to create a new cluster, that cluster gets the default parameter group for the AWS Region it is created in. To give the new cluster the same parameters as the original, you must do the following:

1. In the destination AWS Region, create an Amazon DocumentDB cluster parameter group with the same settings as the original cluster. If one already exists in the new AWS Region, you can use that one.
2. After you restore the snapshot in the destination AWS Region, modify the new Amazon DocumentDB cluster and add the new or existing parameter group from the previous step. For more information, see Modifying an Amazon DocumentDB cluster.

Copying a cluster snapshot

You can copy an Amazon DocumentDB cluster using the AWS Management Console or the AWS CLI, as follows.

Using the AWS Management Console

To make a copy of a cluster snapshot using the AWS Management Console, complete the following steps. This procedure works for copying encrypted or unencrypted cluster snapshots, in the same AWS Region or across Regions.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Snapshots**, and then choose the button to the left of the snapshot that you want to copy.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. From the **Actions** menu, choose **Copy**.
4. In the resulting **Make Copy of cluster snapshot** page, complete the **Settings** section.

   a. **Destination Region** — Optional. To copy the cluster snapshot to a different AWS Region, choose that AWS Region for **Destination Region**.

   b. **New snapshot identifier** — Enter a name for the new snapshot.

      Target snapshot naming constraints:

      - Cannot be the name of an existing snapshot.
      - Length is [1—63] letters, numbers, or hyphens.
      - First character must be a letter.
      - Cannot end with a hyphen or contain two consecutive hyphens.
      - Must be unique for all clusters across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.

   c. **Copy tags** — To copy any tags you have on your source snapshot to your snapshot copy, choose **Copy tags**.

5. Complete the **Encryption-at-rest** section.

a. **Encryption at rest** — If your snapshot is not encrypted, these options are not available to you because you cannot create an encrypted copy from an unencrypted snapshot. If your snapshot is encrypted, you can change the AWS KMS key used during encryption at rest.

   For more information about encrypting snapshot copies, see Copy cluster snapshot encryption.

   For more information about encryption at rest, see Encrypting Amazon DocumentDB data at rest.

b. **AWS KMS Key** — From the drop-down list, choose one of the following:

   - **(default) aws/rds** — The account number and AWS KMS key ID are listed following this option.
   - **<some-key-name>** — If you created a key, it is listed and available for you to choose.
   - **Enter a key ARN** — In the **ARN** box, enter the Amazon Resource Name (ARN) for your AWS KMS key. The format of the ARN is `arn:aws:kms:<region>:<accountID>:key/<key-id>` .

6. To make a copy of the selected snapshot, choose **Copy snapshot**. Alternatively, you can choose **Cancel** to not make a copy of the snapshot.

Using the AWS CLI

To make a copy of an unencrypted cluster snapshot using the AWS CLI, use the `copy-db-cluster-snapshot` operation with the following parameters. If you are copying the snapshot to another AWS Region, run the command in the AWS Region to which the snapshot will be copied.

- **`--source-db-cluster-snapshot-identifier`** — Required. The identifier of the cluster snapshot to make a copy of. The cluster snapshot must exist and be in the available state. If you are copying the snapshot to another AWS Region or if you are copying a shared cluster snapshot, this identifier must be in the ARN format of the source cluster snapshot. This parameter is not case sensitive.
- **`--target-db-cluster-snapshot-identifier`** — Required. The identifier of the new cluster snapshot to create from the source cluster snapshot. This parameter is not case sensitive.

Target snapshot naming constraints:

- Cannot be the name of an existing snapshot.
- Length is [1—63] letters, numbers, or hyphens.
- First character must be a letter.
- Cannot end with a hyphen or contain two consecutive hyphens.
- Must be unique for all clusters across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.

- **`--source-region`** — If you are copying the snapshot to another AWS Region, specify the AWS Region that the encrypted cluster snapshot will be copied from.

  If you're copying the snapshot to another AWS Region and you don't specify `--source-region`, you must specify the `pre-signed-url` option instead. The `pre-signed-url` value must be a URL that contains a Signature Version 4 signed request for the `CopyDBClusterSnapshot` action to be called in the source AWS Region where the cluster snapshot is copied from. To learn more about the `pre-signed-url`, see CopyDBClusterSnapshot.

- **`--kms-key-id`** — The KMS key identifier for the key to use to encrypt the copy of the cluster snapshot.

  If you are copying an encrypted cluster snapshot to another AWS Region, this parameter is required. You must specify a KMS key for the destination AWS Region.

  If you are copying an encrypted cluster snapshot in the same AWS Region, the AWS KMS key parameter is optional. The copy of the cluster snapshot is encrypted with the same AWS KMS key as the source cluster snapshot. If you want to specify a new AWS KMS encryption key to use to encrypt the copy, you can do so using this parameter.

- **`--copy-tags`** — Optional. The tags and values to be copied over.

To cancel a copy operation once it's in progress, you can delete the target cluster snapshot identified by `--target-db-cluster-snapshot-identifier` or `TargetDBClusterSnapshotIdentifier` while that cluster snapshot is in **copying** status.

Example

Example 1: Copy an unencrypted snapshot to the same Region

The following AWS CLI example creates a copy of `sample-cluster-snapshot` named `sample-cluster-snapshot-copy` in the same AWS Region as the source snapshot. When the copy is made, all tags on the original snapshot are copied to the snapshot copy.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-snapshot \
    --source-db-cluster-snapshot-identifier sample-cluster-snapshot \
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy \
    --copy-tags
```

For Windows:

```
aws docdb copy-db-cluster-snapshot ^
    --source-db-cluster-snapshot-identifier sample-cluster-snapshot ^
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy ^
    --copy-tags
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshot": {
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1b",
            "us-east-1c"
        ],
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot-copy",
        "DBClusterIdentifier": "sample-cluster",
        "SnapshotCreateTime": "2020-03-27T08:40:24.805Z",
        "Engine": "docdb",
        "Status": "copying",
        "Port": 0,
        "VpcId": "vpc-abcd0123",
        "ClusterCreateTime": "2020-01-10T22:13:38.261Z",
        "MasterUsername": "master-user",
        "EngineVersion": "4.0.0",
        "SnapshotType": "manual",
        "PercentProgress": 0,
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:111122223333:key/sample-key-id",
```

```
        "DBClusterSnapshotArn": "arn:aws:rds:us-east-1:111122223333:cluster-
snapshot:sample-cluster-snapshot-copy",
        "SourceDBClusterSnapshotArn": "arn:aws:rds:us-east-1:111122223333:cluster-
snapshot:sample-cluster-snapshot"
    }
}
```

Example

Example 2: Copy an unencrypted snapshot across AWS Regions

The following AWS CLI example creates a copy of `sample-cluster-snapshot`, which has the ARN `arn:aws:rds:us-east-1:123456789012:cluster-snapshot:sample-cluster-snapshot`. This copy is named named `sample-cluster-snapshot-copy` and is in the AWS Region in which the command is run.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-snapshot \
    --source-db-cluster-snapshot-identifier arn:aws:rds:us-
east-1:123456789012:cluster-snapshot:sample-cluster-snapshot \
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy
```

For Windows:

```
aws docdb copy-db-cluster-snapshot ^
    --source-db-cluster-snapshot-identifier arn:aws:rds:us-
east-1:123456789012:cluster-snapshot:sample-cluster-snapshot ^
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshot": {
        "AvailabilityZones": [
            "us-east-1a",
            "us-east-1b",
            "us-east-1c"
        ],
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot-copy",
        "DBClusterIdentifier": "sample-cluster",
```

```
        "SnapshotCreateTime": "2020-04-29T16:45:51.239Z",
        "Engine": "docdb",
        "AllocatedStorage": 0,
        "Status": "copying",
        "Port": 0,
        "VpcId": "vpc-abc0123",
        "ClusterCreateTime": "2020-04-28T16:43:00.294Z",
        "MasterUsername": "master-user",
        "EngineVersion": "4.0.0",
        "LicenseModel": "docdb",
        "SnapshotType": "manual",
        "PercentProgress": 0,
        "StorageEncrypted": false,
        "DBClusterSnapshotArn": "arn:aws:rds:us-east-1:123456789012:cluster-
snapshot:sample-cluster-snapshot-copy",
        "SourceDBClusterSnapshotArn": "arn:aws:rds:us-east-1:123456789012:cluster-
snapshot:sample-cluster-snapshot",
    }
}
```

Example

Example 3: Copy an encrypted snapshot across AWS Regions

The following AWS CLI example creates a copy of `sample-cluster-snapshot` from the us-west-2 Region to the us-east-1 Region. This command is called in the us-east-1 Region.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-snapshot \
    --source-db-cluster-snapshot-identifier arn:aws:rds:us-
west-2:123456789012:cluster-snapshot:sample-cluster-snapshot \
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy \
    --source-region us-west-2 \
    --kms-key-id sample-us-east-1-key
```

For Windows:

```
aws docdb copy-db-cluster-snapshot ^
    --source-db-cluster-snapshot-identifier arn:aws:rds:us-
west-2:123456789012:cluster-snapshot:sample-cluster-snapshot ^
    --target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy ^
    --source-region us-west-2 ^
```

```
    --kms-key-id sample-us-east-1-key
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshot": {
        "AvailabilityZones": [],
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot-copy",
        "DBClusterIdentifier": "sample-cluster",
        "SnapshotCreateTime": "2020-04-29T16:45:53.159Z",
        "Engine": "docdb",
        "AllocatedStorage": 0,
        "Status": "copying",
        "Port": 0,
        "ClusterCreateTime": "2020-04-28T16:43:07.129Z",
        "MasterUsername": "chimera",
        "EngineVersion": "4.0.0",
        "LicenseModel": "docdb",
        "SnapshotType": "manual",
        "PercentProgress": 0,
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/0846496a-
d48e-41c4-9353-86d8301d7e35",
        "DBClusterSnapshotArn": "arn:aws:rds:us-east-1:123456789012:cluster-
snapshot:sample-cluster-snapshot-copy",
        "SourceDBClusterSnapshotArn": "arn:aws:rds:us-west-2:123456789012:cluster-
snapshot:sample-cluster-snapshot",
    }
}
```

Example

Example 4: Copy an unencrypted shared snapshot across AWS Regions

The following AWS CLI example, account - 123456789012, creates a copy of an unencrypted cluster snapshot `sample-cluster-snapshot` shared by account - 999999999999 from the us-east-1 Region to the us-west-2 Region. This command is called in the us-west-2 Region. For more information on sharing snapshots, see Sharing a snapshot.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-snapshot \
```

```
--region us-west-2 \
--source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:999999999999:cluster-
snapshot:sample-cluster-snapshot \
--target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy
```

For Windows:

```
aws docdb copy-db-cluster-snapshot ^
--region us-west-2 ^
--source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:999999999999:cluster-
snapshot:sample-cluster-snapshot ^
--target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy
```

Output from this operation looks something like the following.

```
{
 "DBClusterSnapshots": [
 {
 "AvailabilityZones": [],
 "DBClusterSnapshotIdentifier": "sample-cluster-snapshot-copy",
 "DBClusterIdentifier": "sample-cluster",
 "SnapshotCreateTime": "2025-08-22T11:27:00.497000+00:00",
 "Engine": "docdb",
 "Status": "copying",
 "Port": 0,
 "ClusterCreateTime": "2024-07-02T16:44:50.246000+00:00",
 "MasterUsername": "master-user",
 "EngineVersion": "5.0.0",
 "SnapshotType": "manual",
 "PercentProgress": 0,
 "StorageEncrypted": false,
 "DBClusterSnapshotArn": "arn:aws:rds:us-west-2:123456789012:cluster-
snapshot:sample-cluster-snapshot-copy",
 "SourceDBClusterSnapshotArn": "arn:aws:rds:us-east-1:999999999999:cluster-
snapshot:sample-cluster-snapshot"
 }
 ]
}
```

Example

Example 5: Copy an encrypted shared snapshot across AWS Regions

The following AWS CLI example, account - 123456789012 creates a copy of an encrypted cluster snapshot `sample-cluster-snapshot` shared by account - 999999999999 from the us-east-1 Region to the us-west-2 Region. The target snapshot is encrypted with a customer managed KMS key - `arn:aws:kms:us-west-2:123456789012:key/6c1f3264-1797-472b-ba37-03011e682d28`. This command is called in the us-west-2 Region. For more information on sharing snapshots, see Sharing a snapshot.

For Linux, macOS, or Unix:

```
aws docdb copy-db-cluster-snapshot \
--region us-west-2 \
--source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:999999999999:cluster-snapshot:sample-cluster-snapshot \
--target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy \
--kms-key-id arn:aws:kms:us-west-2:123456789012:key/6c1f3264-1797-472b-ba37-03011e682d28
```

For Windows:

```
aws docdb copy-db-cluster-snapshot ^
--region us-west-2 ^
--source-db-cluster-snapshot-identifier arn:aws:rds:us-east-1:999999999999:cluster-snapshot:sample-cluster-snapshot ^
--target-db-cluster-snapshot-identifier sample-cluster-snapshot-copy ^
--kms-key-id arn:aws:kms:us-west-2:123456789012:key/6c1f3264-1797-472b-ba37-03011e682d28
```

Output from this operation looks something like the following.

```
{
 "DBClusterSnapshots": [
 {
 "AvailabilityZones": [],
 "DBClusterSnapshotIdentifier": "sample-cluster-snapshot-copy",
 "DBClusterIdentifier": "sample-cluster",
 "SnapshotCreateTime": "2025-08-22T11:27:00.497000+00:00",
 "Engine": "docdb",
 "Status": "copying",
 "Port": 0,
 "ClusterCreateTime": "2024-07-02T16:44:50.246000+00:00",
```

```
 "MasterUsername": "master-user",
 "EngineVersion": "5.0.0",
 "SnapshotType": "manual",
 "PercentProgress": 0,
 "StorageEncrypted": true,
 "KmsKeyId": "arn:aws:kms:us-west-2:123456789012:key/6c1f3264-1797-472b-
ba37-03011e682d28",
 "DBClusterSnapshotArn": "arn:aws:rds:us-west-2:123456789012:cluster-
snapshot:sample-cluster-snapshot-copy",
 "SourceDBClusterSnapshotArn": "arn:aws:rds:us-east-1:999999999999:cluster-
snapshot:sample-cluster-snapshot"
 }
 ]
}
```

Note

For more information about encrypting snapshot copies, see Copy cluster snapshot encryption.
For more information about encryption at rest, see Encrypting Amazon DocumentDB data at rest.

Sharing Amazon DocumentDB cluster snapshots

Using Amazon DocumentDB, you can share a manual cluster snapshot in the following ways:

- Sharing a manual cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to copy the snapshot.
- Sharing a manual cluster snapshot, whether encrypted or unencrypted, enables authorized AWS accounts to directly restore a cluster from the snapshot instead of taking a copy of it and restoring from that.

Note

To share an automated cluster snapshot, create a manual cluster snapshot by copying the automated snapshot, and then share that copy. This process also applies to AWS Backup–generated resources.

You can share a manual snapshot with up to 20 other AWS accounts. You can also share an unencrypted manual snapshot as public, which makes the snapshot available to all accounts. When sharing a snapshot as public, ensure that none of your private information is included in any of your public snapshots.

When sharing manual snapshots with other AWS accounts, and you restore a cluster from a shared snapshot using the AWS CLI or the Amazon DocumentDB API, you must specify the Amazon Resource Name (ARN) of the shared snapshot as the snapshot identifier.

Sharing an encrypted snapshot

The following restrictions apply to sharing encrypted snapshots:

- You can't share encrypted snapshots as public.
- You can't share a snapshot that has been encrypted using the default AWS KMS encryption key of the account that shared the snapshot.

Follow these steps to share encrypted snapshots.

1. Share the AWS Key Management Service (AWS KMS) encryption key that was used to encrypt the snapshot with any accounts that you want to be able to access the snapshot.

   You can share AWS KMS encryption keys with another AWS accounts by adding the other accounts to the AWS KMS key policy. For details on updating a key policy, see Using Key Policies in AWS KMS in the *AWS Key Management Service Developer Guide*. For an example of creating a key policy, see Creating an IAM policy to enable copying of the encrypted snapshot later in this topic.
2. Use the AWS CLI, as shown below, to share the encrypted snapshot with the other accounts.

Allowing access to an AWS KMS encryption key

For another AWS account to copy an encrypted snapshot shared from your account, the account that you share your snapshot with must have access to the AWS KMS key that encrypted the snapshot. To allow another account access to an AWS KMS key, update the key policy for the AWS KMS key with the ARN of the account that you are sharing to as a principal in the AWS KMS key policy. Then allow the `kms:CreateGrant` action.

After you give an account access to your AWS KMS encryption key, to copy your encrypted snapshot, that account must create an AWS Identity and Access Management (IAM) user if it doesn't already have one. In addition, that account must also attach an IAM policy to that IAM user that allows the user to copy an encrypted snapshot using your AWS KMS key. The account must be an IAM user and cannot be a root AWS account identity due to AWS KMS security restrictions.

In the following key policy example, user 123451234512 is the owner of the AWS KMS encryption key. User 123456789012 is the account that the key is being shared with. This updated key policy gives the account access to the AWS KMS key. It does this by including the ARN for the root AWS account identity for user 123456789012 as a principal for the policy, and by allowing the `kms:CreateGrant` action.

JSON

```
{
    "Id": "key-policy-1",
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Allow use of the key",
            "Effect": "Allow",
            "Principal": {"AWS": [
                "arn:aws:iam::123451234512:user/KeyUser",
                "arn:aws:iam::123456789012:root"
            ]},
            "Action": [
                "kms:CreateGrant",
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ReEncrypt*",
                "kms:GenerateDataKey*",
                "kms:DescribeKey"
            ],
            "Resource": "*"},
            {
            "Sid": "Allow attachment of persistent resources",
            "Effect": "Allow",
            "Principal": {"AWS": [
                "arn:aws:iam::123451234512:user/KeyUser",
                "arn:aws:iam::123456789012:root"
            ]},
```

```
            "Action": [
                "kms:CreateGrant",
                "kms:ListGrants",
                "kms:RevokeGrant"
            ],
            "Resource": "*",
            "Condition": {"Bool": {"kms:GrantIsForAWSResource": true}}
        }
    ]
}
```

Creating an IAM policy to enable copying of the encrypted snapshot

When the external AWS account has access to your AWS KMS key, the owner of that account can create a policy to allow an IAM user that is created for the account to copy an encrypted snapshot that is encrypted with that AWS KMS key.

The following example shows a policy that can be attached to an IAM user for AWS account 123456789012. The policy enables the IAM user to copy a shared snapshot from account 123451234512 that has been encrypted with the AWS KMS key `c989c1dd-a3f2-4a5d-8d96-e793d082ab26` in the us-west-2 Region.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowUseOfTheKey",
            "Effect": "Allow",
            "Action": [
                "kms:Encrypt",
                "kms:Decrypt",
                "kms:ReEncrypt*",
                "kms:GenerateDataKey*",
                "kms:DescribeKey",
                "kms:CreateGrant",
                "kms:RetireGrant"
            ],
            "Resource": ["arn:aws:kms:us-west-2:123451234512:key/c989c1dd-
a3f2-4a5d-8d96-e793d082ab26"]
```

```
        },
        {
            "Sid": "AllowAttachmentOfPersistentResources",
            "Effect": "Allow",
            "Action": [
                "kms:CreateGrant",
                "kms:ListGrants",
                "kms:RevokeGrant"
            ],
            "Resource": ["arn:aws:kms:us-west-2:123451234512:key/c989c1dd-
a3f2-4a5d-8d96-e793d082ab26"],
            "Condition": {
                "Bool": {
                    "kms:GrantIsForAWSResource": true
                }
            }
        }
    ]
}
```

For details on updating a key policy, see Using Key Policies in AWS KMS in the *AWS Key Management Service Developer Guide*.

Sharing a snapshot

You can share an Amazon DocumentDB manual cluster snapshot (or a copy of an automated snapshot) using the AWS Management Console or the AWS CLI:

Using the AWS Management Console

To share a snapshot using the AWS Management Console, complete the following steps:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Snapshots**.
3. Select the manual snapshot that you want to share.
4. In the **Actions** drop-down menu, choose Share.
5. Choose one of the following options for **DB snapshot visibility**:

- If the source is unencrypted, choose **Public** to permit all AWS accounts to restore a cluster from your manual snapshot. Or choose **Private** to permit only AWS accounts that you specify to restore a cluster from your manual snapshot.

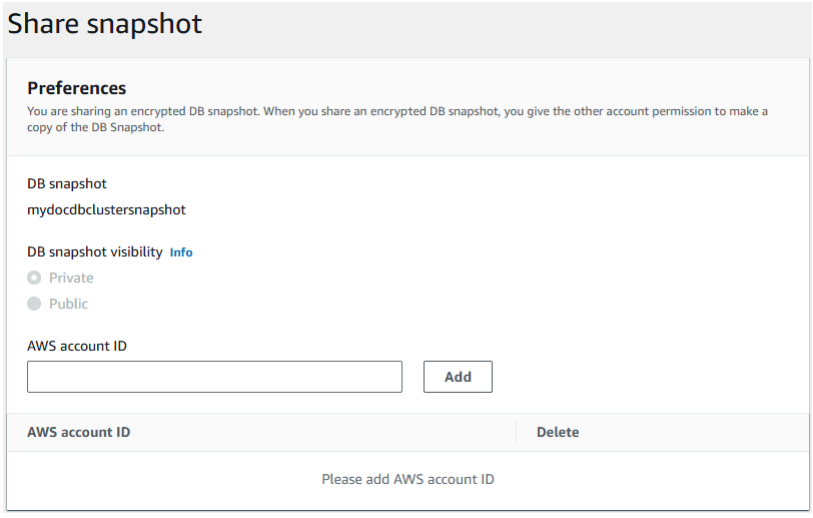
  > **Warning**
  >
  > If you set **DB snapshot visibility** to **Public**, all AWS accounts can restore a cluster from your manual snapshot and have access to your data. Do not share any manual cluster snapshots that contain private information as **Public**.

- If the source is encrypted, **DB snapshot visibility** is set as **Private** because encrypted snapshots can't be shared as public.

  > **Note**
  >
  > Snapshots that have been encrypted with the default AWS KMS key can't be shared.

6. For **AWS Account ID**, enter the AWS account identifier for an account that you want to permit to restore a cluster from your manual snapshot, and then choose **Add**. Repeat to include additional AWS account identifiers, up to 20 AWS accounts.

   If you make an error when adding an AWS account identifier to the list of permitted accounts, you can delete it from the list by choosing **Delete** at the right of the incorrect AWS account identifier.



7. After you have added identifiers for all of the AWS accounts that you want to permit to restore the manual snapshot, choose **Save** to save your changes.

Using the AWS CLI

To share a snapshot using the AWS CLI, use the Amazon DocumentDB `modify-db-snapshot-attribute` operation. Use the `--values-to-add` parameter to add a list of the IDs for the AWS accounts that are authorized to restore the manual snapshot.

The following example permits two AWS account identifiers, 123451234512 and 123456789012, to restore the snapshot named `manual-snapshot1`. It also removes the `all` attribute value to mark the snapshot as private.

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster-snapshot-attribute \
    --db-cluster-snapshot-identifier sample-cluster-snapshot \
    --attribute-name restore \
    --values-to-add '["123451234512","123456789012"]'
```

For Windows:

```
aws docdb modify-db-cluster-snapshot-attribute ^
    --db-cluster-snapshot-identifier sample-cluster-snapshot ^
    --attribute-name restore ^
    --values-to-add '["123451234512","123456789012"]'
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshotAttributesResult": {
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot",
        "DBClusterSnapshotAttributes": [
            {
                "AttributeName": "restore",
                "AttributeValues": [
                    "123451234512",
                    "123456789012"
                ]
            }
        ]
```

```
    }
}
```

To remove an AWS account identifier from the list, use the `--values-to-remove` parameter. The following example prevents AWS account ID 123456789012 from restoring the snapshot.

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster-snapshot-attribute \
    --db-cluster-snapshot-identifier sample-cluster-snapshot \
    --attribute-name restore \
    --values-to-remove '["123456789012"]'
```

For Windows:

```
aws docdb modify-db-cluster-snapshot-attribute ^
    --db-cluster-snapshot-identifier sample-cluster-snapshot ^
    --attribute-name restore ^
    --values-to-remove '["123456789012"]'
```

Output from this operation looks something like the following.

```
{
    "DBClusterSnapshotAttributesResult": {
        "DBClusterSnapshotIdentifier": "sample-cluster-snapshot",
        "DBClusterSnapshotAttributes": [
            {
                "AttributeName": "restore",
                "AttributeValues": [
                    "123451234512"
                ]
            }
        ]
    }
}
```

Restoring from a cluster snapshot

Amazon DocumentDB (with MongoDB compatibility) creates a cluster snapshot of your storage volume. You can create a new cluster by restoring from a cluster snapshot. When you restore the

cluster, you provide the name of the cluster snapshot to restore from and a name for the new cluster that is created by the restore. You can't restore from a snapshot to an existing cluster because a new cluster is created when you restore.

When you are restoring a cluster from a cluster snapshot:

- This action restores only the cluster, and not the instances for that cluster. You must invoke the `create-db-instance` action to create instances for the restored cluster, specifying the identifier of the restored cluster in `--db-cluster-identifier`. You can create instances only after the cluster is *available*.
- You cannot restore an encrypted snapshot to an unencrypted cluster. However, you can restore an unencrypted snapshot to an encrypted cluster by specifying the AWS KMS key.
- To restore a cluster from an encrypted snapshot, you must have access to the AWS KMS key.

> **Note**
>
> You cannot restore a 3.6 cluster to a 4.0 cluster but you can migrate from one cluster version to another. For more information, go to Migrating and upgrading Amazon DocumentDB.

Using the AWS Management Console

The following procedure shows how to restore an Amazon DocumentDB cluster from a cluster snapshot using the Amazon DocumentDB Management Console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Snapshots**, and then choose the button to the left of the snapshot that you want to use to restore a cluster.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. On the **Actions** menu, choose **Restore**.

4. On the **Restore snapshot** page, complete the **Configuration** section.

   a. **Cluster identifier** — The name for the new cluster. You can accept the Amazon DocumentDB supplied name or type a name that you prefer. The Amazon DocumentDBsupplied name is in the format of `docdb-` plus a UTC timestamp; for example, `docdb-yyyy-mm-dd-hh-mm-ss`.

   b. **Instance class** — The instance class for the new cluster. You can accept the default instance class or choose an instance class from the drop-down list.

   c. **Number of instances** — The number of instances you want created with this cluster. You can accept the default of 3 instances (1 primary read/write and 2 read-only replicas) or choose the number of instances from the drop-down list.

5. For **Cluster storage configuration**, choose a storage option.

   > **Note**
   >
   > **Amazon DocumentDB I/O-Optimized** storage configuration is only available on the Amazon DocumentDB 5.0 and 8.0 engine versions.

6. If you are satisfied with the cluster configuration, choose **Restore cluster** and wait while your cluster is restored.

7. If you prefer to change some configurations, such as specifying a non-default Amazon VPC or security group, choose **Show advanced settings** from the bottom left of the page, and then continue with the following steps.

   a. Complete the **Network settings** section.

      - **Virtual Private Cloud (VPC)** — Accept the current VPC, or choose a VPC from the drop-down list.
      - **Subnet Group** — Accept the `default` subnet group, or choose one from the drop-down list.
      - **VPC Security Groups** — Accept the `default (VPC)` security group, or choose one from the list.

   b. Complete the **Cluster options** section.

      - **Database port** — Accept the default port, `27017`, or use the up or down arrow to set the port that you want to use for application connections.

   c. Complete the **Encryption** section.

- **Encryption at rest** — If your snapshot is encrypted, these options are not available to you. If it is not encrypted, you can choose one of the following:
  - To encrypt all your cluster's data, choose **Enable encryption-at-rest**. If you choose this option, you must designate a KMS key.
  - To not encrypt your cluster's data, choose **Disable encryption-at-rest**. If you choose this option, you are finished with the encryption section.
- **AWS KMS Key** — Choose one of the following from the drop-down list:
  - **(default) aws/rds** — The account number and AWS KMS key ID are listed following this option.
  - **Customer-managed key** — This option is available only if you created an IAM encryption key in the AWS Identity and Access Management (IAM) console. You can choose the key to encrypt your cluster.
  - **Enter a key ARN** — In the **ARN** box, enter the Amazon Resource Name (ARN) for your AWS KMS key. The format of the ARN is `arn:aws:kms:<region>:<accountID>:key/<key-id>`.

d. Complete the **Log exports** section.

- **Select the log types to publish to CloudWatch** — Choose one of the following:
  - **Enabled** — Enables your cluster to export DDL logging to Amazon CloudWatch Logs.
  - **Disabled** — Prevents your cluster from exporting DDL logs to Amazon CloudWatch Logs. **Disabled** is the default.
- **IAM role**—From the list, choose *RDS Service Linked Role*.

e. Complete the **Tags** section.

- **Add Tag** — In the *Key* box, enter the name for the tag for your cluster. In the *Value* box, optionally enter the tag value. Tags are used with AWS Identity and Access Management (IAM) policies to manage access to Amazon DocumentDB resources and to control what actions can be applied to the resources.

f. Complete the **Deletion protection** section.

- **Enable deletion protection** — Protects the cluster from being accidentally deleted. While this option is enabled, you can't delete the cluster.

8. Choose **Restore cluster**.

Using the AWS CLI

To restore a cluster from a snapshot using the AWS CLI, use the `restore-db-cluster-from-snapshot` operation with the following parameters. For more information, see RestoreDBClusterFromSnapshot.

- **`--db-cluster-identifier`** — Required. The name of the cluster that is created by the operation. A cluster by this name cannot exist before this operation.

  Cluster naming constraints:

  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all clusters across Amazon RDS, Neptune, and Amazon DocumentDB per AWS account, per Region.
- **`--snapshot-identifier`** — Required. The name of the snapshot used to restore from. A snapshot by this name must exist and be in the *available* state.
- **`--engine`** — Required. Must be `docdb`.
- **`--storage-type standard | iopt1`** — Optional. Default: `standard`.
- **`--kms-key-id`** — Optional. The ARN of the AWS KMS key identifier to use when restoring an encrypted snapshot or encrypting a cluster when restoring from an unencrypted snapshot. Supplying the AWS KMS key ID results in the restored cluster being encrypted with the AWS KMS key, whether or not the snapshot was encrypted.

  The format of the `--kms-key-id` is `arn:aws:kms:<region>:<accountID>:key/<key-id>`. If you do not specify a value for the `--kms-key-id` parameter, then the following occurs:

  - If the snapshot in `--snapshot-identifier` is encrypted, then the restored cluster is encrypted using the same AWS KMS key that was used to encrypt the snapshot.
  - If the snapshot in `--snapshot-identifier` is not encrypted, then the restored cluster is not encrypted.

For Linux, macOS, or Unix:

```
aws docdb restore-db-cluster-from-snapshot \
    --db-cluster-identifier sample-cluster-restore \
```

```
    --snapshot-identifier sample-cluster-snapshot \
    --engine docdb \
    --kms-key-id arn:aws:kms:us-east-1:123456789012:key/SAMPLE-KMS-KEY-ID
```

For Windows:

```
aws docdb restore-db-cluster-from-snapshot ^
    --db-cluster-identifier sample-cluster-restore ^
    --snapshot-identifier sample-cluster-snapshot ^
    --engine docdb ^
    --kms-key-id arn:aws:kms:us-east-1:123456789012:key/SAMPLE-KMS-KEY-ID
```

Output from this operation looks something like the following.

```
{
    "DBCluster": {
        "AvailabilityZones": [
            "us-east-1c",
            "us-east-1b",
            "us-east-1a"
        ],
        "BackupRetentionPeriod": 1,
        "DBClusterIdentifier": "sample-cluster-restore",
        "DBClusterParameterGroup": "default.docdb4.0",
        "DBSubnetGroup": "default",
        "Status": "creating",
        "Endpoint": "sample-cluster-restore.cluster-node.us-
east-1.docdb.amazonaws.com",
        "ReaderEndpoint": "sample-cluster-restore.cluster-node.us-
east-1.docdb.amazonaws.com",
        "MultiAZ": false,
        "Engine": "docdb",
        "EngineVersion": "4.0.0",
        "Port": 27017,
        "MasterUsername": "<master-user>",
        "PreferredBackupWindow": "02:00-02:30",
        "PreferredMaintenanceWindow": "tue:09:50-tue:10:20",
        "DBClusterMembers": [],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcdefgh",
                "Status": "active"
            }
```

```
        ],
        "HostedZoneId": "ABCDEFGHIJKLM",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/<sample-key-id>",
        "DbClusterResourceId": "cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "DBClusterArn": "arn:aws:rds:us-east-1:<accountID>:cluster:sample-cluster-
restore",
        "AssociatedRoles": [],
        "ClusterCreateTime": "2020-04-01T01:43:40.871Z",
        "DeletionProtection": true
    }
}
```

After the cluster status is *available*, create at least one instance for the cluster.

For Linux, macOS, or Unix:

```
aws docdb create-db-instance \
    --db-cluster-identifier sample-cluster-restore  \
    --db-instance-identifier sample-cluster-restore-instance \
    --availability-zone us-east-1b \
    --promotion-tier 2 \
    --db-instance-class db.r5.large \
    --engine docdb
```

For Windows:

```
aws docdb create-db-instance ^
    --db-cluster-identifier sample-cluster-restore  ^
    --db-instance-identifier sample-cluster-restore-instance ^
    --availability-zone us-east-1b ^
    --promotion-tier 2 ^
    --db-instance-class db.r5.large ^
    --engine docdb
```

Output from this operation looks something like the following.

```
{
    "DBInstance": {
        "DBInstanceIdentifier": "sample-cluster-restore-instance",
        "DBInstanceClass": "db.r5.large",
        "Engine": "docdb",
        "DBInstanceStatus": "creating",
```

```
        "PreferredBackupWindow": "02:00-02:30",
        "BackupRetentionPeriod": 1,
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcdefgh",
                "Status": "active"
            }
        ],
        "AvailabilityZone": "us-west-2b",
        "DBSubnetGroup": {
            "DBSubnetGroupName": "default",
            "DBSubnetGroupDescription": "default",
            "VpcId": "vpc-6242c31a",
            "SubnetGroupStatus": "Complete",
            "Subnets": [
                {
                    "SubnetIdentifier": "subnet-abcdefgh",
                    "SubnetAvailabilityZone": {
                        "Name": "us-west-2a"
                    },
                    "SubnetStatus": "Active"
                },
                {
                    ...
                }
            ]
        },
        "PreferredMaintenanceWindow": "fri:09:43-fri:10:13",
        "PendingModifiedValues": {},
        "EngineVersion": "4.0.0",
        "AutoMinorVersionUpgrade": true,
        "PubliclyAccessible": false,
        "DBClusterIdentifier": "sample-cluster-restore",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/<sample-key-id>",
        "DbiResourceId": "db-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "CACertificateIdentifier": "rds-ca-2019",
        "PromotionTier": 2,
        "DBInstanceArn": "arn:aws:rds:us-east-1:<accountID>:db:sample-cluster-
restore-instance"
    }
}
```

Restoring to a point in time

You can restore a cluster to any point in time that is within the cluster's backup retention period using the AWS Management Console or AWS Command Line Interface (AWS CLI).

> **Note**
>
> You cannot conduct a point-in-time restore of a 3.6 cluster to a 4.0 cluster but you can migrate from one cluster version to another. For more information, go to Migrating and upgrading Amazon DocumentDB.

Keep the following in mind when restoring a cluster to a point in time.

- The new cluster is created with the same configuration as the source cluster, except that the new cluster is created with the default parameter group. To set the new cluster's parameter group to the source cluster's parameter group, modify the cluster after it is *available*. For more information on modifying a cluster, see Modifying an Amazon DocumentDB cluster.

Using the AWS Management Console

You can restore a cluster to a point-in-time within its backup retention period by completing the following using the AWS Management Console.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**. In the list of clusters, choose the button to the left of the cluster that you want to restore.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. On the **Actions** menu, choose **Restore to point in time**.
4. Complete the **Restore time** section, which specifies the date and time to restore to.

a. **Restore date**—Choose or enter a date that is between the **Earliest restore time** and the **Latest restore time**.

b. **Restore time**—Choose or enter the hour, minute, and seconds that are between the **Earliest restore time** and the **Latest restore time**.

5. Complete the **Configuration** section.

   a. **Cluster identifier** — Accept the default identifier, or enter an identifier that you prefer.

      Cluster naming constraints:

      - Length is [1—63] letters, numbers, or hyphens.
      - First character must be a letter.
      - Cannot end with a hyphen or contain two consecutive hyphens.
      - Must be unique for all clusters across Amazon RDS, Neptune and Amazon DocumentDB per AWS account, per Region.

   b. **Instance class** — From the drop-down list, choose the instance class that you want for the cluster's instances.

   c. **Number of instances** — From the drop-down list, choose the number of instances that you want created when the cluster is restored.

6. For **Cluster storage configuration**, choose a storage option.

   > **Note**
   >
   > **Amazon DocumentDB I/O-Optimized** storage configuration is only available on the Amazon DocumentDB 5.0 and 8.0 engine versions.

7. Optional. To configure the network settings, cluster options, and enable log exports, choose **Show advanced settings**, and then complete the following sections. Otherwise, continue with the next step.

   - **Network settings**

     1. **Virtual Private Cloud (VPC)** — From the drop-down list, choose the VPC that you want to use for this cluster.
     2. **Subnet group** — From the drop-down list, choose the subnet group for this cluster.

3. **VPC security groups** — From the drop-down list, choose the VPC security groups for this cluster.

- **Cluster options**

  1. **Port** — Accept the default port (27017), or use the up and down arrows to set the port for communicating with this cluster.

- **Log exports**

  1. **Audit logs** — Select this option to enable exporting audit logs to Amazon CloudWatch Logs. If you select this option, you must enable `audit_logs` in the cluster's custom parameter group. For more information, see Auditing Amazon DocumentDB events.
  2. **Profiler logs** — Select this option to enable exporting operation profiler logs to Amazon CloudWatch Logs. If you select this option, you must also modify the following parameters in the cluster's custom parameter group:
     - `profiler` — Set to `enabled`.
     - `profiler_threshold_ms` — Set to a value `[0-INT_MAX]` to set the threshold for profiling operations.
     - `profiler_sampling_rate` — Set to a value `[0.0-1.0]` to set the percentage of slow operations to profile.

     For more information, see Profiling Amazon DocumentDB operations.
  3. **Profiler logs** — Export profiler logs to Amazon CloudWatch
  4. **IAM role** — From the drop-down list, choose *RDS Service Linked Role*.

- **Tags**

  1. **Add Tag** — In the *Key* box, enter the name for the tag for your cluster. In the *Value* box, optionally enter the tag value. Tags are used with AWS Identity and Access Management (IAM) policies to manage access to Amazon DocumentDB resources and to control what actions can be applied to the resources.

- **Deletion protection**

  1. **Enable deletion protection** — Protects the cluster from being accidentally deleted. While this option is enabled, you can't delete the cluster.

8. To restore the cluster, choose **Create cluster**. Alternatively, you can choose **Cancel** to cancel the operation.

Using the AWS CLI

To restore a cluster to a point in time using the snapshot's backup retention period, use the `restore-db-cluster-to-point-in-time` operation with the following parameters.

- **`--db-cluster-identifier`**— Required. The name of the new cluster to be created. This cluster cannot exist before the operation. The parameter value must meet the following constraints.

  Cluster naming constraints:
  - Length is [1—63] letters, numbers, or hyphens.
  - First character must be a letter.
  - Cannot end with a hyphen or contain two consecutive hyphens.
  - Must be unique for all clusters across Amazon RDS, Neptune and Amazon DocumentDB per AWS account, per Region.
- **`--restore-to-time`** — The UTC date and time to restore the cluster to. For example, `2018-06-07T23:45:00Z`.

  Time Constraints:
  - Must be before the latest restorable time for the cluster.
  - Must be specified if the `--use-latest-restorable-time` parameter is not provided.
  - Cannot be specified if the `--use-latest-restorable-time` parameter is `true`.
  - Cannot be specified if the `--restore-type` parameter value is `copy-on-write`.
- **`--source-db-cluster-identifier`** — The name of the source cluster from which to restore. This cluster must exist and be available.
- **`--use-latest-restorable-time`** or **`--no-use-latest-restorable-time`** — Whether to restore to the latest restorable backup time. Cannot be specified if the `--restore-to-time` parameter is provided.
- **`--storage-type standard | iopt1`** — Optional. Default: `standard`.

The AWS CLI operation `restore-db-cluster-to-point-in-time` only restores the cluster, not the instances for that cluster. You must invoke the `create-db-instance` operation to

create instances for the restored cluster, specifying the identifier of the restored cluster in `--db-cluster-identifier`. You can create instances only after the `restore-db-cluster-to-point-in-time` operation has completed and the restored cluster is *available*.

Example

The following example creates `sample-cluster-restored` from the snapshot `sample-cluster-snapshot` to the latest restorable time.

For Linux, macOS, or Unix:

```
aws docdb restore-db-cluster-to-point-in-time \
    --db-cluster-identifier sample-cluster-restored \
    --source-db-cluster-identifier sample-cluster-snapshot \
    --use-latest-restorable-time
```

For Windows:

```
aws docdb restore-db-cluster-to-point-in-time ^
    --db-cluster-identifier sample-cluster-restored ^
    --source-db-cluster-identifier sample-cluster-snapshot ^
    --use-latest-restorable-time
```

Example

The following example creates `sample-cluster-restored` from the snapshot `sample-cluster-snapshot` to 03:15 on December 11, 2018 (UTC), which is within the backup retention period of `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb restore-db-cluster-to-point-in-time \
    --db-cluster-identifier sample-cluster-restore \
    --source-db-cluster-identifier sample-cluster \
    --restore-to-time 2020-05-12T03:15:00Z
```

For Windows:

```
aws docdb restore-db-cluster-to-point-in-time ^
    --db-cluster-identifier sample-cluster-restore ^
    --source-db-cluster-identifier sample-cluster ^
```

```
    --restore-to-time 2020-05-12T03:15:00Z
```

Output from this operation looks something like the following.

```
{
    "DBCluster": {
        "AvailabilityZones": [
            "us-east-1c",
            "us-west-2b",
            "us-west-2a"
        ],
        "BackupRetentionPeriod": 1,
        "DBClusterIdentifier": "sample-cluster-restored",
        "DBClusterParameterGroup": "sample-parameter-group",
        "DBSubnetGroup": "default",
        "Status": "creating",
        "Endpoint": "sample-cluster-restored.node.us-east-1.docdb.amazonaws.com",
        "ReaderEndpoint": "sample-cluster-restored.node.us-
east-1.docdb.amazonaws.com",
        "MultiAZ": false,
        "Engine": "docdb",
        "EngineVersion": "4.0.0",
        "Port": 27017,
        "MasterUsername": "master-user",
        "PreferredBackupWindow": "02:00-02:30",
        "PreferredMaintenanceWindow": "tue:09:50-tue:10:20",
        "DBClusterMembers": [],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abc0123",
                "Status": "active"
            }
        ],
        "HostedZoneId": "ABCDEFGHIJKLM",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID^>:key/sample-key",
        "DbClusterResourceId": "cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "DBClusterArn": "arn:aws:rds:us-east-1:<accountID>:cluster:sample-cluster-
restored",
        "AssociatedRoles": [],
        "ClusterCreateTime": "2020-04-24T20:14:36.713Z",
        "DeletionProtection": false
    }
```

```
}
```

Deleting a cluster snapshot

A manual snapshot is a full backup that is deleted only when you manually delete it using the AWS Management Console or AWS CLI. You cannot manually delete an automatic snapshot because automatic snapshots are deleted only when the snapshot's retention period expires or you delete the snapshot's cluster.

Using the AWS Management Console

To delete a manual cluster snapshot using the AWS Management Console, complete the following steps.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Snapshots**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the list of snapshots, choose the button to the left of the snapshot that you want to delete. The snapshot's type must be **manual**.

   1. You can verify that that the snapshot's type is **manual** by checking if it is listed as `manual` or `automatic` under the **Type** column.

4. From the **Actions** menu, choose **Delete**. If the **Delete** option is unavailable, you probably chose an automatic snapshot.
5. On the delete confirmation screen, to delete the snapshot, choose **Delete**. To keep the snapshot, choose **Cancel**.

Using the AWS CLI

An Amazon DocumentDB manual cluster snapshot is a full backup that you can manually delete using the AWS CLI. You cannot manually delete an automatic snapshot.

To delete a manual cluster snapshot using the AWS CLI, use the `delete-db-cluster-snapshot` operation with the following parameters.

Parameters

- **`--db-cluster-snapshot-identifier`** — Required. The name of the manual snapshot to delete.

The following example deletes the cluster snapshot `sample-cluster-snapshot`.

For Linux, macOS, or Unix:

```
aws docdb delete-db-cluster-snapshot \
    --db-cluster-snapshot-identifier sample-cluster-snapshot
```

For Windows:

```
aws docdb delete-db-cluster-snapshot ^
    --db-cluster-snapshot-identifier sample-cluster-snapshot
```

Output from this operation lists the details of the cluster snapshot you deleted.

Monitoring and logging in Amazon DocumentDB

Monitoring your AWS services is an important part of keeping your systems healthy and functioning optimally. Collect monitoring data from all parts of your AWS solution so that you can more easily debug and fix failures or performance issues. Before you begin monitoring your AWS solutions, consider the following questions:

- What are your monitoring goals?
- What resources are you going to monitor?
- How frequently will you monitor these resources?
- What monitoring tools will you use?
- Who is responsible for doing the monitoring?
- Who is to be notified and by what means if something goes wrong?

To understand your current performance patterns, identify performance anomalies, and formulate methods to address issues, you should establish baseline performance metrics for various times and under differing load conditions. As you monitor your AWS solution, store your historical monitoring data for future reference and for establishing your baselines.

In general, acceptable values for performance metrics depend on what your baseline looks like and what your application is doing. Investigate consistent or trending variances from your baseline. The following is advice about specific types of metrics:

- **High CPU or RAM use** — High values for CPU or RAM use might be appropriate, provided that they are in keeping with your goals for your application (like throughput or concurrency) and are expected.
- **Storage volume consumption** —Investigate storage consumption (`VolumeBytesUsed`) if space that is used is consistently at or above 85 percent of the total storage volume space. Determine whether you can delete data from the storage volume or archive data to a different system to free up space. For more information, see Amazon DocumentDB storage and Amazon DocumentDB quotas.
- **Network traffic** — For network traffic, talk with your system administrator to understand what the expected throughput is for your domain network and internet connection. Investigate network traffic if throughput is consistently lower than expected.

- **Database connections** — Consider constraining database connections if you see high numbers of user connections in conjunction with decreases in instance performance and response time. The best number of user connections for your instance will vary based on your instance class and the complexity of the operations being performed.
- **IOPS metrics**—The expected values for IOPS metrics depend on disk specification and server configuration, so use your baseline to know what is typical. Investigate if values are consistently different from your baseline. For best IOPS performance, make sure that your typical working set fits into memory to minimize read and write operations.

Amazon DocumentDB (with MongoDB compatibility) provides a variety of Amazon CloudWatch metrics that you can monitor to determine the health and performance of your Amazon DocumentDB clusters and instances. You can view Amazon DocumentDB metrics using various tools, including the Amazon DocumentDB console, AWS CLI, CloudWatch API, and Performance Insights.

Topics

- Monitoring an Amazon DocumentDB cluster's status
- Monitoring an Amazon DocumentDB instance's status
- Viewing Amazon DocumentDB recommendations
- Using Amazon DocumentDB event subscriptions
- Monitoring Amazon DocumentDB with CloudWatch
- Logging Amazon DocumentDB API calls with AWS CloudTrail
- Profiling Amazon DocumentDB operations
- Monitoring with Performance Insights

Monitoring an Amazon DocumentDB cluster's status

The status of a cluster indicates the health of the cluster. You can view the status of a cluster by using the Amazon DocumentDB console or the AWS CLI `describe-db-clusters` command.

Topics

- Cluster status values
- Monitoring a cluster's status

Cluster status values

The following table lists the valid values for a cluster's status.

Cluster Status	Description
`active`	The cluster is active. This status applies to elastic clusters only.
`available`	The cluster is healthy and available. This status applies to instance-based clusters only.
`backing-up`	The cluster is currently being backed up.
`creating`	The cluster is being created. It is inaccessible while it is being created.
`deleting`	The cluster is being deleted. It is inaccessible while it is being deleted.
`failing-over`	A failover from the primary instance to an Amazon DocumentDB replica is being performed.
`inaccessible-encryption-credentials`	The AWS KMS key used to encrypt or decrypt the cluster can't be accessed.
`maintenance`	A maintenance update is being applied to the cluster. This status is used for cluster-level maintenance that Amazon

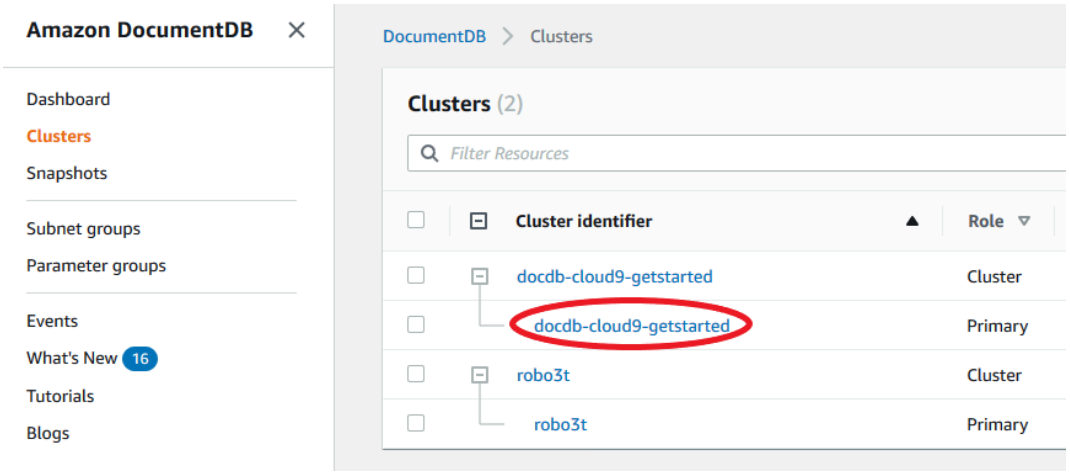
Cluster Status	Description
	DocumentDB schedules well in advance.
`migrating`	A cluster snapshot is being restored to a cluster.
`migration-failed`	A migration failed.
`modifying`	The cluster is being modified because of a customer request to modify the cluster.
`renaming`	The cluster is being renamed because of a customer request to rename it.
`resetting-master-credentials`	The master credentials for the cluster are being reset because of a customer request to reset them.
`upgrading`	The cluster engine version is being upgraded.

Monitoring a cluster's status

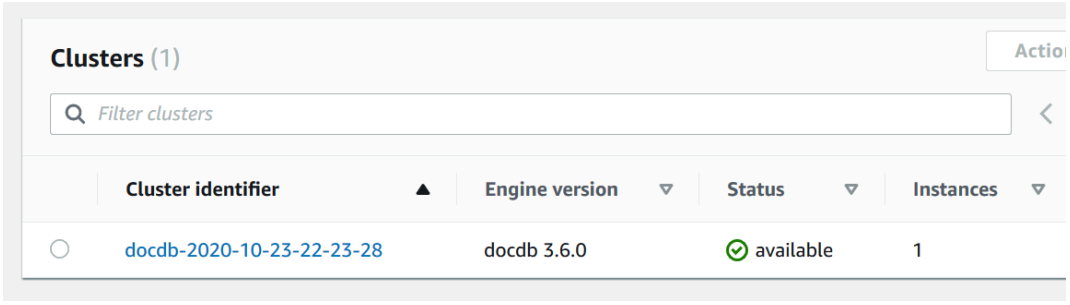
Using the AWS Management Console

When using the AWS Management Console to determine the status of a cluster, use the following procedure.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.
3. In the Clusters navigation box, you'll see the column **Cluster identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. In the **Cluster identifier** column, find the name of the instance that you are interested in. Then, to find the status of the instance, read across that row to the **Status** column, as shown below.



Using the AWS CLI

When using the AWS CLI to determine the status of a cluster, use the `describe-db-clusters` operation. The following code finds the status of the cluster `sample-cluster`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster  \
    --query 'DBClusters[*].[DBClusterIdentifier,Status]'
```

For Windows:

```
aws docdb describe-db-clusters ^
    --db-cluster-identifier sample-cluster  ^
    --query 'DBClusters[*].[DBClusterIdentifier,Status]'
```

Output from this operation looks something like the following.

```
[
    [
        "sample-cluster",
        "available"
    ]
]
```

Monitoring an Amazon DocumentDB instance's status

Amazon DocumentDB provides information about the current condition of each configured instance in the database.

There are three types of status that you can view for an Amazon DocumentDB instance:

- Instance status: This status is shown in the **Status** column of the cluster table in the AWS Management Console and shows the current life cycle condition of the instance. The values shown in the **Status** column are derived from the `Status` field of the `DescribeDBCluster` API response.
- Instance health status: This status is shown in the **Instance health** column of the cluster table in the AWS Management Console and shows whether the database engine, the component responsible for managing and retrieving data, is running. The values shown in the **Instance health** column are based on the Amazon CloudWatch `EngineUptime` system metric.
- Maintenance status: This status is shown in the **Maintenance** column of the cluster table in the AWS Management Console and indicates the status of any maintenance event that needs to be applied to an instance. Maintenance status is independent of the other instance status' and is derived from the `PendingMaintenanceAction` API. For more information about maintenance status, see Maintaining Amazon DocumentDB.

Topics

- Instance status values
- Monitoring instance status using the AWS Management Console or AWS CLI
- Instance health status values
- Monitoring instance health status using the AWS Management Console

Instance status values

The following table lists the possible status values for instances and how you are billed for each status. It shows if you will be billed for the instance and storage, only storage, or not billed. For all instance statuses, you are always billed for backup usage.

Instance status	Billed	Description
`available`	Billed	The instance is healthy and available.
`backing-up`	Billed	The instance is currently being backed up.
`configuring-log-exports`	Billed	Publishing log files to Amazon CloudWatch Logs is being enabled or disabled for this instance.
`creating`	Not billed	The instance is being created. The instance is not accessible while it is being created.
`deleting`	Not billed	The instance is being deleted.
`failed`	Not billed	The instance has failed and Amazon DocumentDB was unable to recover it. To recover the data, perform a point-in-time restore to the latest restorable time of the instance.
`inaccessible-encryption-credentials`	Not billed	The AWS KMS key that is used to encrypt or decrypt the instance could not be accessed.
`incompatible-network`	Not billed	Amazon DocumentDB is attempting to perform a recovery action on an instance but is unable to do so because the VPC is in a state that is preventing the action from being completed. This status can occur if, for example, all available IP addresses in a subnet were in use and Amazon

Instance status	Billed	Description
		DocumentDB was unable to get an IP address for the instance.
`maintenance`	Billed	Amazon DocumentDB is applying a maintenance update to the instance. This status is used for instance-level maintenance that Amazon DocumentDB schedules well in advance. We're evaluating ways to expose additional maintenance actions to customers through this status.
`modifying`	Billed	The instance is being modified because of a request to modify the instance.
`rebooting`	Billed	The instance is being rebooted because of a request or an Amazon DocumentDB process that requires the rebooting of the instance.
`renaming`	Billed	The instance is being renamed because of a request to rename it.
`resetting-master-credentials`	Billed	The master credentials for the instance are being reset because of a request to reset them.
`restore-error`	Billed	The instance encountered an error attempting to restore to a point-in-time or from a snapshot.
`starting`	Billed for storage	The instance is starting.
`stopped`	Billed for storage	The instance is stopped.
`stopping`	Billed for storage	The instance is being stopped.

Instance status	Billed	Description
`storage-full`	Billed	The instance has reached its storage capacity allocation. This is a critical status and should be remedied immediately; scale up your storage by modifying the instance. Set Amazon CloudWatch alarms to warn you when storage space is getting low so you don't run into this situation.

Monitoring instance status using the AWS Management Console or AWS CLI

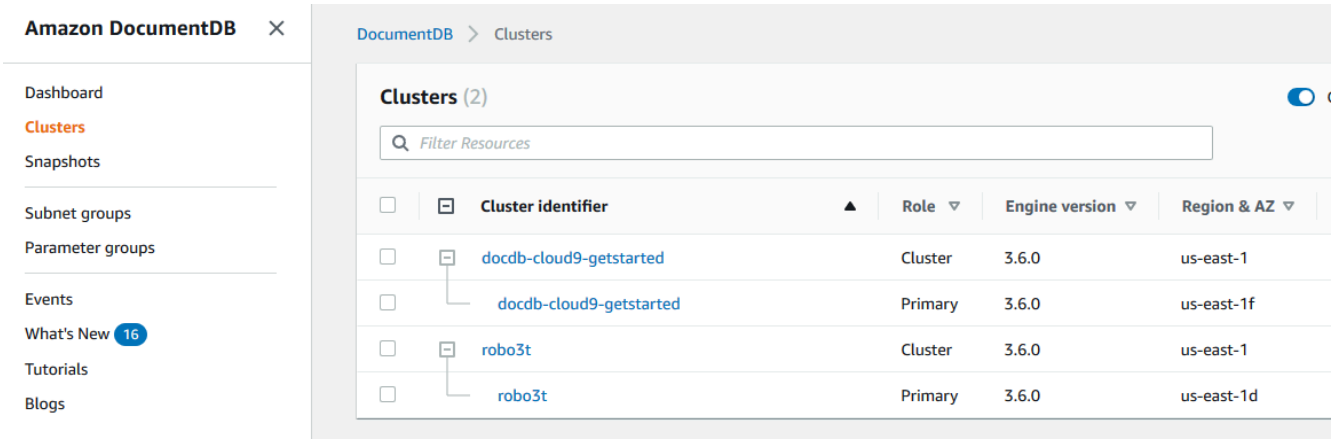
Use the AWS Management Console or AWS CLI to monitor the status of you instance.

Using the AWS Management Console

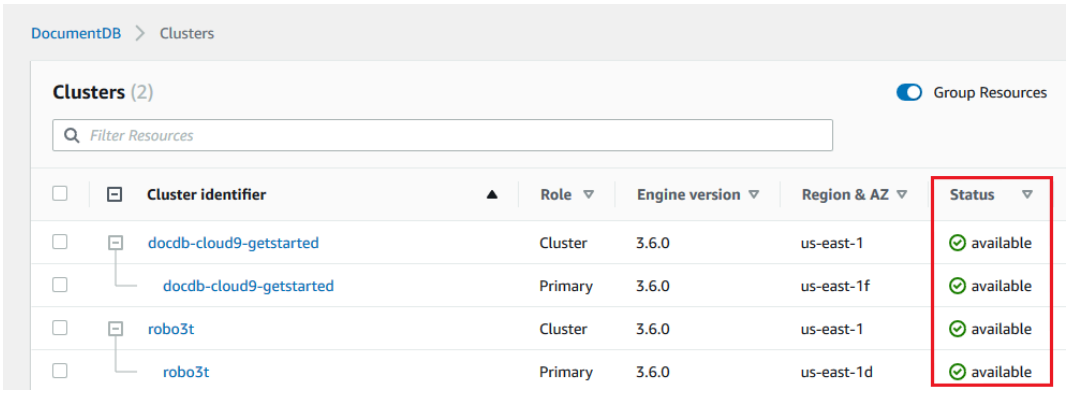
When using the AWS Management Console to determine the status of a cluster, use the following procedure.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

> **Note**
>
> Note that in the Clusters navigation box, the column **Cluster identifier** shows both clusters and instances. Instances are listed underneath clusters, similar to the following image.



3. Find the name of the instance that you are interested in. Then, to find the status of the instance, read across that row to the **Status** column, as shown following.



Using the AWS CLI

When using the AWS CLI to determine the status of a cluster, use the `describe-db-instances` operation. The following code finds the status of the instance `sample-cluster-instance-01`.

For Linux, macOS, or Unix:

```
aws docdb describe-db-instances \
          --db-instance-identifier sample-cluster-instance-01  \
          --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceStatus]'
```

For Windows:

```
aws docdb describe-db-instances ^
```

```
          --db-instance-identifier sample-cluster-instance-01  ^
          --query 'DBInstances[*].[DBInstanceIdentifier,DBInstanceStatus]'
```

Output from this operation looks something like the following.

```
[
          [
              "sample-cluster-instance-01",
              "available"
          ]
      ]
```

Instance health status values

The following table lists the possible health status values for instances. The **Instance health** column, located in the **Clusters** table in the AWS Management Console, shows whether the database engine, the component responsible for storing, managing, and retrieving data, is operating normally. This column also indicates if the `EngineUptime` system metric, available in CloudWatch, is showing the health status of each instance.

Instance health status	Description
healthy	Database engine is running in the Amazon DocumentDB instance.
unhealthy	Database engine is not running or has restarted less than a minute ago.

Monitoring instance health status using the AWS Management Console

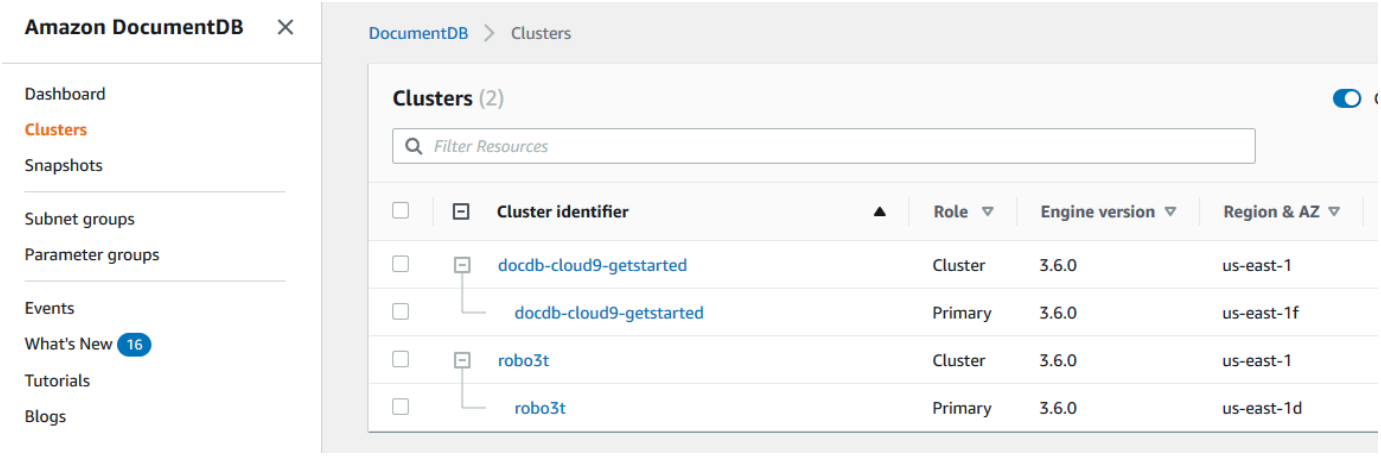
Use the AWS Management Console to monitor the health status of you instance.

When using AWS Management Console, use the following steps to understand the instance's health status.

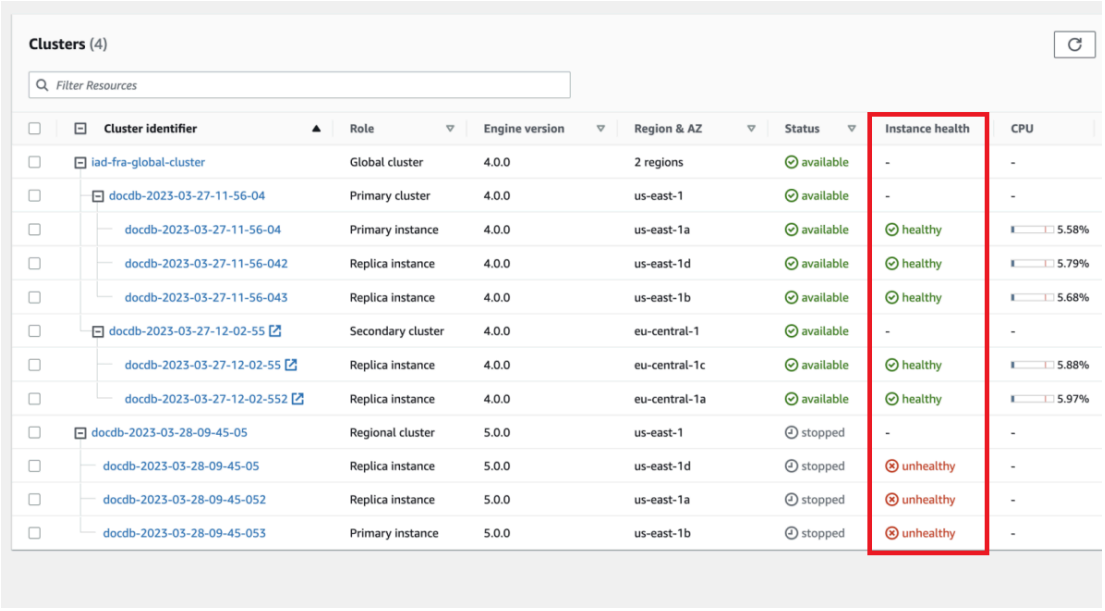
1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.

2. In the navigation pane, choose **Clusters**.

> **Note**
>
> In the **Clusters** navigation box, the column **Cluster identifier** shows both clusters and instances. Instances are listed underneath clusters, similar to the following image.



3. Find the name of the instance that you are interested in. Then, to find the status of the instance, read across that row to the **Instance health** column, as shown in the following image:



> **Note**
>
> Instance health status polling occurs every 60 seconds and is based on the CloudWatch `EngineUptime` system metric. The values in the **Instance health** column are automatically updated.

Viewing Amazon DocumentDB recommendations

Amazon DocumentDB provides a list of automated recommendations for database resources, such as instances and clusters. These recommendations provide best practice guidance by analyzing your cluster and instance configurations.

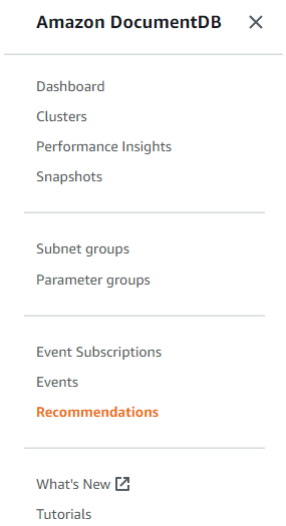
As an example of these recommendations, see the following:

Type	Description	Recommendation	Additional information
One instance	Cluster only contains one instance	Performance and availability: add another instance with the same instance class in a different Availability Zone.	Amazon DocumentDB high availability and replication

Amazon DocumentDB generates recommendations for a resource when the resource is created or modified. Amazon DocumentDB also periodically scans your resources and generates recommendations.

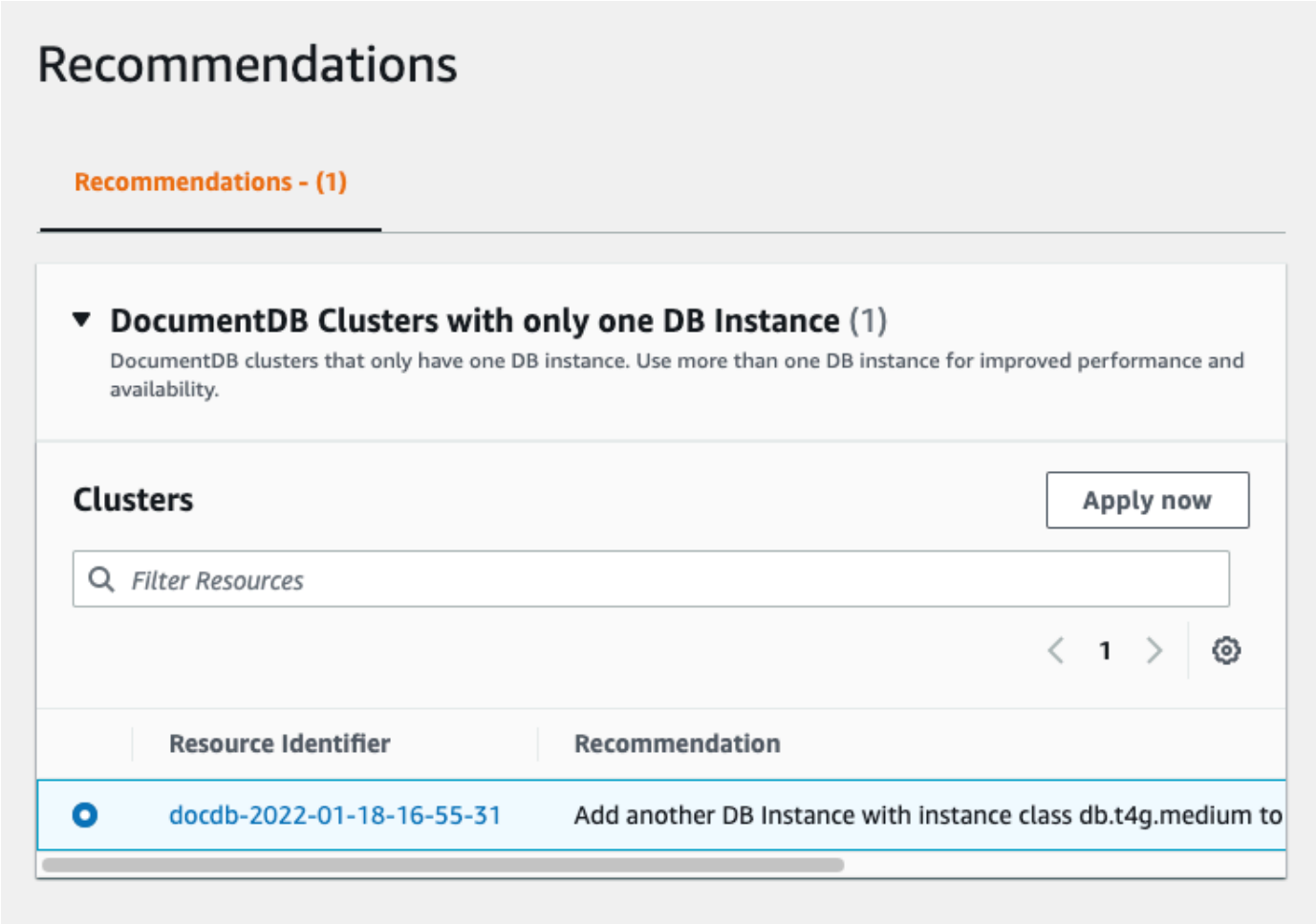
To view and take action on Amazon DocumentDB recommendations

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Recommendations**:



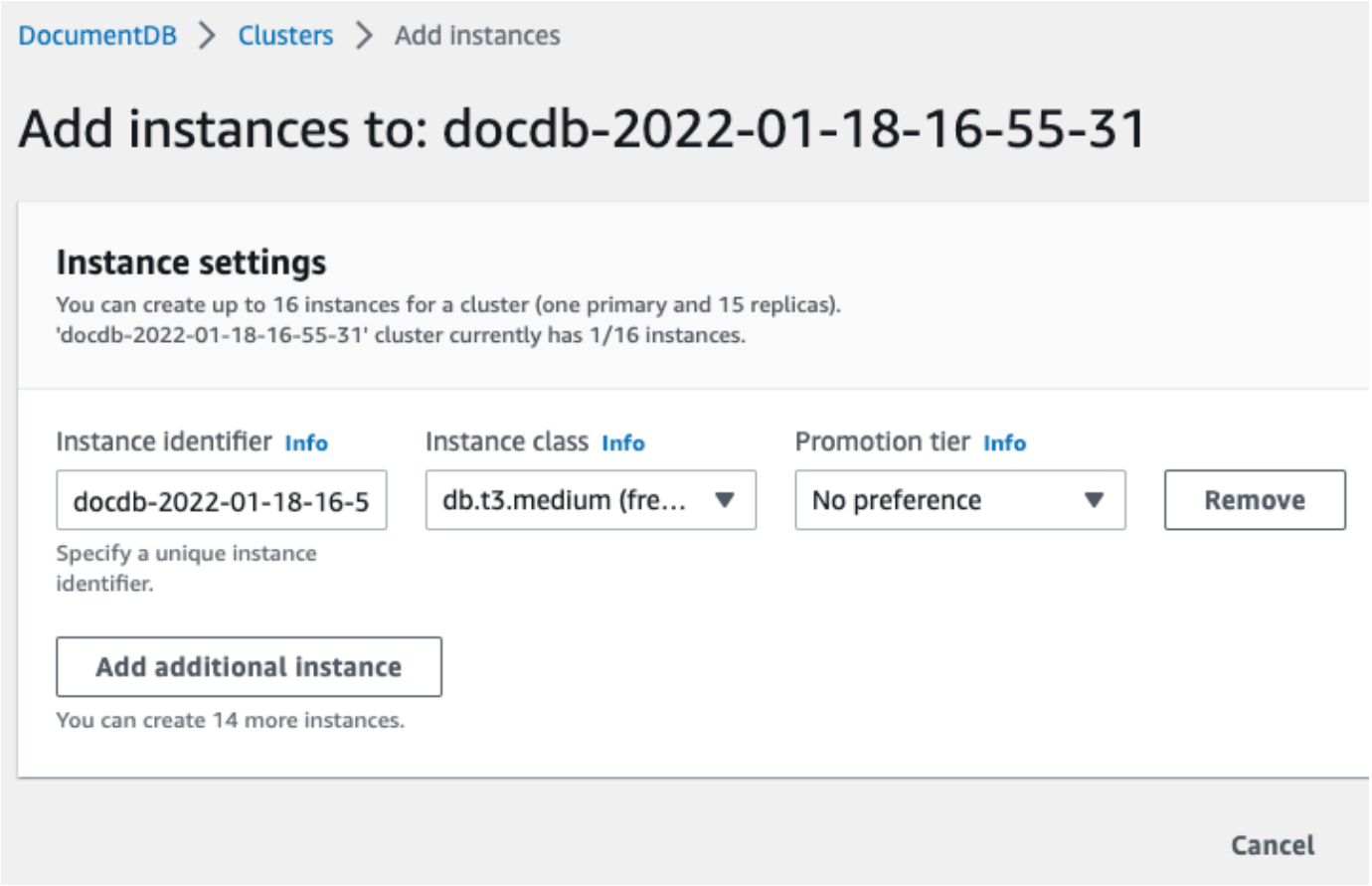
3. In the **Recommendations** dialog, expand the section of interest and select the recommended task.

   In the following example, the recommended task applies to an Amazon DocumentDB cluster with only one instance. The recommendation is to add another instance to improve performance and availability.



4. Choose **Apply now**.

   For this example, the **Add instances** dialog appears:



5. Modify your new instance's settings and choose **Create**.

Using Amazon DocumentDB event subscriptions

Amazon DocumentDB uses Amazon Simple Notification Service (Amazon SNS) to provide notifications when an Amazon DocumentDB event occurs. These notifications can be in any form that is supported by Amazon SNS for an AWS Region, such as an email, a text message, or a call to an HTTP endpoint.

Amazon DocumentDB groups these events into categories that you can subscribe to so that you can be notified when an event in that category occurs. You can subscribe to an event category for an instance, cluster, snapshot, cluster snapshot, or for a parameter group. For example, if you subscribe to the Backup category for a given instance, you are notified whenever a backup-related event occurs that affects the instance. You also receive notification when an event subscription changes.

Events occur at both the cluster and the instance level, so you can receive events if you subscribe to a cluster or an instance.

Event subscriptions are sent to the addresses you provide when you create the subscription. You might want to create several different subscriptions, such as a subscription that receives all event notifications and another subscription that includes only critical events for your production instances. You can easily turn off notification without deleting a subscription. To do so, set the **Enabled** radio button to **No** in the Amazon DocumentDB console.

> **Important**
>
> Amazon DocumentDB doesn't guarantee the order of events sent in an event stream. The event order is subject to change.

Amazon DocumentDB uses the Amazon Resource Name (ARN) of an Amazon SNS topic to identify each subscription. The Amazon DocumentDB console creates the ARN for you when you create the subscription.

Billing for Amazon DocumentDB event subscriptions is through Amazon SNS. Amazon SNS fees apply when using event notification. For more information, see Amazon Simple Notification Service Pricing. Other than Amazon SNS charges, Amazon DocumentDB does not bill for event subscriptions.

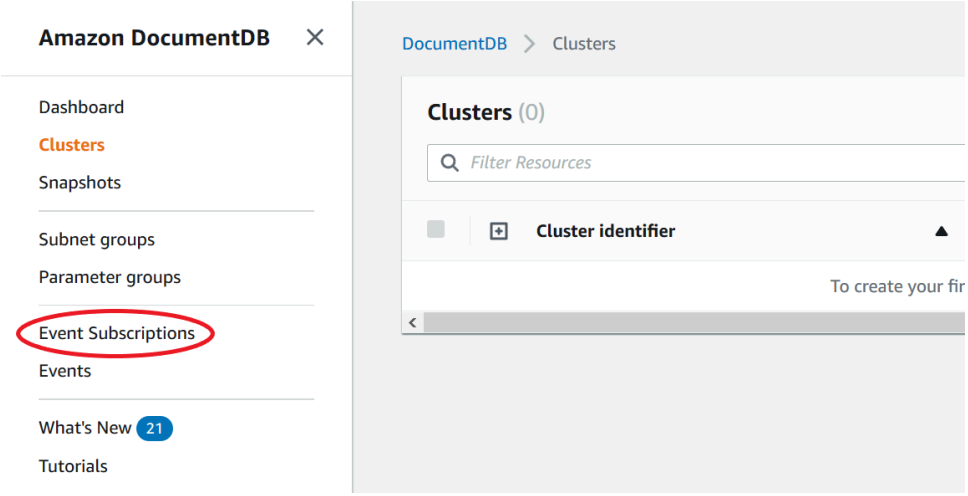
Topics

- Subscribing to Amazon DocumentDB events
- Managing Amazon DocumentDB event notification subscriptions
- Amazon DocumentDB event categories and messages

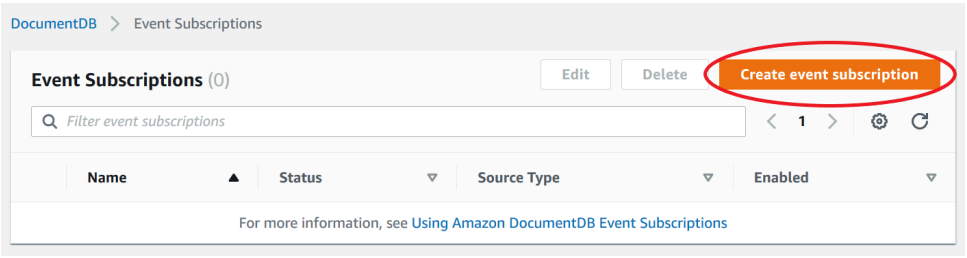
Subscribing to Amazon DocumentDB events

You can use the Amazon DocumentDB console to subscribe to event subscriptions, as follows:

1. Sign in to the AWS Management Console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Event subscriptions**.

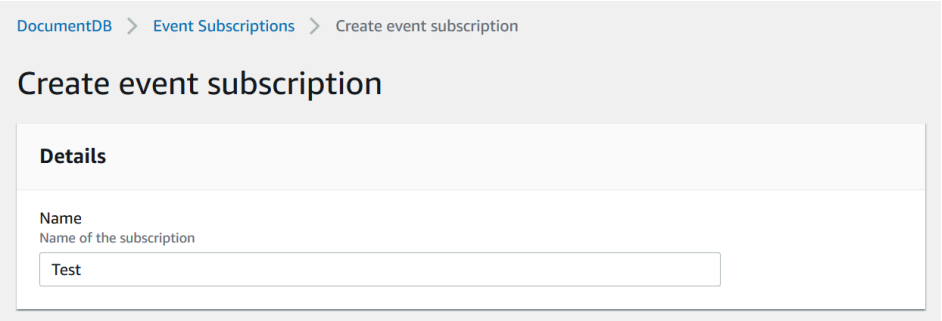


3. In the **Event subscriptions** pane, choose **Create event subscription**.

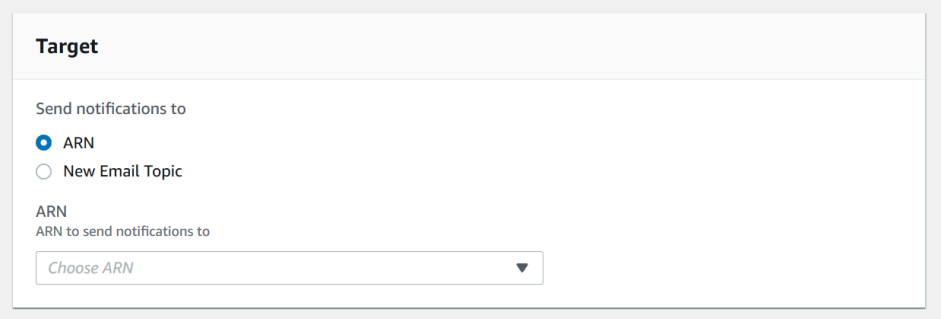


4. In the **Create event subscription** dialog box, do the following:

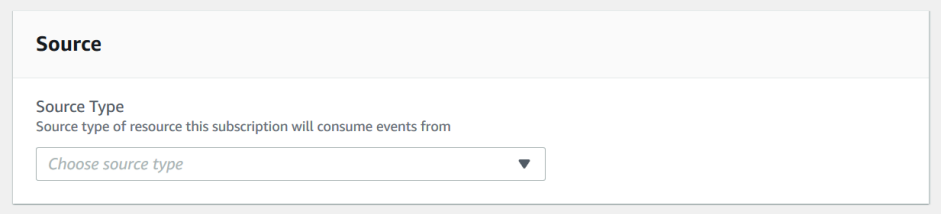
   - For **Name**, enter a name for the event notification subscription.



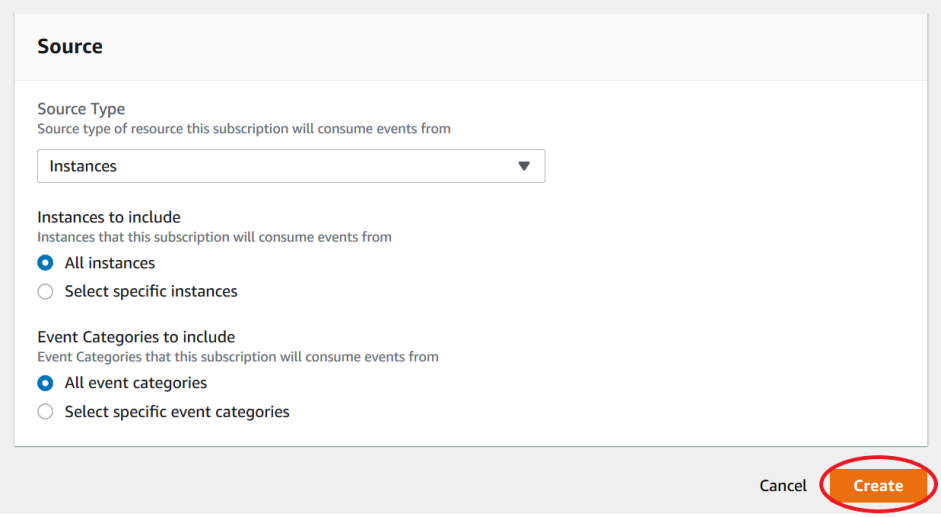
   - For **Target**, choose where you want to send notifications to. You can choose an existing **ARN** or choose **New Email Topic** to enter the name of a topic and a list of recipients.



- For **Source**, choose a source type. Depending on the source type you selected, choose the event categories and the sources that you want to receive event notifications from.



- Choose **Create**.



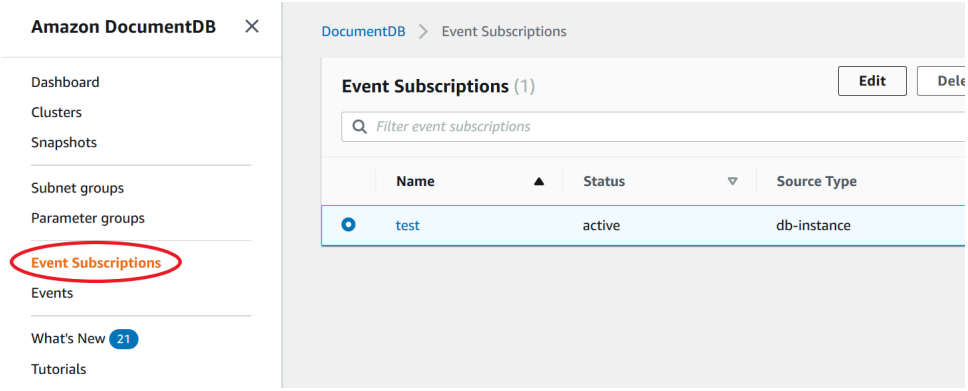
Managing Amazon DocumentDB event notification subscriptions

If you choose **Event subscriptions** in the navigation pane of the Amazon DocumentDB console, you can view subscription categories and a list of your current subscriptions. You can also modify or delete a specific subscription.

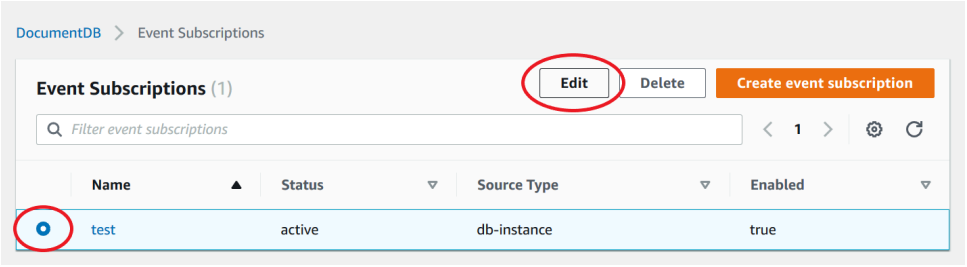
To modify your current Amazon DocumentDB event notification subscriptions

1. Sign in to the AWS Management Console at https://console.aws.amazon.com/docdb.

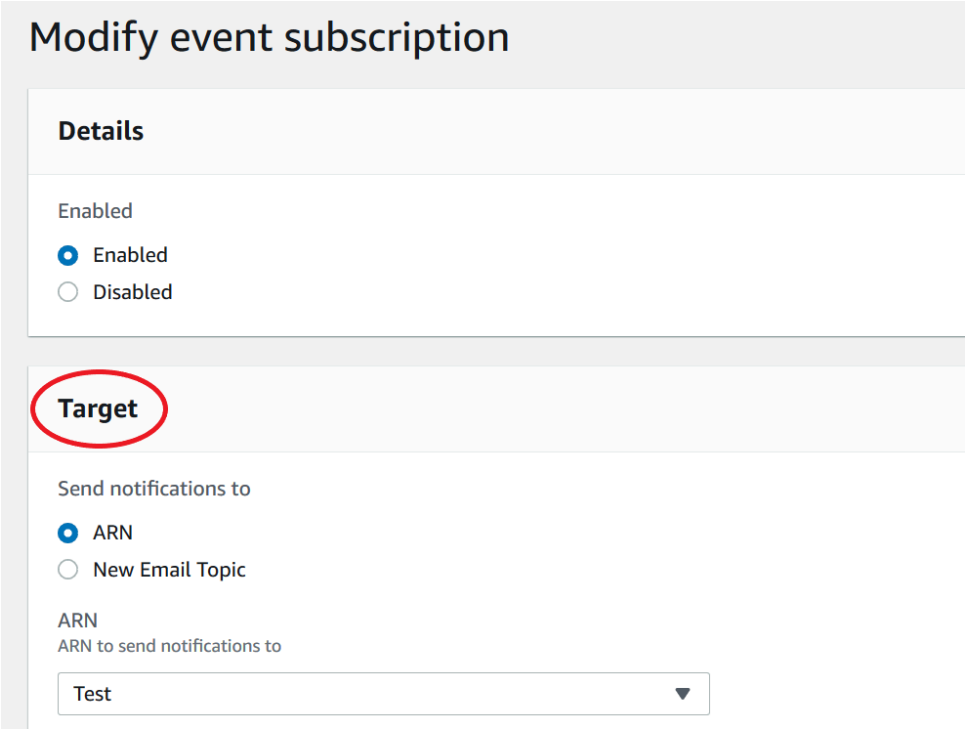
2. In the navigation pane, choose **Event subscriptions**. The **Event subscriptions** pane shows all your event notification subscriptions.



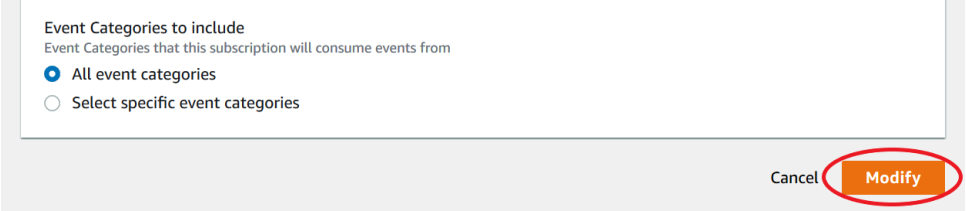
3. In the **Event subscriptions** pane, choose the subscription that you want to modify and choose **Edit**.



4. Make your changes to the subscription in either the **Target** or **Source** section. You can add or remove source identifiers by selecting or deselecting them in the Source section.



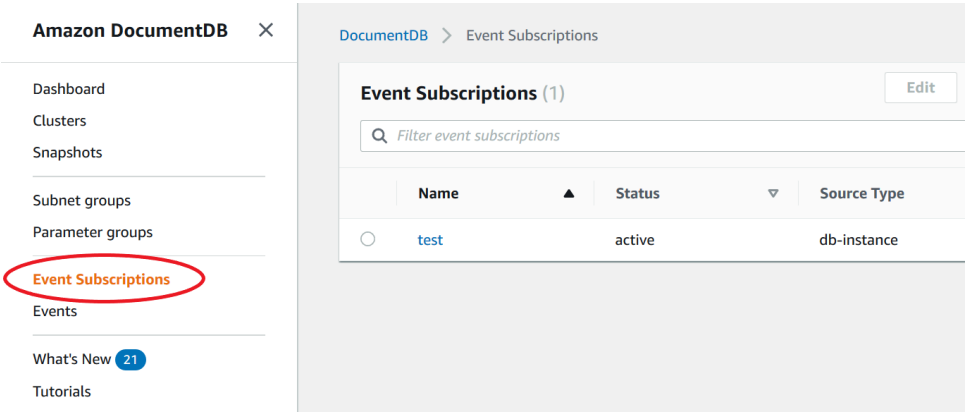
5. Choose **Modify**. The Amazon DocumentDB console indicates that the subscription is being modified.



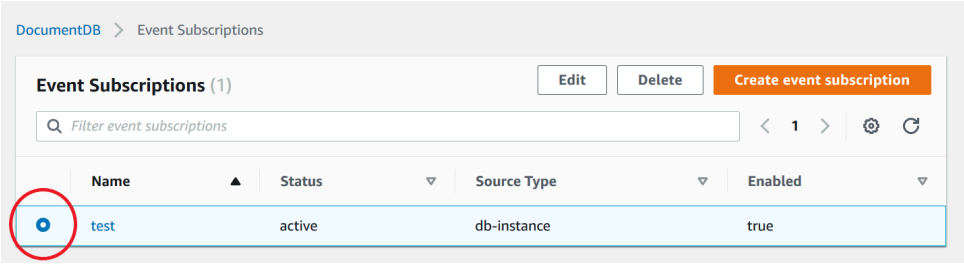
Deleting an Amazon DocumentDB event notification subscription

You can delete a subscription when you no longer need it. All subscribers to the topic will no longer receive event notifications specified by the subscription.

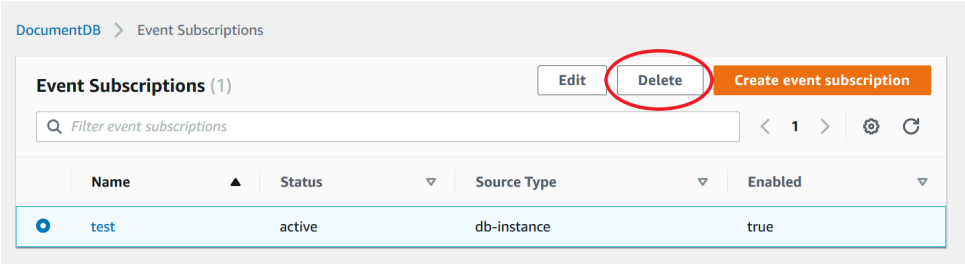
1. Sign in to the AWS Management Console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Event subscriptions**.



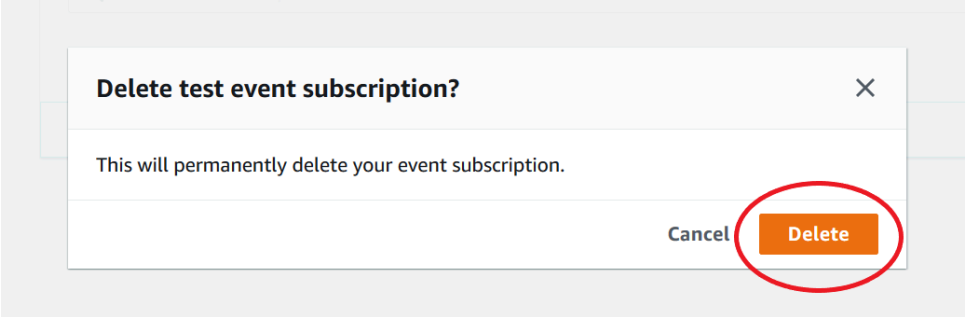
3. In the **Event subscriptions** pane, choose the subscription that you want to delete.



4. Choose **Delete**.



5. A pop-up window will appear asking you if you want to permanently delete this notification. Choose **Delete**.



Amazon DocumentDB event categories and messages

Amazon DocumentDB generates a significant number of events in categories that you can subscribe to using the console. Each category applies to a source type, which can be an instance, cluster, snapshot, or parameter group.

> **Note**
>
> Amazon DocumentDB uses existing Amazon RDS event definitions and IDs.

Amazon DocumentDB events originating from instances

Category	Description
availability	The instance restarted.
availability	The instance shutdown.
configuration change	Applying modification to an instance class.

Category	Description
configuration change	Finished applying modification to an instance class.
configuration change	Reset primary credentials.
creation	Instance created.
deletion	Instance deleted
failure	The instance has failed due to an incompatible configuration or an underlying storage issue. Begin a point-in-time-restore for the instance.
notification	Instance stopped.
notification	Instance started.
notification	Instance is being started due to it exceeding the maximum allowed time being stopped.
recovery	Recovery of the instance has started. Recovery time will vary with the amount of data to be recovered.
recovery	Recovery of the instance is complete.
security patching	The operating system update is available for your instance. For information about applying updates, see Maintaining Amazon DocumentDB.

Amazon DocumentDB events originating from a cluster

Category	Description
creation	Cluster created

Category	Description
deletion	Cluster deleted.
failover	Promoting previous primary again.
failover	Completed failover to instance.
failover	Started failover to DB instance: %s
failover	Started same AZ failover to DB instance: %s
failover	Started cross AZ failover to DB instance: %s
maintenance	Cluster has been patched.
maintenance	Database cluster is in a state that cannot be upgraded: %s
notification	The cluster stopped.
notification	The cluster started.
notification	The cluster stop failed.
notification	The cluster is being started due to it exceeding the maximum allowed time being stopped.
notification	Renamed cluster from %s to %s.

Amazon DocumentDB events originating from cluster snapshot

The following table shows the event category and a list of events when an Amazon DocumentDB cluster snapshot is the source type.

Category	Description
backup	Creating manual cluster snapshot.
backup	Manual cluster snapshot created.

Category	Description
backup	Creating automated cluster snapshot.
backup	Automated cluster snapshot created.

Amazon DocumentDB events originating from parameter group

The following table shows the event category and a list of events when a parameter group is the source type.

Category	Description
configuration change	Updated parameter %s to %s with apply method %s

Monitoring Amazon DocumentDB with CloudWatch

Amazon DocumentDB (with MongoDB compatibility) integrates with Amazon CloudWatch so that you can gather and analyze operational metrics for your clusters. You can monitor these metrics using the CloudWatch console, the Amazon DocumentDB console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API.

CloudWatch also lets you set alarms so that you can be notified if a metric value breaches a threshold that you specify. You can even set up Amazon CloudWatch Events to take corrective action if a breach occurs. For more information about using CloudWatch and alarms, see the Amazon CloudWatch documentation.

Topics

- Amazon DocumentDB metrics
- Viewing CloudWatch data
- Amazon DocumentDB dimensions
- Monitoring opcounter metrics
- Monitoring database connections

Amazon DocumentDB metrics

To monitor the health and performance of your Amazon DocumentDB cluster and instances, you can view the following metrics in the Amazon DocumentDB console.

> **Note**
>
> Unless otherwise noted, metrics in the following tables apply to both instance-based and elastic clusters.

Topics

- Resource utilization metrics
- Latency metrics
- NVMe-backed instance metrics
- Operations metrics
- Command-level metrics
- Throughput metrics
- System metrics
- T3 instance metrics

Resource utilization metrics

Metric	Description	
`BackupRetentionPeriodStorageUsed`	The total amount of backup storage in bytes used to support the point-in-time restore feature within the Amazon DocumentDB's retention window. Included in the total reported by the `TotalBackupStorageBilled` metric. Computed	

Metric	Description
	separately for each Amazon DocumentDB cluster.
ChangeStreamLogSize	The amount of storage used by your cluster to store the change stream log in megabytes. This value is a subset of the total storage for the cluster (VolumeBytesUsed) and affects the cost of the cluster. For storage pricing information, see the Amazon DocumentDB product page. The change stream log size is a function of how much change is happening on your cluster and the change stream long retention duration. For more information on change streams, see Using change streams with Amazon DocumentDB.
CPUUtilization	The percentage of CPU used by an instance.
DatabaseConnections	The number of connections (active and idle) open on an instance taken at a 1-minute frequency.
DatabaseConnectionsMax	The maximum number of open database connections (active and idle) on an instance in a 1-minute period.

Metric	Description
DatabaseConnectionsLimit	The maximum number of concurrent database connections (active and idle) allowed on an instance at any given time.
DatabaseConnectionsActiveMax	The maximum number of active database connections on an instance in a 1-minute period. Active connections are those currently running an operation, as opposed to idle connections. For the active connection quotas per instance type, see Instance quotas.
DatabaseCursors	The number of cursors open on an instance taken at a 1-minute frequency.
DatabaseCursorsMax	The maximum number of open cursors on an instance in a 1-minute period.
DatabaseCursorsLimit	The maximum number of cursors allowed on an instance at any given time.
DatabaseCursorsTimedOut	The number of cursors that timed out in a 1-minute period.
FreeableMemory	The amount of available random access memory, in bytes.

Metric	Description
FreeLocalStorage	This metric reports the amount of storage available to each instance for temporary tables and logs. This value depends on the instance class. You can increase the amount of free storage space for an instance by choosing a larger instance class for your instance. (This doesn't apply to DocumentDB Serverless.)
LowMemThrottleQueueDepth	The queue depth for requests that are throttled due to low available memory taken at a 1-minute frequency.
LowMemThrottleMaxQueueDepth	The maximum queue depth for requests that are throttled due to low available memory in a 1-minute period.
LowMemNumOperationsThrottled	The number of requests that are throttled due to low available memory in a 1-minute period.

Metric	Description
SnapshotStorageUsed	The total amount of backup storage in bytes consumed by all snapshots for a given Amazon DocumentDB cluster outside its backup retention window. Included in the total reported by the TotalBackupStorageBilled metric. Computed separately for each Amazon DocumentDB cluster.
SwapUsage	The amount of swap space used on the instance.
TLSHandshakesClassical	The number of successful TLS handshakes that didn't use post-quantum key exchange on an instance in a 1-minute period. The sum of TLSHandshakesClassical and TLSHandshakesPostQuantum is the total number of successful TLS handshakes. To calculate the percentage of handshakes that used post-quantum key exchange, divide the TLSHandshakesPostQuantum value by this sum, and then multiply the result by 100. This metric applies to instance-based clusters that run Amazon DocumentDB 8.0.3 or later.

Metric	Description
TLSHandshakesPostQuantum	The number of successful TLS handshakes that used hybrid post-quantum key exchange on an instance in a 1-minute period. This metric applies to instance-based clusters that run Amazon DocumentDB 8.0.3 or later. For more information about post-quantum key exchange, see Amazon DocumentDB post-quantum key exchange.
TotalBackupStorageBilled	The total amount of backup storage in bytes for which you are billed for a given Amazon DocumentDB cluster. Includes the backup storage measured by the BackupRetentionPeriodStorageUsed and SnapshotStorageUsed metrics. Computed separately for each Amazon DocumentDB cluster.
TransactionsOpen	The number of transactions open on an instance taken at a 1-minute frequency.
TransactionsOpenMax	The maximum number of transactions open on an instance in a 1-minute period.

Metric	Description
TransactionsOpenLimit	The maximum number of concurrent transactions allowed on an instance at any given time.
VolumeBytesUsed	The amount of storage used by your cluster in bytes. This value affects the cost of the cluster. For pricing information, see the Amazon DocumentDB product page.

Latency metrics

Metric	Description
DBClusterReplicaLagMaximum	The maximum amount of lag, in milliseconds, between the primary instance and each Amazon DocumentDB instance in the cluster.
DBClusterReplicaLagMinimum	The minimum amount of lag, in milliseconds, between the primary instance and each replica instance in the cluster.
DBInstanceReplicaLag	The amount of lag, in milliseconds, when replicating updates from the primary instance to a replica instance.
ReadLatency	The average amount of time taken per disk I/O operation.

Metric	Description
WriteLatency	The average amount of time, in milliseconds, taken per disk I/O operation.

NVMe-backed instance metrics

Metric	Description
NVMeStorageCacheHitRatio	The percentage of requests that are served by the tiered cache.
FreeNVMeStorage	The amount of available Ephemeral NVMe storage.
ReadIOPSNVMeStorage	The average number of disk read I/O operations to Ephemeral NVMe storage.
ReadLatencyNVMeStorage	The average amount of time taken per disk read I/O operation for Ephemeral NVMe storage.
ReadThroughputNVMeStorage	The average number of bytes read from disk per second for Ephemeral NVMe storage.
WriteIOPSNVMeStorage	The average number of disk write I/O operations to Ephemeral NVMe storage.
WriteLatencyNVMeStorage	The average amount of time taken per disk write I/O

Metric	Description
	operation for Ephemeral NVMe storage.
WriteThroughputNVMeStorage	The average number of bytes written to disk per second for Ephemeral NVMe storage.

Operations metrics

Metric	Description
DocumentsDeleted	The number of deleted documents in a 1-minute period.
DocumentsInserted	The number of inserted documents in a 1-minute period.
DocumentsReturned	The number of returned documents in a 1-minute period.
DocumentsUpdated	The number of updated documents in a 1-minute period.
OpcountersCommand	The number of commands issued in a 1-minute period.
OpcountersDelete	The number of delete operations issued in a 1-minute period.
OpcountersGetmore	The number of getmores issued in a 1-minute period.

Metric	Description
OpcountersInsert	The number of insert operations issued in a 1-minute period.
OpcountersQuery	The number of queries issued in a 1-minute period.
OpcountersUpdate	The number of update operations issued in a 1-minute period.
TransactionsStarted	The number of transactions started on an instance in a 1-minute period.
TransactionsCommitted	The number of transactions committed on an instance in a 1-minute period.
TransactionsAborted	The number of transactions aborted on an instance in a 1-minute period.
TTLDeletedDocuments	The number of documents deleted by a TTLMonitor in a 1-minute period.

Command-level metrics

Amazon DocumentDB publishes per-command latency and concurrency metrics to CloudWatch. These metrics help you identify which operations contribute most to your workload and detect tail-latency issues or concurrency pressure.

Each command in the following table emits three metrics in a 1-minute period:

- AvgDuration — The average execution time of the command, in microseconds.

- `P100Duration` — The maximum execution time of the command (100th percentile), in microseconds.
- `MaxConcurrent` — The maximum number of concurrent executions of the command.

Metric names follow the pattern `Command.`*`commandName`*`.`*`metricName`*. For example, `Command.find.AvgDuration` or `Command.aggregate.MaxConcurrent`.

Metric	Description
`Command.find`	Read queries using find.
`Command.insert`	Document insert operations.
`Command.findAndModify`	Atomic find-and-modify operations.
`Command.update`	Document update operations.
`Command.delete`	Document delete operations.
`Command.aggregate`	Aggregation pipeline executions.
`Command.count`	Count operations.
`Command.distinct`	Distinct value queries.
`Command.getMore`	Cursor batch retrieval operations.
`Command.abortTransaction`	Transaction abort operations.
`Command.commitTransaction`	Transaction commit operations.

Throughput metrics

Metric	Description
NetworkReceiveThroughput	The amount of network throughput, in bytes per second, received from clients by each instance in the cluster. This throughput doesn't include network traffic between instances in the cluster and the cluster volume.
NetworkThroughput	The amount of network throughput, in bytes per second, both received from and transmitted to clients by each instance in the Amazon DocumentDB cluster. This throughput doesn't include network traffic between instances in the cluster and the cluster volume.
NetworkTransmitThroughput	The amount of network throughput, in bytes per second, sent to clients by each instance in the cluster. This throughput doesn't include network traffic between instances in the cluster and the cluster volume.
ReadIOPS	The average number of disk read I/O operations per

Metric	Description
	second. Amazon DocumentDB reports read and write IOPS separately, and on 1-minute intervals.
ReadThroughput	The average number of bytes read from disk per second.
StorageNetworkReceiveThroughput	The amount of network throughput, in bytes per second, received from the Amazon DocumentDB cluster storage volume by each instance in the cluster.
StorageNetworkTransmitThroughput	The amount of network throughput, in bytes per second, sent to the Amazon DocumentDB cluster storage volume by each instance in the cluster.
StorageNetworkThroughput	The amount of network throughput, in bytes per second, received and sent to the Amazon DocumentDB cluster storage volume by each instance in the Amazon DocumentDB cluster.

Metric	Description
VolumeReadIOPs	The average number of billed read I/O operations from a cluster volume, reported at 5-minute intervals. Billed read operations are calculated at the cluster volume level, aggregated from all instances in the cluster, and then reported at 5-minute intervals. The value is calculated by taking the value of the read operations metric over a 5-minute period. You can determine the amount of billed read operations per second by taking the value of the billed read operations metric and dividing by 300 seconds. For example, if the VolumeReadIOPs returns 13,686, then the billed read operations per second is 45 (13,686 / 300 = 45.62). You accrue billed read operations for queries that request database pages that are not present in the buffer cache and therefore must be loaded from storage. You might see spikes in billed read operations as query results are read from storage and

Metric	Description	
	then loaded into the buffer cache.	

Metric	Description
VolumeWriteIOPs	The average number of billed write I/O operations from a cluster volume, reported at 5-minute intervals. Billed write operations are calculated at the cluster volume level, aggregated from all instances in the cluster, and then reported at 5-minute intervals. The value is calculated by taking the value of the write operations metric over a 5-minute period. You can determine the amount of billed write operations per second by taking the value of the billed write operations metric and dividing by 300 seconds. For example, if the VolumeWriteIOPs returns 13,686, then the billed write operations per second is 45 (13,686 / 300 = 45.62). Note that VolumeReadIOPs and VolumeWriteIOPs metrics are calculated by the DocumentDB storage layer and it includes IOs performed by the primary and replica instances. The data is aggregated every 20-30 minutes and then reported

Metric	Description	
	at 5-minute intervals, thus emitting the same data point for the metric in the time period. If you are looking for a metric to correlate to your insert operations over a 1-minute interval, you can use the instance level WriteIOPS metric. The metric is available in the monitoring tab of your Amazon DocumentDB primary instance.	
`WriteIOPS`	The average number of disk write I/O operations per second. When used on a cluster level, `WriteIOPs` are evaluated across all instances in the cluster. Read and write IOPS are reported separately, on 1-minute intervals.	
`WriteThroughput`	The average number of bytes written to disk per second.	

System metrics

Metric	Description	
`AvailableMVCCIds`	A counter that shows the number of remaining write operations available before reaching zero. When this counter reaches zero, your cluster will enter read-only	

Metric	Description
	mode until IDs are reclaimed and recycled. The counter decreases with each write operation and increases as garbage collection recycles old MVCC IDs.
BufferCacheHitRatio	The percentage of requests that are served by the buffer cache.
DiskQueueDepth	The number of I/O operations that are waiting to be written to or read from disk.
EngineUptime	The amount of time, in seconds, that the instance has been running.
IndexBufferCacheHitRatio	The percentage of index requests that are served by the buffer cache. You might see a spike greater than 100 percent for the metric right after you drop an index, collection or database. This will automatically be corrected after 60 seconds. This limitation will be fixed in a future patch update.

Metric	Description
LongestActiveGCRuntime	Duration in seconds of the longest active garbage collection process. Updates every minute and tracks only active operations, excluding processes that complete within the one-minute window.

T3 instance metrics

Metric	Description
CPUCreditUsage	The number of CPU credits spent during the measurement period.
CPUCreditBalance	The number of CPU credits that an instance has accrued. This balance is depleted when the CPU bursts and CPU credits are spent more quickly than they are earned.
CPUSurplusCreditBalance	The number of surplus CPU credits spent to sustain CPU performance when the CPUCreditBalance value is zero.
CPUSurplusCreditsCharged	The number of surplus CPU credits exceeding the maximum number of CPU credits that can be earned in a 24-hour period, and

Metric	Description	
	thus attracting an additiona l charge. For more informati on, see Monitoring your CPU credits.	

Viewing CloudWatch data

You can view Amazon CloudWatch data using the CloudWatch console, the Amazon DocumentDB console, AWS Command Line Interface (AWS CLI), or the CloudWatch API.

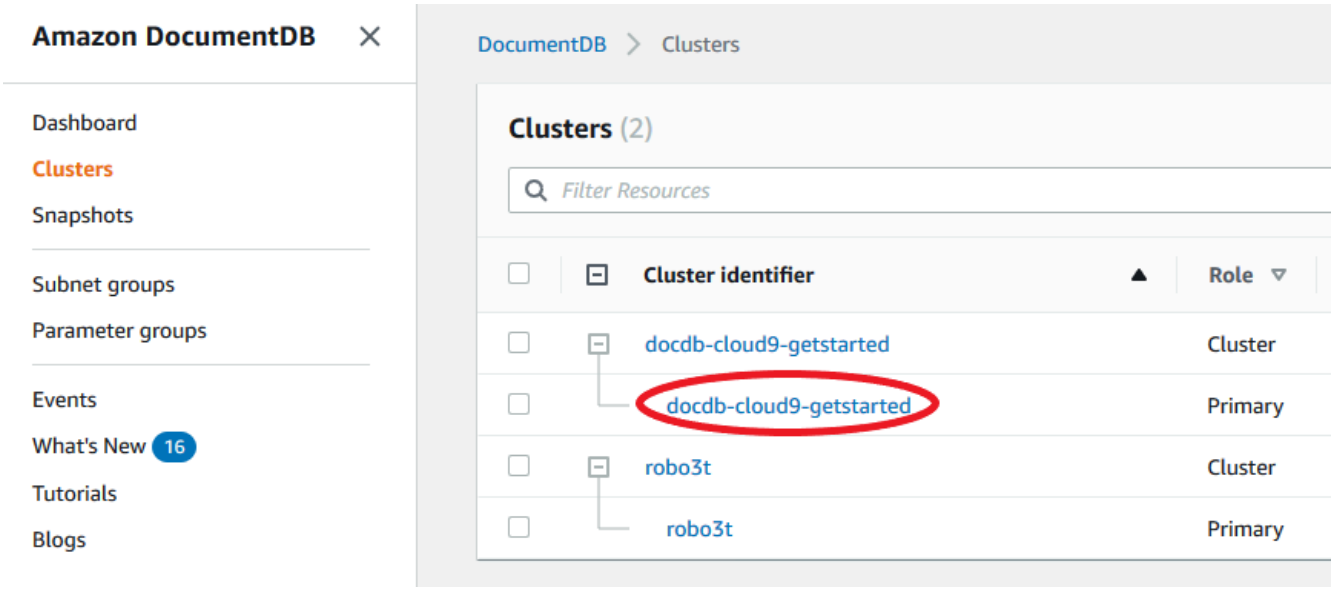
Using the AWS Management Console

To view CloudWatch metrics using the Amazon DocumentDB Management Console, complete the following steps.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

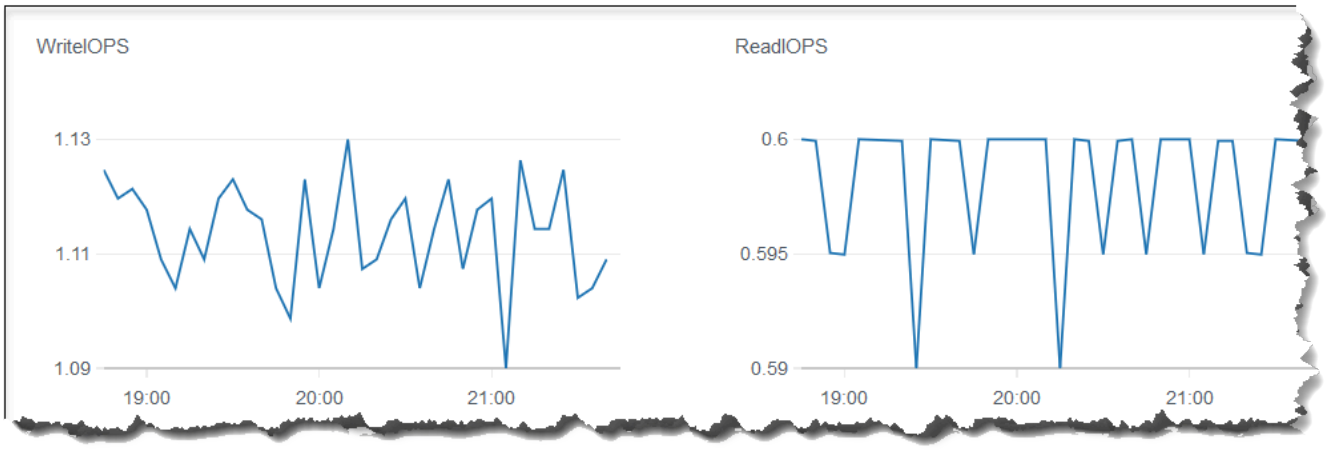
> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the Clusters navigation box, you'll see the column **Cluster Identifier**. Your instances are listed under clusters, similar to the following screenshot.



4. From the list of instances, choose the name of the instance that you want metrics for.
5. In the resulting instance summary page, choose the **Monitoring** tab to view graphical representations of your Amazon DocumentDB instance's metrics. Because a graph must be generated for each metric, it might take a few minutes for the **CloudWatch** graphs to populate.

   The following image shows the graphical representations of two CloudWatch metrics in the Amazon DocumentDB console, `WriteIOPS` and `ReadIOPS`.



Using the CloudWatch Management Console

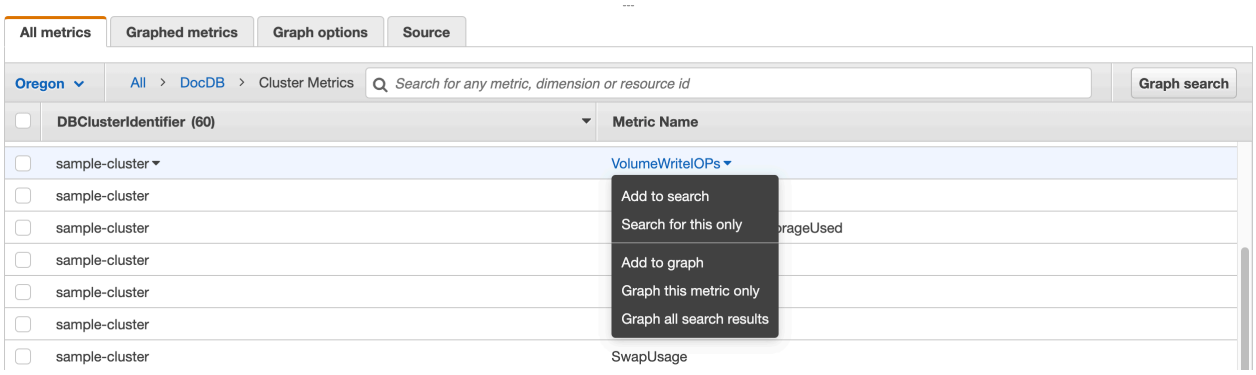
To view CloudWatch metrics using the CloudWatch Management Console, complete the following steps.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/cloudwatch.
2. In the navigation pane, choose **Metrics**. Then, from the list of service names, choose **DocDB**.
3. Choose a metric dimension (for example, **Cluster Metrics**).
4. The **All metrics** tab displays all metrics for that dimension in **DocDB**.

   a. To sort the table, use the column heading.

   b. To graph a metric, select the check box next to the metric. To select all metrics, select the check box in the heading row of the table.

   c. To filter by metric, hover over the metric name and select the dropdown arrow next to the metric name. Then, choose **Add to search**, as shown in the following image.



Using the AWS CLI

To view CloudWatch data for Amazon DocumentDB, use the CloudWatch `get-metric-statistics` operation with the following parameters.

Parameters

- **`--namespace`** — Required. The service namespace for which you want CloudWatch metrics. For Amazon DocumentDB, this must be `AWS/DocDB`.
- **`--metric-name`** — Required. The name of the metric for which you want data.
- **`--start-time`** — Required. The timestamp that determines the first data point to return.

  The value specified is inclusive; results include data points with the specified timestamp. The timestamp must be in ISO 8601 UTC format (for example, 2016-10-03T23:00:00Z).
- **`--end-time`** — Required. The timestamp that determines the last data point to return.

The value specified is inclusive; results include data points with the specified timestamp. The timestamp must be in ISO 8601 UTC format (for example, 2016-10-03T23:00:00Z).

- **`--period`** — Required. The granularity, in seconds, of the returned data points. For metrics with regular resolution, a period can be as short as one minute (60 seconds) and must be a multiple of 60. For high-resolution metrics that are collected at intervals of less than one minute, the period can be 1, 5, 10, 30, 60, or any multiple of 60.
- **`--dimensions`** — Optional. If the metric contains multiple dimensions, you must include a value for each dimension. CloudWatch treats each unique combination of dimensions as a separate metric. If a specific combination of dimensions was not published, you can't retrieve statistics for it. You must specify the same dimensions that were used when the metrics were created.
- **`--statistics`** — Optional. The metric statistics, other than percentile. For percentile statistics, use `ExtendedStatistics`. When calling `GetMetricStatistics`, you must specify either `Statistics` or `ExtendedStatistics`, but not both.

  Permitted values:

  - `SampleCount`
  - `Average`
  - `Sum`
  - `Minimum`
  - `Maximum`
- **`--extended-statistics`** — Optional. The `percentile` statistics. Specify values between p0.0 and p100. When calling `GetMetricStatistics`, you must specify either `Statistics` or `ExtendedStatistics`, but not both.
- **`--unit`** — Optional. The unit for a given metric. Metrics may be reported in multiple units. Not supplying a unit results in all units being returned. If you specify only a unit that the metric does not report, the results of the call are null.

  Possible values:

  - `Seconds`
  - `Microseconds`
  - `Milliseconds`
  - `Bytes`

- Kilobytes
- Megabytes
- Gigabytes
- Terabytes
- Bits
- Kilobytes
- Megabits
- Gigabits
- Terabits
- Percent
- Count
- Bytes/Second
- Kilobytes/Second
- Megabytes/Second
- Gigabytes/Second
- Terabytes/Second
- Bits/Second
- Kilobits/Second
- Megabits/Second
- Gigabits/Second
- Terabits/Second
- Count/Second
- None

Example

The following example finds the maximum CPUUtilization for a 2-hour period taking a sample every 60 seconds.

For Linux, macOS, or Unix:

```
aws cloudwatch get-metric-statistics \
       --namespace AWS/DocDB \
```

```
        --dimensions \
            Name=DBInstanceIdentifier,Value=docdb-2019-01-09-23-55-38 \
        --metric-name CPUUtilization \
        --start-time 2019-02-11T05:00:00Z \
        --end-time 2019-02-11T07:00:00Z \
        --period 60 \
        --statistics Maximum
```

For Windows:

```
aws cloudwatch get-metric-statistics ^
        --namespace AWS/DocDB ^
        --dimensions ^
            Name=DBInstanceIdentifier,Value=docdb-2019-01-09-23-55-38 ^
        --metric-name CPUUtilization ^
        --start-time 2019-02-11T05:00:00Z ^
        --end-time 2019-02-11T07:00:00Z ^
        --period 60 ^
        --statistics Maximum
```

Output from this operation look something like the following.

```
{
        "Label": "CPUUtilization",
        "Datapoints": [
            {
                "Unit": "Percent",
                "Maximum": 4.49152542374361,
                "Timestamp": "2019-02-11T05:51:00Z"
            },
            {
                "Unit": "Percent",
                "Maximum": 4.25000000000485,
                "Timestamp": "2019-02-11T06:44:00Z"
            },

            ********* some output omitted for brevity *********

            {
                "Unit": "Percent",
                "Maximum": 4.33333333331878,
                "Timestamp": "2019-02-11T06:07:00Z"
            }
```

```
        ]
    }
```

Amazon DocumentDB dimensions

The metrics for Amazon DocumentDB are qualified by the values for the account or operation. You can use the CloudWatch console to retrieve Amazon DocumentDB data filtered by any of the dimensions in the following table.

Dimension	Description
`DBClusterIdentifier`	Filters the data that you request for a specific Amazon DocumentDB cluster.
`DBClusterIdentifier, Role`	Filters the data that you request for a specific Amazon DocumentDB cluster, aggregating the metric by instance role (WRITER/READER). For example, you can aggregate metrics for all READER instances that belong to a cluster.
`DBInstanceIdentifier`	Filters the data that you request for a specific database instance.

Monitoring opcounter metrics

Opcounter metrics have a non-zero value (usually ~50) for idle clusters. This is because Amazon DocumentDB performs periodic health checks, internal operations, and metrics collection tasks.

Monitoring database connections

When you view the number of connections by using database engine commands such as `db.runCommand( { serverStatus: 1 })`, you might see up to 10 more connections than you see in `DatabaseConnections` through CloudWatch. This occurs because Amazon DocumentDB performs periodic health checks and metrics collection tasks that don't get accounted for in `DatabaseConnections`. `DatabaseConnections` represents customer-initiated connections only.

Logging Amazon DocumentDB API calls with AWS CloudTrail

Amazon DocumentDB (with MongoDB compatibility) is integrated with AWS CloudTrail, a service that provides a record of actions taken by users, roles, or an AWS service in Amazon DocumentDB (with MongoDB compatibility). CloudTrail captures all AWS CLI API calls for Amazon DocumentDB as events, including calls from the Amazon DocumentDB console and from code calls to the Amazon DocumentDB SDK. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon DocumentDB. If you don't configure a trail, you can still view the most recent events on the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Amazon DocumentDB (with MongoDB compatibility), the IP address from which the request was made, who made the request, when it was made, and other details.

> **Important**
>
> For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS). Amazon DocumentDB console, AWS CLI, and API calls are logged as calls made to the Amazon RDS API.

To learn more about AWS CloudTrail, see AWS CloudTrail User Guide.

Amazon DocumentDB information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon DocumentDB (with MongoDB compatibility), that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for Amazon DocumentDB (with MongoDB compatibility), create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following topics in the *AWS CloudTrail User Guide*:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions
- Receiving CloudTrail Log Files from Multiple Accounts

Every event or log entry includes information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Profiling Amazon DocumentDB operations

You can use the profiler in Amazon DocumentDB (with MongoDB compatibility) to log the execution time and details of operations that were performed on your cluster. The profiler is useful for monitoring the slowest operations on your cluster to help you improve individual query performance and overall cluster performance.

By default, the profiler feature is disabled. When enabled, the profiler logs operations that are taking longer than a customer-defined threshold value (for example, 100 ms) to Amazon CloudWatch Logs. Logged details include the profiled command, time, plan summary, and client metadata. After the operations are logged to CloudWatch Logs, you can use CloudWatch Logs Insights to analyze, monitor, and archive your Amazon DocumentDB profiling data. Common queries are provided in the section Common queries.

When enabled, the profiler uses additional resources in your cluster. Start with a high threshold value (for example, 500 ms) and gradually lower the value to identify slow operations. Starting with a threshold value of 50 ms can cause performance issues on your cluster for high throughput applications. The profiler is enabled at the cluster level and works on all instances and databases in a cluster. Amazon DocumentDB logs operations to Amazon CloudWatch Logs on a best-effort basis.

Although Amazon DocumentDB imposes no additional charge to enable the profiler, you are charged the standard rates for the usage of CloudWatch Logs. For information about CloudWatch Logs pricing, see Amazon CloudWatch pricing.

Topics

- Supported operations
- Limitations
- Enabling the Amazon DocumentDB profiler
- Disabling the Amazon DocumentDB profiler
- Disabling profiler logs export
- Accessing your Amazon DocumentDB profiler logs
- Common queries

Supported operations

Amazon DocumentDB profiler supports the following operations:

- `aggregate`
- `count`
- `delete`
- `distinct`
- `find` (OP_QUERY and command)
- `findAndModify`
- `insert`
- `update`

Limitations

The slow query profiler is only able to emit profiler logs if the entire result set of the query is able to fit in one batch, and if the result set is under 16MB (maximum BSON size). Result sets greater than 16MB are automatically split into multiple batches.

Most drivers or shells might set a small default batch size. You can specify the batch size as part of your query. To capture slow query logs, use a batch size that exceeds the size of your expected

result set. If you are unsure of the result set size, or if it varies, set the batch size to a large number (for example, 100k).

However, using a larger batch size means more results will have to be retrieved from the database before a response is sent to the client. For some queries, that may create longer delays before you get results. If you do not plan to consume the entire result set, it is possible that you will spend more I/Os to process the query and throw away the result.

Enabling the Amazon DocumentDB profiler

Enabling the profiler on a cluster is a three-step process. Ensure that all steps are completed, or profiling logs will not be sent to CloudWatch Logs. Profiler is set at the cluster level and is performed on all of the cluster's databases and instances.

To enable the profiler on a cluster

1. Because you can't modify a default cluster parameter group, ensure that you have an available custom cluster parameter group. For more information, see Creating Amazon DocumentDB cluster parameter groups.
2. Using an available custom cluster parameter group, modify the following parameters: `profiler`, `profiler_threshold_ms`, and `profiler_sampling_rate`. For more information, see Modifying Amazon DocumentDB cluster parameter groups.
3. Create or modify your cluster to use the custom cluster parameter group and to enable exporting `profiler` logs to CloudWatch Logs.

The following sections show how to implement these steps using the AWS Management Console and the AWS Command Line Interface (AWS CLI).

Using the AWS Management Console

1. Before you begin, create a Amazon DocumentDB cluster and a custom cluster parameter group if you don't already have one. For more information, see Creating Amazon DocumentDB cluster parameter groups and Creating an Amazon DocumentDB cluster.
2. Using an available custom cluster parameter group, modify the following parameters. For more information, see Modifying Amazon DocumentDB cluster parameter groups.

- `profiler` — Enables or disables query profiling. Permitted values are `enabled` and `disabled`. The default value is `disabled`. To enable profiling, set the value to `enabled`.
- `profiler_threshold_ms` — When `profiler` is set to `enabled`, all commands that are taking longer than `profiler_threshold_ms` are logged to CloudWatch. Permitted values are `[50-INT_MAX]`. The default value is `100`.
- `profiler_sampling_rate` — The fraction of slow operations that should be profiled or logged. Permitted values are `[0.0-1.0]`. The default value is `1.0`.

3. Modify your cluster to use the custom cluster parameter group and set the profiler log exports to publish to Amazon CloudWatch.

   a. In the navigation pane, choose **Clusters** to add your custom parameter group to a cluster.

   b. Choose the button to the left of the name of the cluster that you want to associate your parameter group with. Select **Actions**, and then **Modify** to modify your cluster.

   c. Under **Cluster options**, choose the custom parameter group from the step above to add it to your cluster.

   d. Under **Log exports**, select **Profiler logs** to publish to Amazon CloudWatch.

   e. Choose **Continue** to view a summary of your modifications.

   f. After verifying your changes, you can apply them immediately or during the next maintenance window under **Scheduling of modifications** .

   g. Choose **Modify cluster** to update your cluster with your new parameter group.

Using the AWS CLI

The following procedure enables the profiler on all supported operations for the cluster `sample-cluster`.

1. Before you begin, ensure that you have an available custom cluster parameter group by running the following command, and reviewing the output for a cluster parameter group that doesn't have `default` in the name and has `docdb3.6` as the parameter group family. If you don't have a non-default cluster parameter group, see Creating Amazon DocumentDB cluster parameter groups.

```
aws docdb describe-db-cluster-parameter-groups \
```

```
    --query 'DBClusterParameterGroups[*].
[DBClusterParameterGroupName,DBParameterGroupFamily]'
```

In the following output, only `sample-parameter-group` meets both criteria.

```
[
       [
           "default.docdb3.6",
           "docdb3.6"
       ],
       [
           "sample-parameter-group",
           "docdb3.6"
       ]
]
```

2. Using your custom cluster parameter group, modify the following parameters:

   - `profiler` — Enables or disables query profiling. Permitted values are `enabled` and `disabled`. The default value is `disabled`. To enable profiling, set the value to `enabled`.
   - `profiler_threshold_ms` — When `profiler` is set to `enabled`, all commands taking longer than `profiler_threshold_ms` are logged to CloudWatch. Permitted values are `[50-INT_MAX]`. The default value is `100`.
   - `profiler_sampling_rate` — The fraction of slow operations that should be profiled or logged. Permitted values are `[0.0-1.0]`. The default value is `1.0`.

```
aws docdb modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name sample-parameter-group \
    --parameters
 ParameterName=profiler,ParameterValue=enabled,ApplyMethod=immediate \

 ParameterName=profiler_threshold_ms,ParameterValue=100,ApplyMethod=immediate \

 ParameterName=profiler_sampling_rate,ParameterValue=0.5,ApplyMethod=immediate
```

3. Modify your Amazon DocumentDB cluster so that it uses the `sample-parameter-group` custom cluster parameter group from the previous step and sets the parameter `--enable-cloudwatch-logs-exports` to `profiler`.

The following code modifies the cluster `sample-cluster` to use the `sample-parameter-group` from the previous step, and adds `profiler` to the enabled CloudWatch Logs exports.

```
aws docdb modify-db-cluster \
       --db-cluster-identifier sample-cluster \
       --db-cluster-parameter-group-name sample-parameter-group \
       --cloudwatch-logs-export-configuration '{"EnableLogTypes":["profiler"]}'
```

Output from this operation looks something like the following.

```
{
    "DBCluster": {
        "AvailabilityZones": [
            "us-east-1c",
            "us-east-1b",
            "us-east-1a"
        ],
        "BackupRetentionPeriod": 1,
        "DBClusterIdentifier": "sample-cluster",
        "DBClusterParameterGroup": "sample-parameter-group",
        "DBSubnetGroup": "default",
        "Status": "available",
        "EarliestRestorableTime": "2020-04-07T02:05:12.479Z",
        "Endpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "ReaderEndpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "MultiAZ": false,
        "Engine": "docdb",
        "EngineVersion": "3.6.0",
        "LatestRestorableTime": "2020-04-08T22:08:59.317Z",
        "Port": 27017,
        "MasterUsername": "test",
        "PreferredBackupWindow": "02:00-02:30",
        "PreferredMaintenanceWindow": "tue:09:50-tue:10:20",
        "DBClusterMembers": [
            {
                "DBInstanceIdentifier": "sample-instance-1",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
```

```
            },
            {
                "DBInstanceIdentifier": "sample-instance-2",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            }
        ],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcd0123",
                "Status": "active"
            }
        ],
        "HostedZoneId": "ABCDEFGHIJKLM",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DbClusterResourceId": "cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "DBClusterArn": "arn:aws:rds:us-east-1:<accountID>:cluster:sample-
cluster",
        "AssociatedRoles": [],
        "ClusterCreateTime": "2020-01-10T22:13:38.261Z",
        "EnabledCloudwatchLogsExports": [
            "profiler"
        ],
        "DeletionProtection": true
    }
}
```

Disabling the Amazon DocumentDB profiler

To disable the profiler, you disable both the `profiler` parameter and the export of `profiler` logs to CloudWatch Logs.

Disabling the Profiler

You can disable the `profiler` parameter using either the AWS Management Console or AWS CLI, as follows.

Using the AWS Management Console

The following procedure uses the AWS Management Console to disable Amazon DocumentDB `profiler`.

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Parameter groups**. Then choose the name of the cluster parameter group that you want to disable the profiler on.
3. In the resulting **Cluster parameters** page, select the button to the left of the `profiler` parameter and choose **Edit**.
4. In the **Modify profiler** dialog box, choose `disabled` in the list.
5. Choose **Modify cluster parameter**.

Using the AWS CLI

To disable `profiler` on a cluster using the AWS CLI, modify the cluster as follows.

```
aws docdb modify-db-cluster-parameter-group \
    --db-cluster-parameter-group-name sample-parameter-group \
    --parameters
 ParameterName=profiler,ParameterValue=disabled,ApplyMethod=immediate
```

Disabling profiler logs export

You can disable exporting `profiler` logs to CloudWatch Logs by using either the AWS Management Console or AWS CLI, as follows.

Using the AWS Management Console

The following procedure uses the AWS Management Console to disable Amazon DocumentDB exporting logs to CloudWatch.

1. Open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**. Choose the button to the left of the name of the cluster for which you want to disable exporting logs.
3. On the **Actions** menu, choose **Modify**.

4. Scroll down to the **Log exports** section and unselect **Profiler logs**.
5. Choose **Continue**.
6. Review your changes, and then choose when you want this change applied to your cluster:
   - **Apply during the next scheduled maintenance window**
   - **Apply immediately**
7. Choose **Modify cluster**.

Using the AWS CLI

The following code modifies the cluster `sample-cluster` and disables CloudWatch profiler logs.

Example

For Linux, macOS, or Unix:

```
aws docdb modify-db-cluster \
   --db-cluster-identifier sample-cluster \
   --cloudwatch-logs-export-configuration '{"DisableLogTypes":["profiler"]}'
```

For Windows:

```
aws docdb modify-db-cluster ^
   --db-cluster-identifier sample-cluster ^
   --cloudwatch-logs-export-configuration '{"DisableLogTypes":["profiler"]}'
```

Output from this operation looks something like the following.

```
{
    "DBCluster": {
        "AvailabilityZones": [
            "us-east-1c",
            "us-east-1b",
            "us-east-1a"
        ],
        "BackupRetentionPeriod": 1,
        "DBClusterIdentifier": "sample-cluster",
        "DBClusterParameterGroup": "sample-parameter-group",
        "DBSubnetGroup": "default",
        "Status": "available",
```

```
        "EarliestRestorableTime": "2020-04-08T02:05:17.266Z",
        "Endpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "ReaderEndpoint": "sample-cluster.node.us-east-1.docdb.amazonaws.com",
        "MultiAZ": false,
        "Engine": "docdb",
        "EngineVersion": "3.6.0",
        "LatestRestorableTime": "2020-04-09T05:14:44.356Z",
        "Port": 27017,
        "MasterUsername": "test",
        "PreferredBackupWindow": "02:00-02:30",
        "PreferredMaintenanceWindow": "tue:09:50-tue:10:20",
        "DBClusterMembers": [
            {
                "DBInstanceIdentifier": "sample-instance-1",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            },
            {
                "DBInstanceIdentifier": "sample-instance-2",
                "IsClusterWriter": true,
                "DBClusterParameterGroupStatus": "in-sync",
                "PromotionTier": 1
            }
        ],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-abcd0123",
                "Status": "active"
            }
        ],
        "HostedZoneId": "ABCDEFGHIJKLM",
        "StorageEncrypted": true,
        "KmsKeyId": "arn:aws:kms:us-east-1:<accountID>:key/sample-key",
        "DbClusterResourceId": "cluster-ABCDEFGHIJKLMNOPQRSTUVWXYZ",
        "DBClusterArn": "arn:aws:rds:us-east-1:<accountID>:cluster:sample-cluster",
        "AssociatedRoles": [],
        "ClusterCreateTime": "2020-01-10T22:13:38.261Z",
        "DeletionProtection": true
    }
}
```

Accessing your Amazon DocumentDB profiler logs

Follow these steps to access your profile logs on Amazon CloudWatch.

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Make sure that you are in the same Region as your Amazon DocumentDB cluster.
3. In the navigation pane, choose **Logs**.
4. To find the profiler logs for your cluster, in the list, choose `/aws/docdb/`*`yourClusterName`*`/profiler`.

   The profile logs for each of your instances are available under each of the respective instance names.

Common queries

The following are some common queries you can use to analyze your profiled commands. For more information about CloudWatch Logs Insights, see Analyzing Log Data with CloudWatch Logs Insights and Sample Queries.

Get the 10 slowest operations on a specified collection

```
filter ns="test.foo" | sort millis desc | limit 10
```

Get all the update operations on a collection that took more than 60 ms

```
filter millis > 60 and op = "update"
```

Get the 10 slowest operations in the last month

```
sort millis desc | limit 10
```

Get all the queries with a COLLSCAN plan summary

```
filter planSummary="COLLSCAN"
```

Monitoring with Performance Insights

Performance Insights adds to the existing Amazon DocumentDB monitoring features to illustrate your cluster performance and help you analyze any issues that affect it. With the Performance Insights dashboard, you can visualize the database load and filter the load by waits, query statements, hosts, or application.

> **Note**
>
> Performance Insights is only available for Amazon DocumentDB 3.6, 4.0, 5.0, and 8.0 instance-based clusters.

How is it useful?

- Visualize database performance — Visualize the load to determine when and where the load is on the database
- Determine what is causing load on database — Determine which queries, hosts, and applications are contributing to the load on your instance
- Determine when there is load on your database — Zoom in on the Performance Insights dashboard to focus on specific events or zoom out to look at trends across a larger time span
- Alert on database load — Access new database load metrics automatically from CloudWatch where you can monitor the DB load metrics alongside other Amazon DocumentDB metrics and set alerts on them

What are the limitations of Amazon DocumentDB Performance Insights?

- Performance Insights in the AWS GovCloud (US-East) and AWS GovCloud (US-West) Regions are not available
- Performance Insights for Amazon DocumentDB retains up to 7 days of performance data
- Queries longer than 1,024 bytes are not aggregated in Performance Insights

Topics

- Performance Insights concepts
- Enabling and disabling Performance Insights

- Configuring access policies for Performance Insights
- Analyzing metrics with the Performance Insights dashboard
- Retrieving metrics with the Performance Insights API
- Amazon CloudWatch metrics for Performance Insights
- Performance Insights for counter metrics

Performance Insights concepts

Topics

- Average active sessions
- Dimensions
- Max vCPU

Average active sessions

Database load (DB load) measures the level of activity in your database. The key metric in Performance Insights is DB Load, which is collected every second. The unit for the DBLoad metric is the *Average Active Sessions (AAS)* for an Amazon DocumentDB instance.

An *active* session is a connection that has submitted work to the Amazon DocumentDB instance and is waiting for a response. For example, if you submit a query to an Amazon DocumentDB instance, the database session is active while the instance is processing the query.

To obtain the average active sessions, Performance Insights samples the number of sessions concurrently running a query. The AAS is the total number of sessions divided by the total number of samples. The following table shows five consecutive samples of a running query.

Sample	Number of sessions running query	AAS	Calculation
1	2	2	2 sessions / 1 sample
2	0	1	2 sessions / 2 samples

Sample	Number of sessions running query	AAS	Calculation
3	4	2	6 sessions / 3 samples
4	0	1.5	6 sessions / 4 samples
5	4	2	10 sessions / 5 samples

In the preceding example, the DB Load for the time interval from 1-5 is 2 AAS. An increase in DB load means that, on average, more sessions are running on the database.

Dimensions

The DB Load metric is different from the other time-series metrics because you can break it into subcomponents called dimensions. You can think of dimensions as categories for the different characteristics of the DB Load metric. When you are diagnosing performance issues, the most useful dimensions are **wait states** and **top query**.

wait states

A *wait state* causes a query statement to wait for a specific event to happen before it can continue running. For example, a query statement might wait until a locked resource is unlocked. By combining DB Load with wait states, you can get a complete picture of the session state. Here are various Amazon DocumentDB wait states:

Amazon DocumentDB wait state	Wait State Description
Latch	The Latch wait state occurs when the session is waiting to page the buffer pool. Frequent paging in and out of the buffer pool can happen more often when there are frequent large queries being processed by the system, collection scans, or when the buffer pool is too small to handle the working set.

Amazon DocumentDB wait state	Wait State Description
CPU	The CPU wait state occurs when the session is waiting on CPU.
CollectionLock	The CollectionLock wait state occurs when the session is waiting to acquire a lock on the collection. These events occur when there are DDL operations on the collection.
DocumentLock	The DocumentLock wait state occurs when the session is waiting to acquire a lock on a document. High number of concurrent writes to the same document will contribute to more DocumentLock wait states on that document.
SystemLock	The SystemLock wait state occurs when the session is waiting on the system. This can occur when there are frequent long running queries, long running transactions, or high concurrency on the system.
IO	The IO wait state occurs when the session waiting on IO to complete.
BufferLock	The BufferLock wait state occurs when the session is waiting to acquire a lock on a shared page in the buffer. BufferLock wait states can be prolonged if other processes are holding open cursors on the requested pages.
LowMemThrottle	The LowMemThrottle wait state occurs when the session is waiting due to heavy memory pressure on the Amazon DocumentDB instance. If this state persists for a long time, consider scaling up the instance to provide additional memory. For more information, see Resource Governor.

Amazon DocumentDB wait state	Wait State Description
BackgroundActivity	The BackgroundActivity wait state occurs when the session is waiting on internal system processes.
Other	The Other wait state is an internal wait state. If this state persists for a long time, consider terminating this query. For more informati on, see How Do I Find and Terminate Long Running or Blocked Queries?

Top queries

Whereas wait states show bottlenecks, top queries show which queries are contributing the most to DB load. For example, many queries might be currently running on the database, but a single query might consume 99% of the DB load. In this case, the high load might indicate a problem with the query.

Max vCPU

In the dashboard, the **Database load** chart collects, aggregates, and displays session information. To see whether active sessions are exceeding the maximum CPU, look at their relationship to the **Max vCPU** line. The **Max vCPU** value is determined by the number of vCPU (virtual CPU) cores for your Amazon DocumentDB instance.

If the DB load is often above the **Max vCPU** line, and the primary wait state is CPU, the CPU is overloaded. In this case, you might want to throttle connections to the instance, tune any queries with a high CPU load, or consider a larger instance class. High and consistent instances of any wait state indicate that there might be bottlenecks or resource contention issues to resolve. This can be true even if the DB load doesn't cross the **Max vCPU** line.

Enabling and disabling Performance Insights

To use Performance Insights, enable it on your DB instance. You can disable it later if necessary. Enabling and disabling Performance Insights doesn't cause downtime, a reboot, or a failover.

The Performance Insights agent consumes limited CPU and memory on the DB host. When the DB load is high, the agent limits the performance impact by collecting data less frequently.

Enabling Performance Insights when creating a cluster

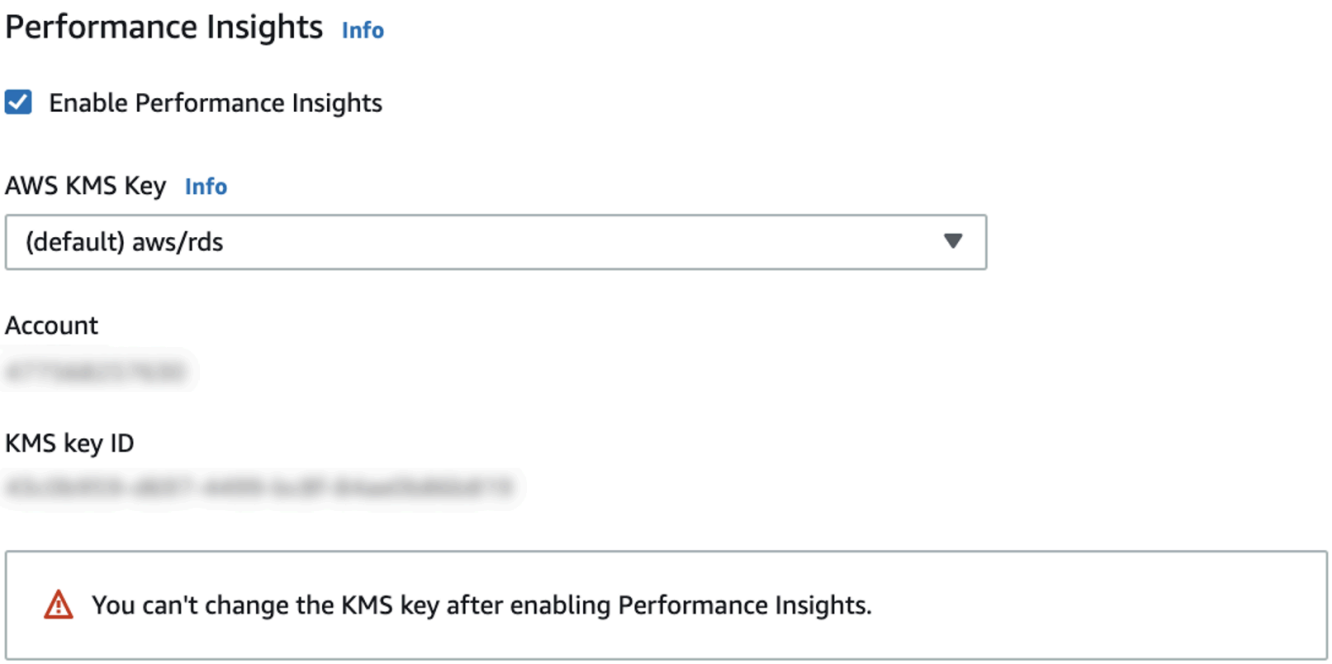
In the console, you can enable or disable Performance Insights when you create or modify a new DB instance.

Using the AWS Management Console

In the console, you can enable Performance Insights when you create an Amazon DocumentDB cluster. When you create a new Amazon DocumentDB cluster, enable Performance Insights by choosing **Enable Performance Insights** in the **Performance Insights** section.

Console instructions

1. To create a cluster, follow the instructions for Creating an Amazon DocumentDB cluster.
2. Select **Enable Performance Insights** in the Performance Insights section.



> **Note**
>
> The Performance Insights data retention period will be seven days.

AWS KMS key — Specify your AWS KMS key. Performance Insights encrypts all potentially sensitive data using your AWS KMS key. Data is encrypted in flight and at rest. For more information, see Configuring an AWS AWS KMS policy for Performance Insights.

Enabling and disabling when modifying an instance

You can modify a DB instance to enable or disable Performance Insights using the console or AWS CLI.

Using the AWS Management Console

Console instructions

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. Choose **Clusters**.
3. Choose a DB instance, and choose **Modify**.
4. In the Performance Insights section, choose either **Enable Performance Insights** or **Disable Performance Insights**.

   > **Note**
   >
   > If you choose **Enable Performance Insights**, you can specify your AWS AWS KMS key. Performance Insights encrypts all potentially sensitive data using your AWS KMS key. Data is encrypted in flight and at rest. For more information, see Encrypting Amazon DocumentDB Data at Rest.

5. Choose **Continue**.
6. For **Scheduling of Modifications**, choose **Apply immediately**. If you choose **Apply during the next scheduled maintenance window**, your instance ignores this setting and enables Performance Insights immediately.
7. Choose **Modify instance**.

Using the AWS CLI

When you use the `create-db-instance` or `modify-db-instance` AWS AWS CLI commands, you can enable Performance Insights by specifying `--enable-performance-insights`, or disable it by specifying `--no-enable-performance-insights`.

The following procedure describes how to enable or disable Performance Insights for a DB instance using the AWS AWS CLI.

AWS AWS CLI instructions

Call the `modify-db-instance` AWS AWS CLI command and provide the following values:

- `--db-instance-identifer` — The name of the DB instance
- `--enable-performance-insights` to enable or `--no-enable-performance-insights` to disable

Example

The following example enables Performance Insights for `sample-db-instance`:

For Linux, macOS, or Unix:

```
aws docdb modify-db-instance \
    --db-instance-identifier sample-db-instance \
    --enable-performance-insights
```

For Windows:

```
aws docdb modify-db-instance ^
    --db-instance-identifier sample-db-instance ^
    --enable-performance-insights
```

Configuring access policies for Performance Insights

To access Performance Insights, you must have the appropriate permissions from AWS Identity and Access Management (IAM). You have the following options for granting access:

- Attach the `AmazonRDSPerformanceInsightsReadOnly` managed policy to a permission set or role.
- Create a custom IAM policy and attach it to a permission set or role.

Also, if you specified a customer managed key when you turned on Performance Insights, make sure that users in your account have the `kms:Decrypt` and `kms:GenerateDataKey` permissions on the KMS key.

> **Note**
>
> For encryption-at-rest with AWS KMS keys and security groups management, Amazon DocumentDB leverages operational technology that is shared with Amazon RDS.

Attaching the AmazonRDSPerformanceInsightsReadOnly policy to an IAM principal

`AmazonRDSPerformanceInsightsReadOnly` is an AWS-managed policy that grants access to all read-only operations of the Amazon DocumentDB Performance Insights API. Currently, all operations in this API are read-only. If you attach `AmazonRDSPerformanceInsightsReadOnly` to a permission set or role, the recipient can use Performance Insights with other console features.

Creating a custom IAM policy for Performance Insights

For users who don't have the `AmazonRDSPerformanceInsightsReadOnly` policy, you can grant access to Performance Insights by creating or modifying a user-managed IAM policy. When you attach the policy to a permission set or role, the recipient can use Performance Insights.

To create a custom policy

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Policies**.
3. Choose **Create policy**.
4. On the **Create Policy** page, choose the JSON tab.
5. Copy and paste the following text, replacing *`us-east-1`* with the name of your AWS Region and *`111122223333`* with your customer account number.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "rds:DescribeDBInstances",
```

```
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "rds:DescribeDBClusters",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:DescribeDimensionKeys",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:GetDimensionKeyDetails",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:GetResourceMetadata",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:GetResourceMetrics",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:ListAvailableResourceDimensions",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        },
        {
            "Effect": "Allow",
            "Action": "pi:ListAvailableResourceMetrics",
            "Resource": "arn:aws:pi:us-east-1:111122223333:metrics/rds/*"
        }
    ]
}
```

6. Choose **Review policy**.
7. Provide a name for the policy and optionally a description, and then choose **Create policy**.

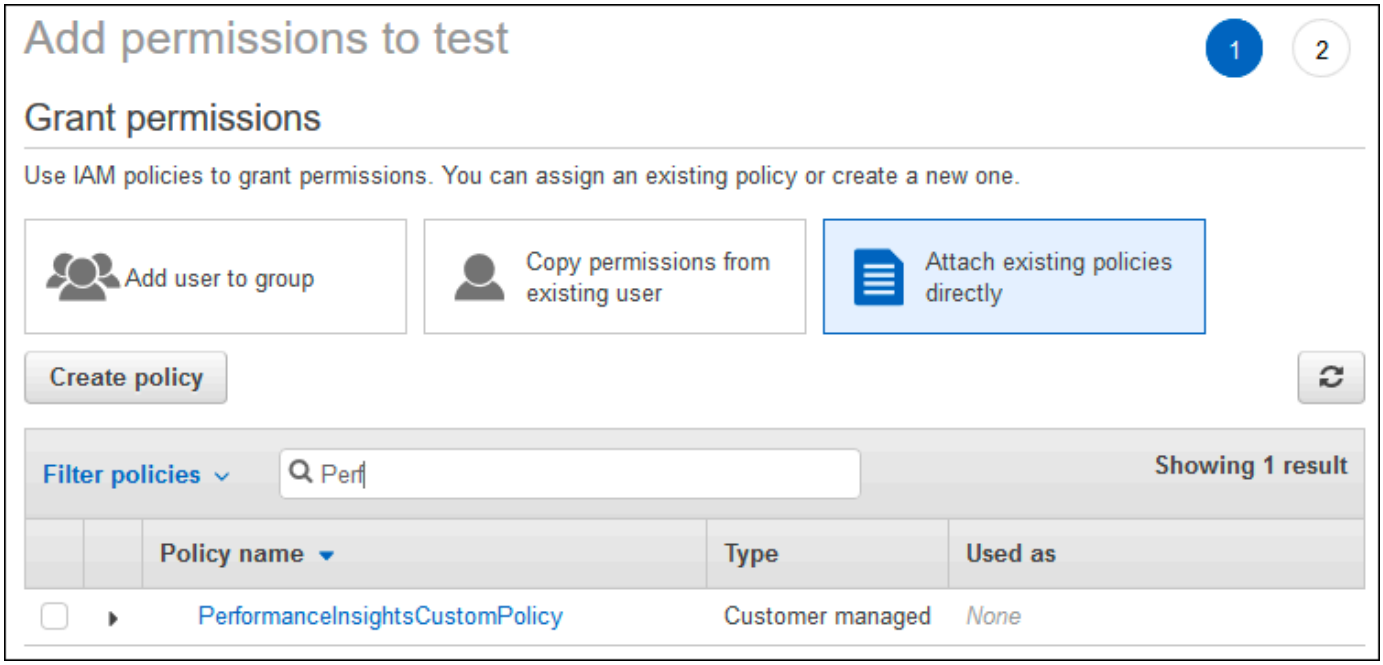
You can now attach the policy to a permission set or role. The following procedure assumes that you already have a user available for this purpose.

To attach the policy to a user

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane, choose **Users**.
3. Choose an existing user from the list.

> **Important**
>
> To use Performance Insights, make sure that you have access to Amazon DocumentDB in addition to the custom policy. For example, the **AmazonDocDBReadOnlyAccess** predefined policy provides read-only access to Amazon DocDB.For more information, see Managing access using policies.

4. On the **Summary** page, choose **Add permissions**.
5. Choose **Attach existing policies directly**. For **Search**, type the first few characters of your policy name, as shown following.



6. Choose your policy, and then choose **Next: Review**.
7. Choose **Add permissions**.

Configuring an AWS KMS policy for Performance Insights

Performance Insights uses an AWS KMS key to encrypt sensitive data. When you enable Performance Insights through the API or the console, you have the following options:

- Choose the default AWS managed key.

  Amazon DocumentDB uses the AWS managed key for your new DB instance. Amazon DocumentDB creates an AWS managed key for your AWS account. Your AWS account has a different AWS managed key for Amazon DocumentDB for each AWS Region.
- Choose a customer managed key.

  If you specify a customer managed key, users in your account that call the Performance Insights API need the `kms:Decrypt` and `kms:GenerateDataKey` permissions on the KMS key. Manage the permissions of the KMS key using the AWS KMS key policy instead of using IAM policies. For more information, see Using key policies in AWS KMS.

Example

The following sample key policy shows how to add statements to your KMS key policy. These statements allow access to Performance Insights. Depending on how you use the AWS KMS, you might want to change some restrictions. Before adding statements to your policy, remove all comments.

Analyzing metrics with the Performance Insights dashboard

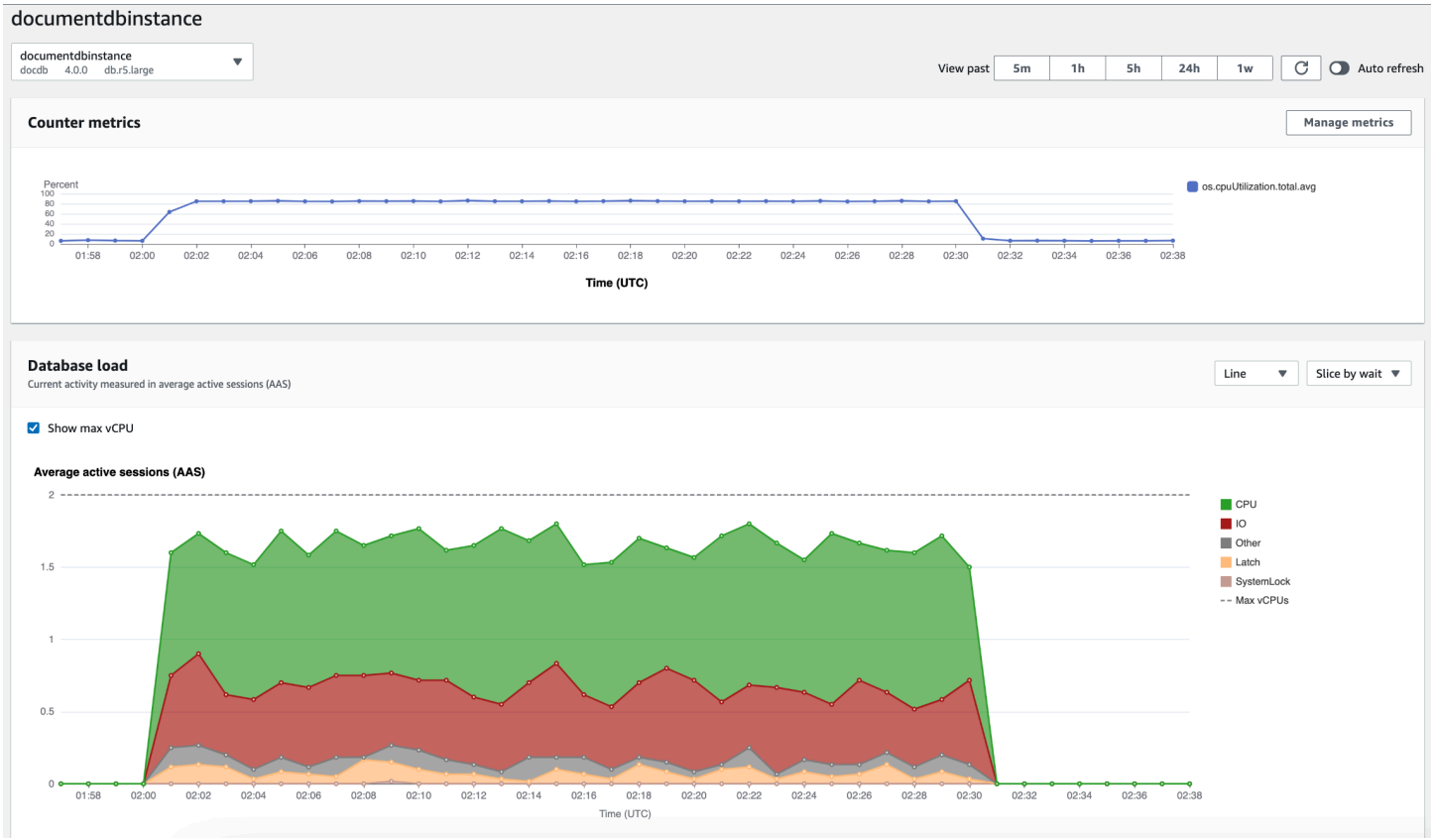
The Performance Insights dashboard contains database performance information to help you analyze and troubleshoot performance issues. On the main dashboard page, you can view information about the database load (DB load). You can "slice" DB load by dimensions such as wait states or query.

Topics

- Overview of the Performance Insights dashboard
- Opening the Performance Insights dashboard
- Analyzing database load by wait states
- Overview of the Top queries tab
- Zooming in on the database load chart

Overview of the Performance Insights dashboard

The dashboard is the easiest way to interact with Performance Insights. The following example shows the dashboard for an Amazon DocumentDB instance. By default, the Performance Insights dashboard shows data for the last hour.



The dashboard is divided into the following parts:

1. **Counter metrics** – Shows data for specific performance counter metrics.
2. **Database load** – Shows how the DB load compares to DB instance capacity as represented by the **Max vCPU** line.
3. **Top dimensions** – Shows the top dimensions contributing to DB load. These dimensions include `waits`, `queries`, `hosts`, `databases`, and `applications`.

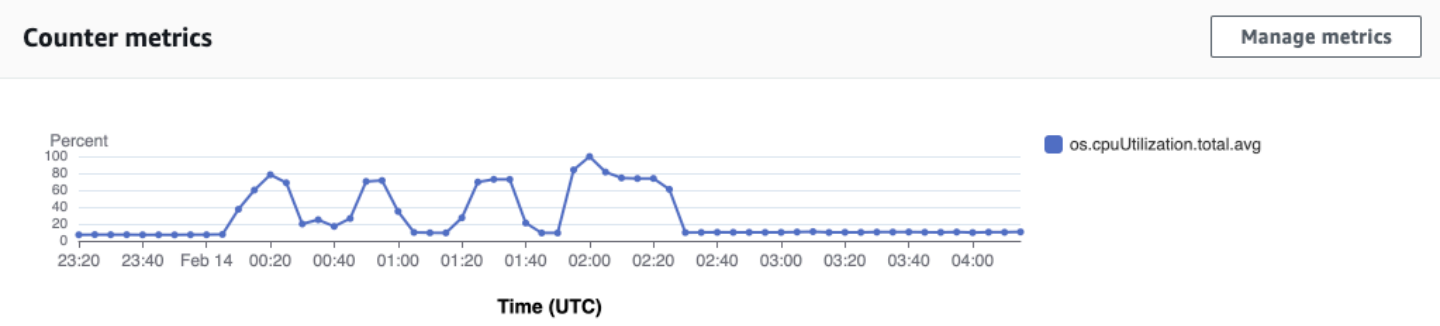
Topics

- Counter metrics chart
- Database load chart
- Top dimensions table

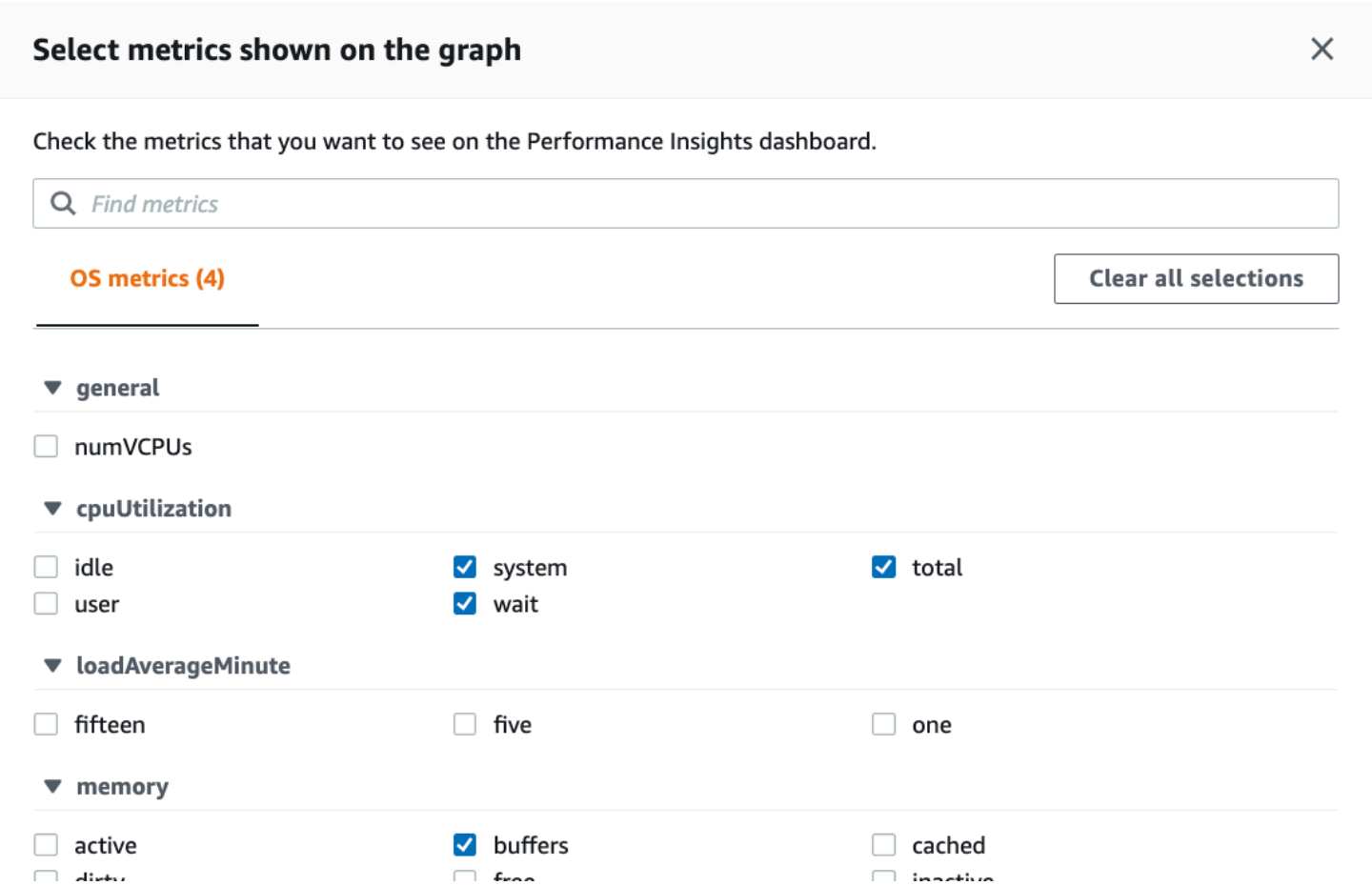
Counter metrics chart

With counter metrics, you can customize the Performance Insights dashboard to include up to 10 additional graphs. These graphs show a selection of dozens of operating system metrics. You can correlate this information with DB load to help identify and analyze performance problems.

The **Counter metrics** chart displays data for performance counters.



To change the performance counters, choose **Manage metrics**. You can select multiple **OS metrics** as shown in the following screenshot. To see details for any metric, hover over the metric name.



Database load chart

The **Database load** chart shows how the database activity compares to instance capacity as represented by the **Max vCPU** line. By default, the stacked line chart represents DB load as average active sessions per unit of time. The DB load is sliced (grouped) by wait states.



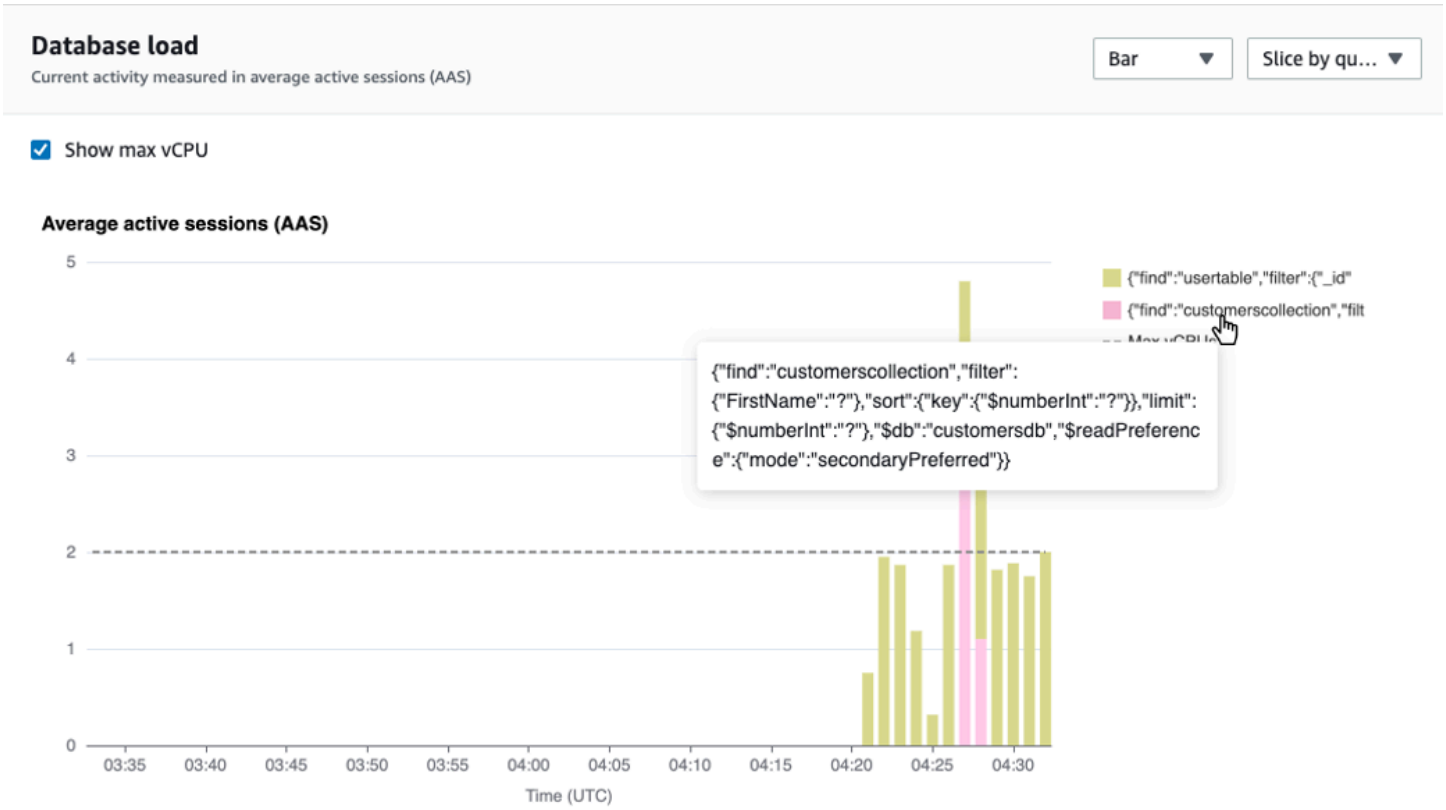
DB load sliced by dimensions

You can choose to display load as active sessions grouped by any supported dimensions. The following image shows the dimensions for the Amazon DocumentDB instance.

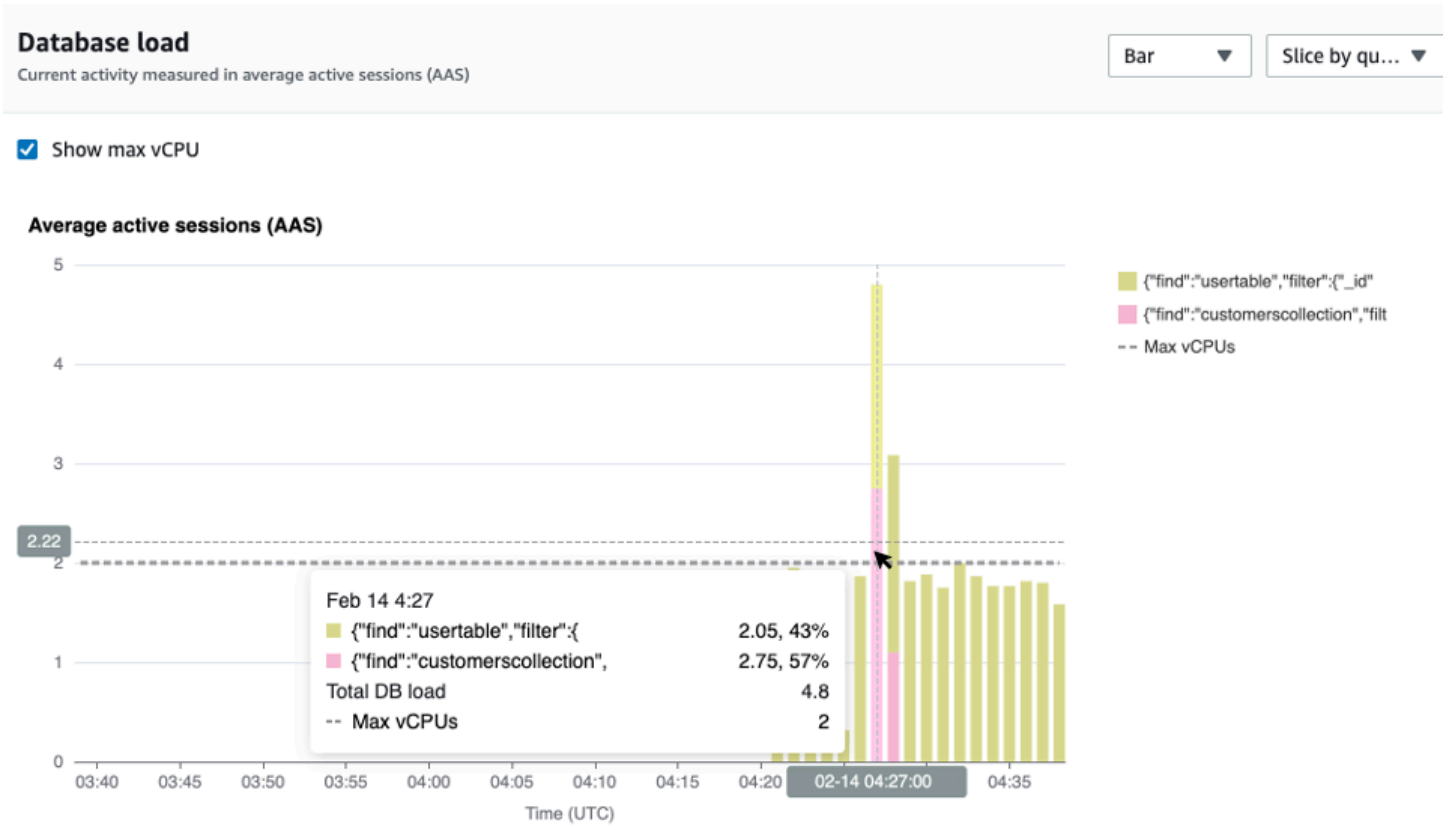


DB load details for a dimension item

To see details about a DB load item within a dimension, hover over the item name. The following image shows details for a query statement.



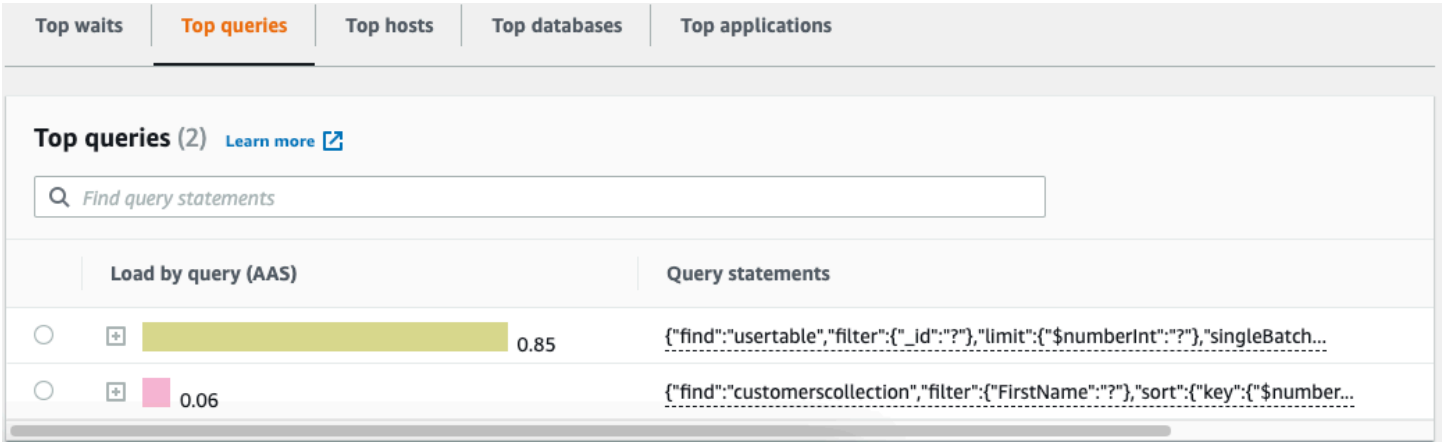
To see details for any item for the selected time period in the legend, hover over that item.



Top dimensions table

The **Top dimensions table** slices DB load by different dimensions. A dimension is a category or "slice by" for different characteristics of DB load. If the dimension is query, **Top queries** shows the query statements that contribute most to the DB load.

Choose any of the following dimension tabs.



The following table provides a brief description of each tab.

Tab	Description
Top waits	The waits for which the database backend is waiting
Top queries	The statements that are

Dimension	Description
	currently running
The host	The IP and port of the connected client
The database	The databases of the database to which the client is connected

Tab	Description
Top applications	The application that is connected to the database

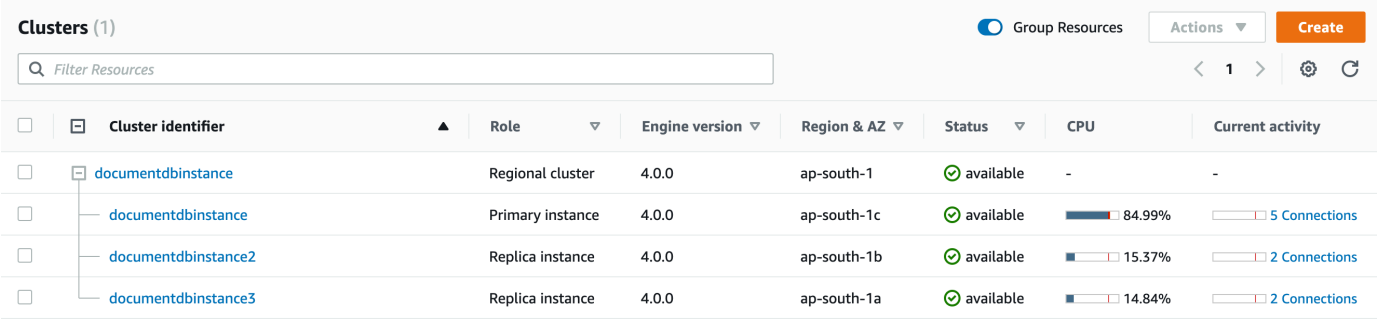
To learn how to analyze queries by using the **Top queries** tab, see Overview of the Top queries tab.

Opening the Performance Insights dashboard

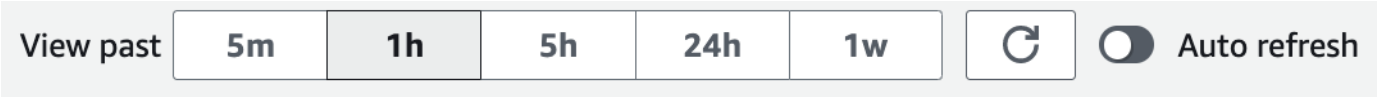
To view the Performance Insights dashboard in the AWS Management Console, use the following steps:

1. Open the Performance Insights console at https://console.aws.amazon.com/docdb/.
2. Choose a DB instance. The Performance Insights dashboard is shown for that Amazon DocumentDB instance.

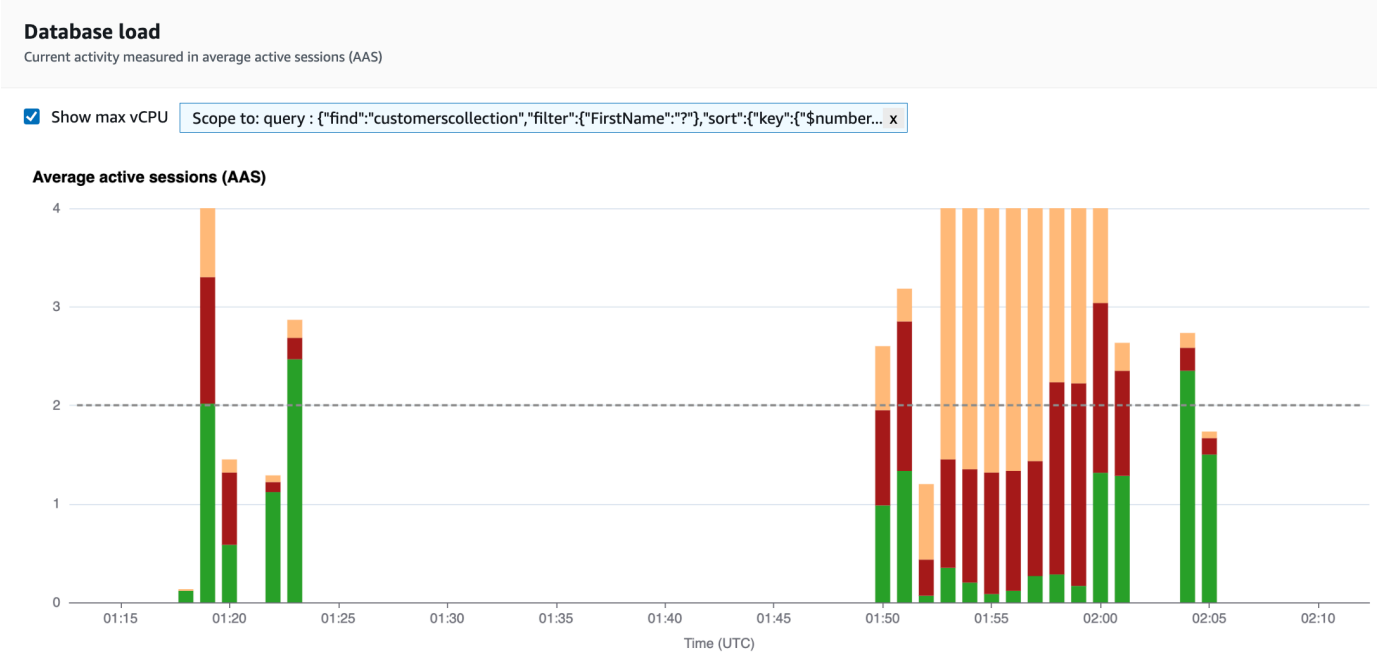
   For Amazon DocumentDB instances with Performance Insights enabled, you can also reach the dashboard by choosing the **Sessions** item in the list of instances. Under **Current activity**, the **Sessions** item shows the database load in average active sessions over the last five minutes. The bar graphically shows the load. When the bar is empty, the instance is idle. As the load increases, the bar fills with blue. When the load passes the number of virtual CPUs (vCPUs) on the instance class, the bar turns red, indicating a potential bottleneck.



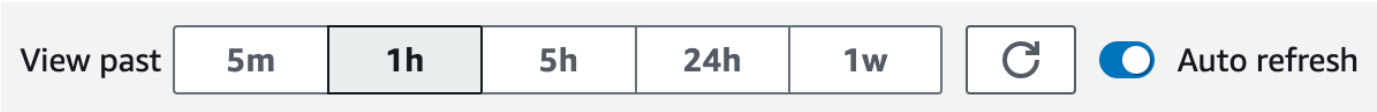
3. (Optional) Choose a different time interval by selecting a button in the upper right. For example, to change the interval to 1 hour, select **1h**.



In the following screenshot, the DB load interval is 1 hour.



4. To refresh your data automatically, enable **Auto refresh**.



The Performance Insight dashboard automatically refreshes with new data. The refresh rate depends on the amount of data displayed:

- 5 minutes refreshes every 5 seconds.

- 1 hour refreshes every minute.
- 5 hours refreshes every minute.
- 24 hours refreshes every 5 minutes.
- 1 week refreshes every hour.

Analyzing database load by wait states

If the **Database load (DB load)** chart shows a bottleneck, you can find out where the load is coming from. To do so, look at the top load items table below the **Database load** chart. Choose a particular item, like a query or an application, to drill down into that item and see details about it.

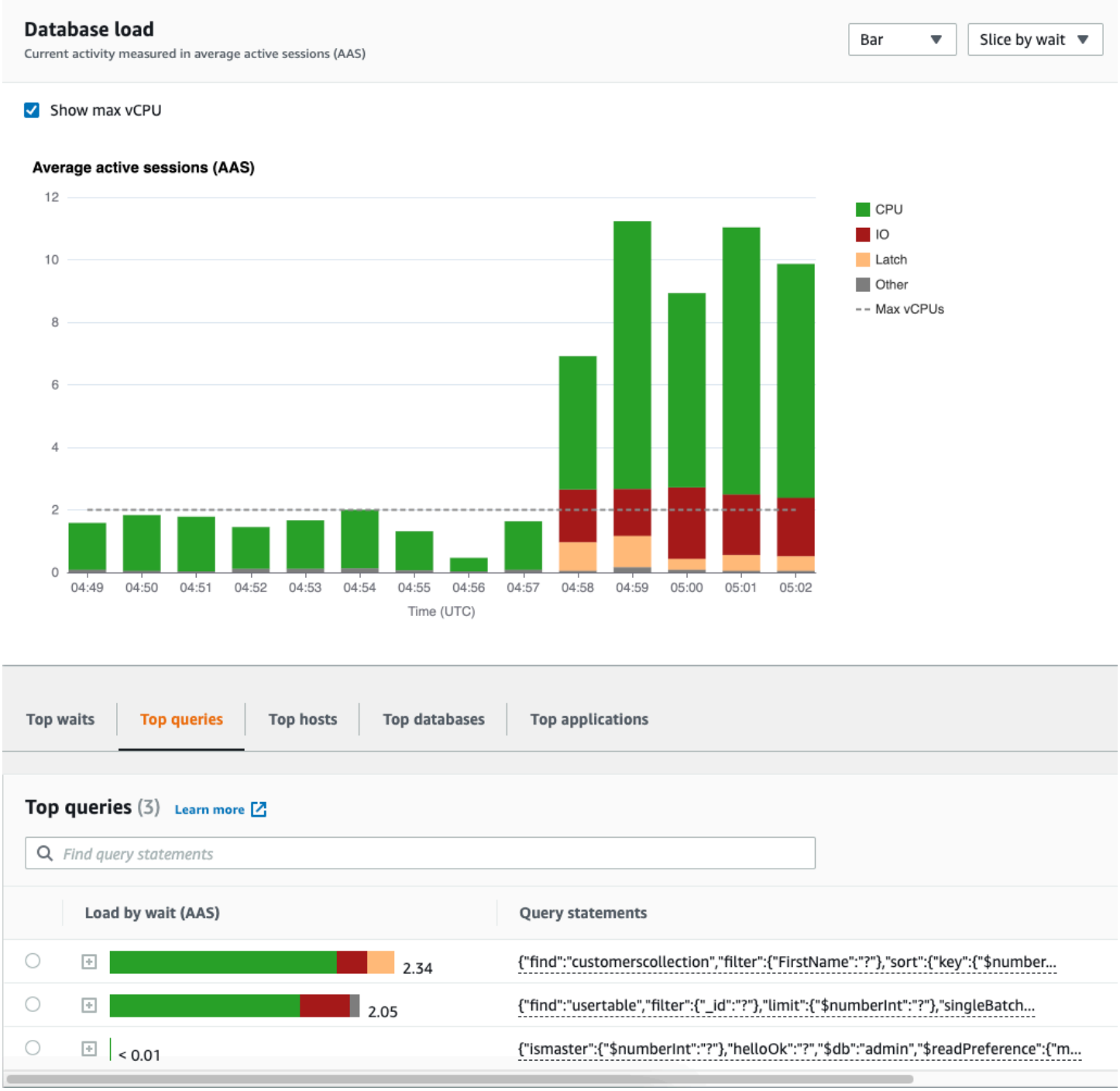
DB load grouped by waits and top queries typically provides the most insight into performance issues. DB load grouped by waits shows if there are any resource or concurrency bottlenecks in the database. In this case, the **Top queries** tab of the top load items table shows which queries are driving that load.

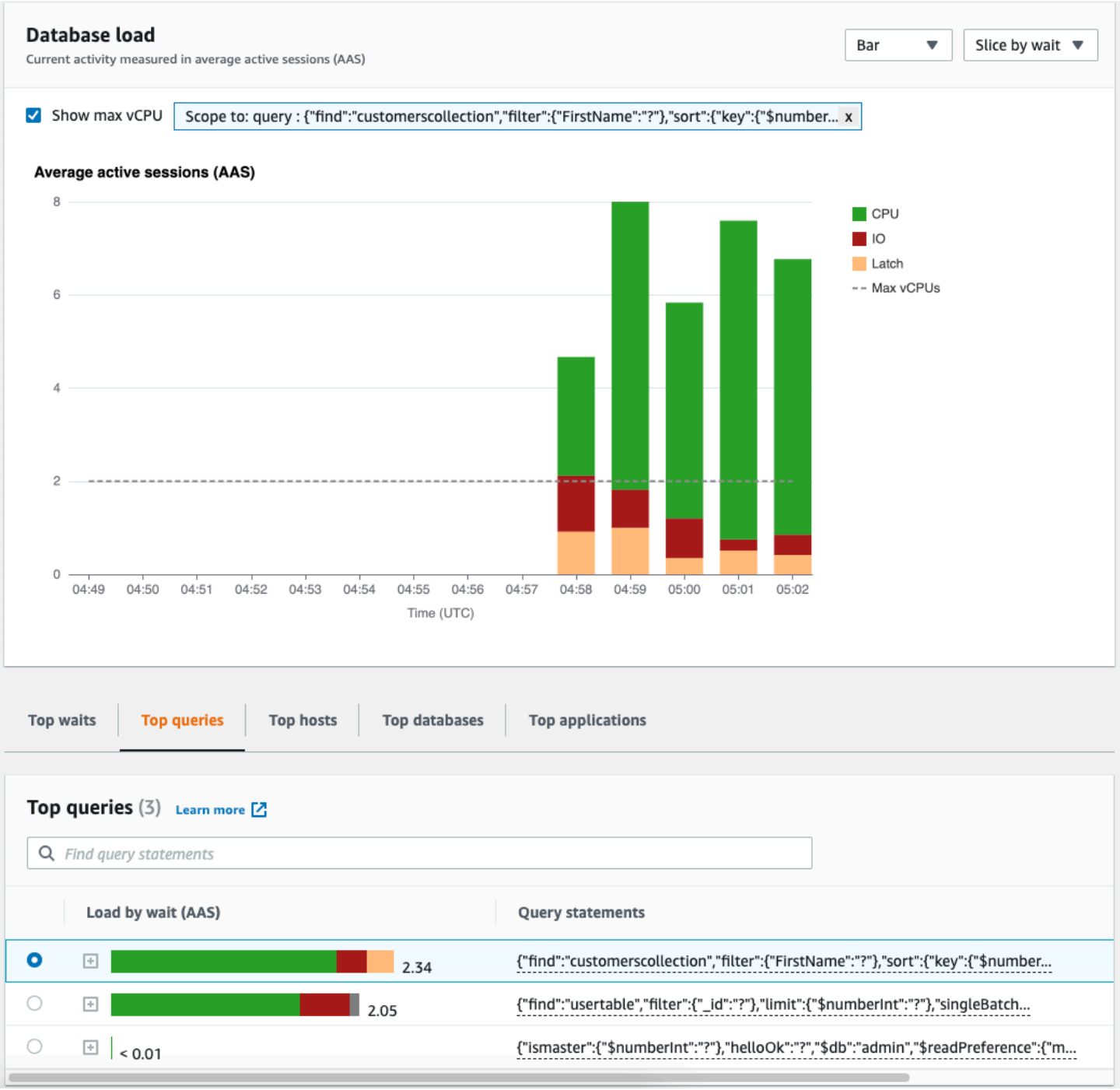
Your typical workflow for diagnosing performance issues is as follows:

1. Review the **Database load** chart and see if there are any incidents of database load exceeding the **Max CPU** line.
2. If there is, look at the **Database load** chart and identify which wait state or states are primarily responsible.
3. Identify the digest queries causing the load by seeing which of the queries the **Top queries** tab on the top load items table are contributing most to those wait states. You can identify these by the **Load by Wait (AAS)** column.
4. Choose one of these digest queries in the **Top queries** tab to expand it and see the child queries that it is composed of.

You can also see which hosts or applications are contributing the most load by selecting **Top hosts** or **Top applications**, respectively. Application names are specified in the connection string to the Amazon DocumentDB instance. Unknown indicates that the application field was not specified.

For example, in the following dashboard, **CPU** waits account for most of the DB load. Selecting the top query under **Top queries** will scope the Database load chart to focus on the most load that is being contributed by the select query.


Database load
Current activity measured in average active sessions (AAS)
Bar
Slice by wait
Show max vCPU
Average active sessions (AAS)
12
10
8
6
4
2
0
04:49
04:50
04:51
04:52
04:53
04:54
04:55
04:56
04:57
04:58
04:59
05:00
05:01
05:02
Time (UTC)
CPU
IO
Latch
Other
Max vCPUs
Top waits
Top queries
Top hosts
Top databases
Top applications
Top queries (3)
Learn more
Find query statements
Load by wait (AAS)
Query statements
2.34
{"find":"customerscollection","filter":{"FirstName":"?"},"sort":{"key":{"$number...
2.05
{"find":"usertable","filter":{"_id":"?"},"limit":{"$numberInt":"?"},"singleBatch...
< 0.01
{"ismaster":{"$numberInt":"?"},"helloOk":"?","$db":"admin","$readPreference":{"m...



Overview of the Top queries tab

By default, the **Top query** tab shows the queries that are contributing the most to DB load. You can analyze the query text to help tune your queries.

Topics

- Query digests

- Load by waits (AAS)
- Viewing detailed query information
- Accessing statement query text
- Viewing and downloading statement query text

Query digests

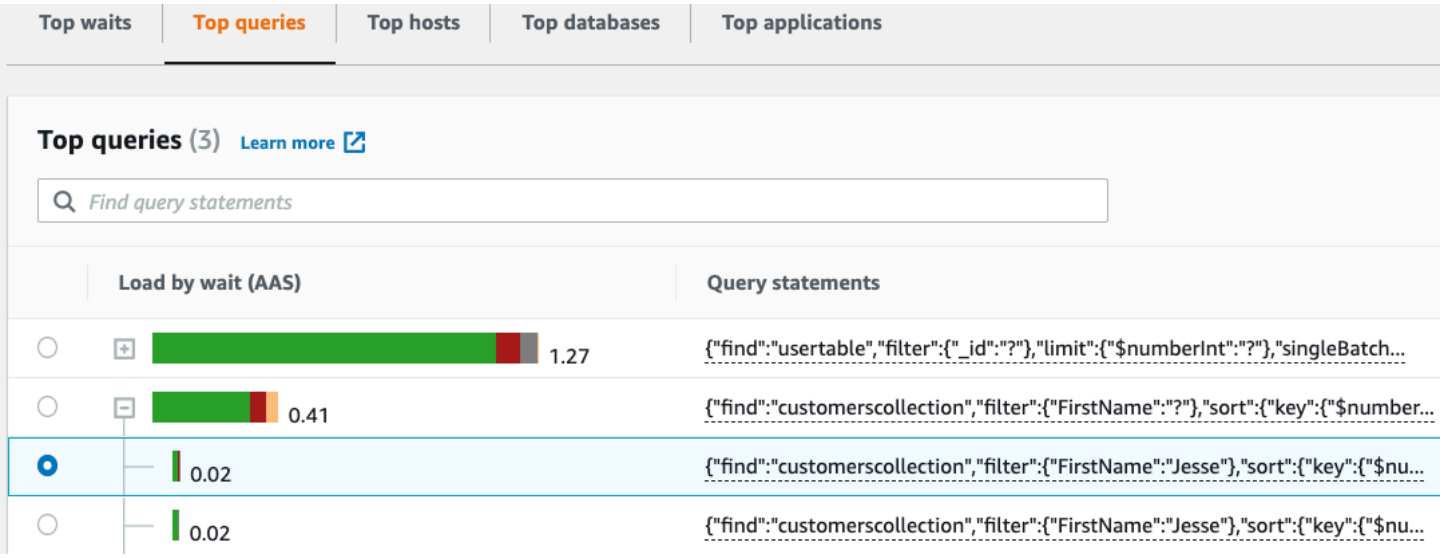
A *query digest* is a composite of multiple actual queries that are structurally similar but might have different literal values. The digest replaces hardcoded values with a question mark. For example, a query digest might look like this:

```
{"find":"customerscollection","filter":{"FirstName":"?"},"sort":{"key":
{"$numberInt":"?"}},"limit":{"$numberInt":"?"}}
```

This digest might include the following child queries:

```
{"find":"customerscollection","filter":{"FirstName":"Karrie"},"sort":{"key":
{"$numberInt":"1"}},"limit":{"$numberInt":"3"}}
{"find":"customerscollection","filter":{"FirstName":"Met"},"sort":{"key":
{"$numberInt":"1"}},"limit":{"$numberInt":"3"}}
{"find":"customerscollection","filter":{"FirstName":"Rashin"},"sort":{"key":
{"$numberInt":"1"}},"limit":{"$numberInt":"3"}}
```

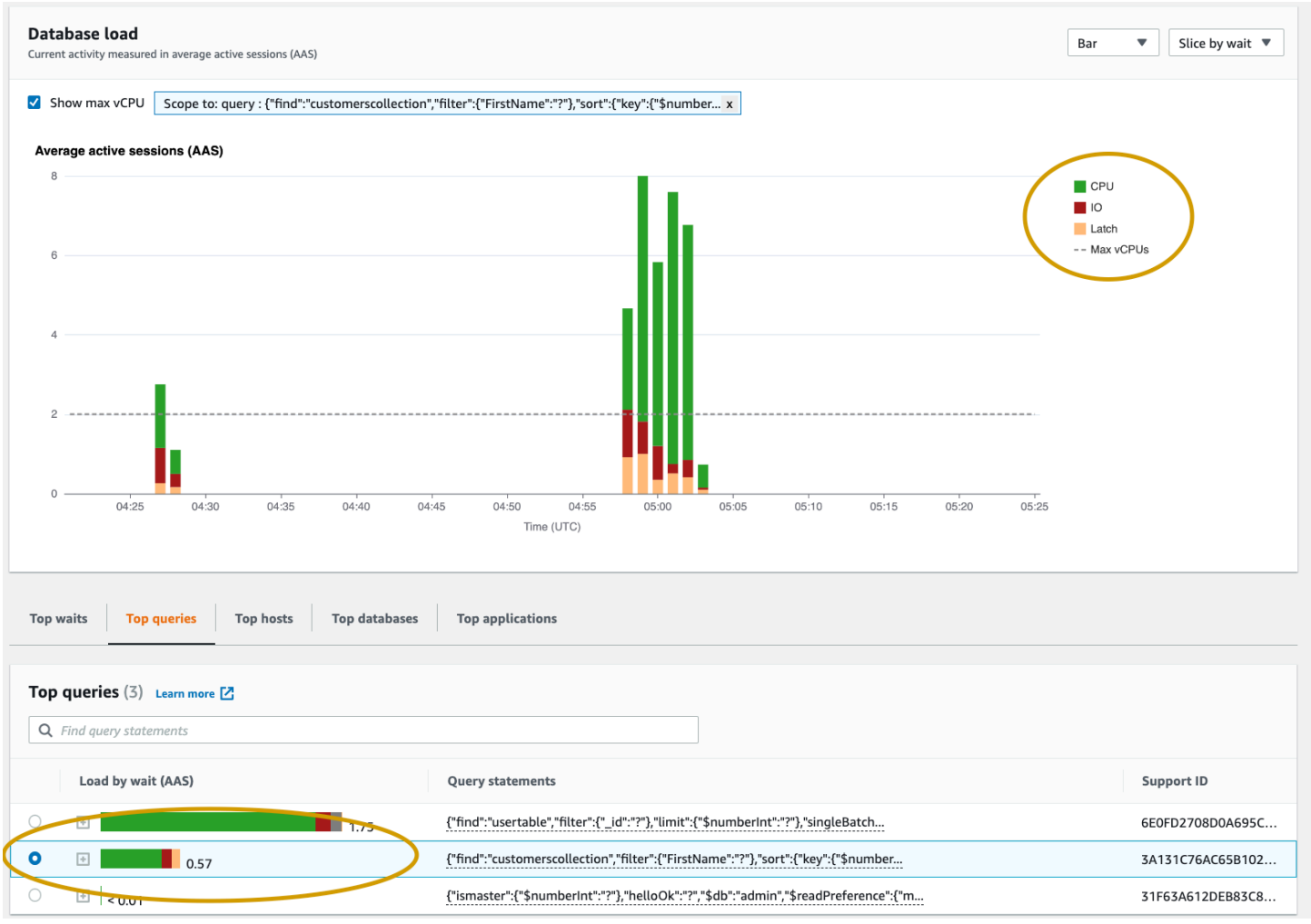
To see the literal query statements in a digest, select the query, and then choose the plus symbol (+). In the following screenshot, the selected query is a digest.



> **Note**
>
> A query digest groups similar query statements, but does not redact sensitive information.

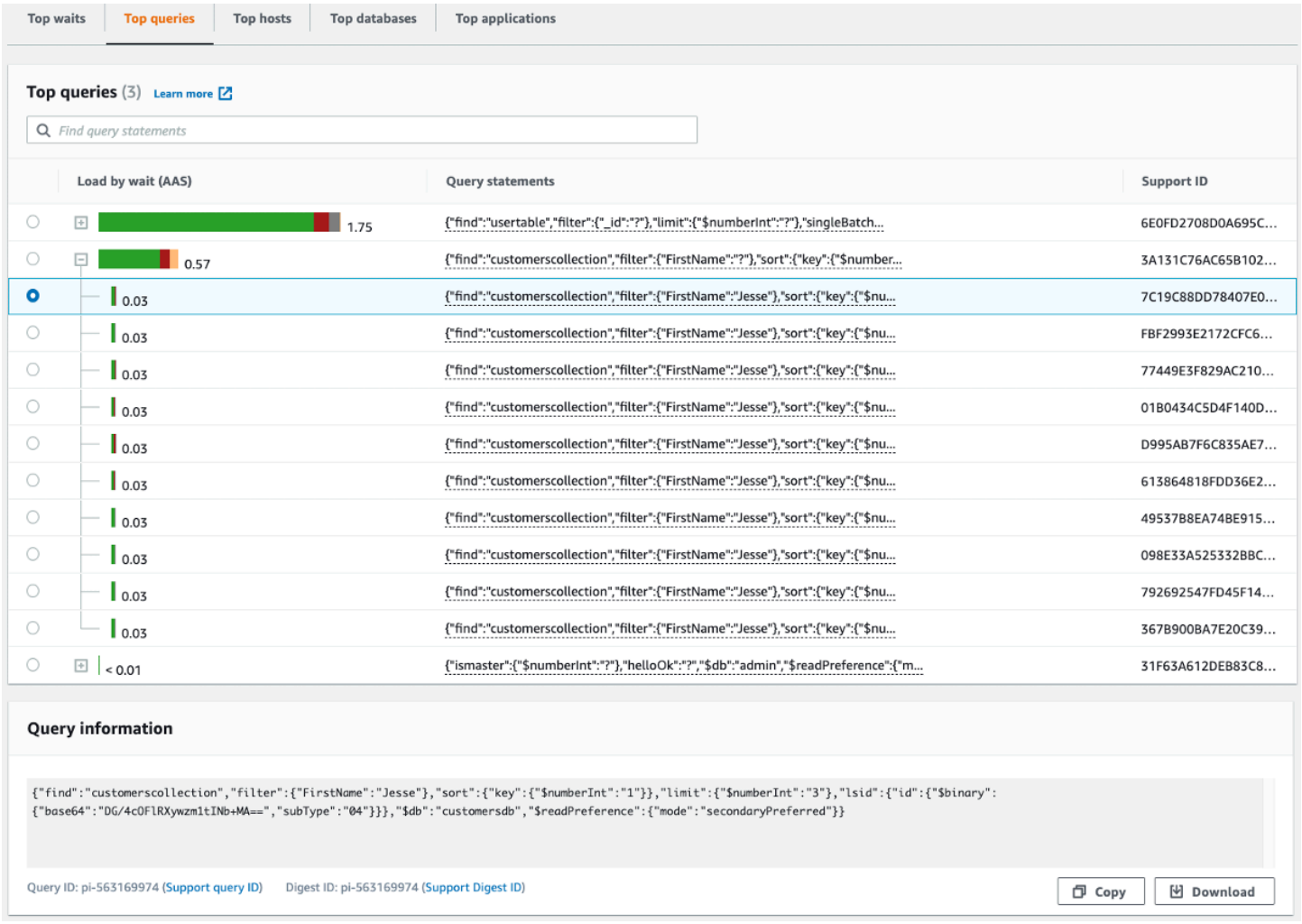
Load by waits (AAS)

In **Top queries**, the **Load by waits (AAS)** column illustrates the percentage of the database load associated with each top load item. This column reflects the load for that item by whatever grouping is currently selected in the **DB load chart**. For example, you might group the **DB load chart** by wait states. In this case, the **DB Load by Waits** bar is sized, segmented, and color-coded to show how much of a given wait state that query is contributing to. It also shows which wait states are affecting the selected query.



Viewing detailed query information

In the **Top query** table, you can open a *digest statement* to view its information. The information appears in the bottom pane.



The following types of identifiers (IDs) are associated with query statements:

1. **Support query ID** – A hash value of the query ID. This value is only for referencing a query ID when you are working with AWS Support. AWS Support doesn't have access to your actual query IDs and query text.
2. **Support digest ID** – A hash value of the digest ID. This value is only for referencing a digest ID when you are working with AWS Support. AWS Support doesn't have access to your actual digest IDs and query text.

Accessing statement query text

By default, each row in the **Top queries** table shows 500 bytes of query text for each query statement. When a digest statement exceeds 500 bytes, you can view more text by opening the statement in the Performance Insights dashboard. In this case, the maximum length for the displayed query is 1 KB. If you view a full query statement, you can also choose **Download**.

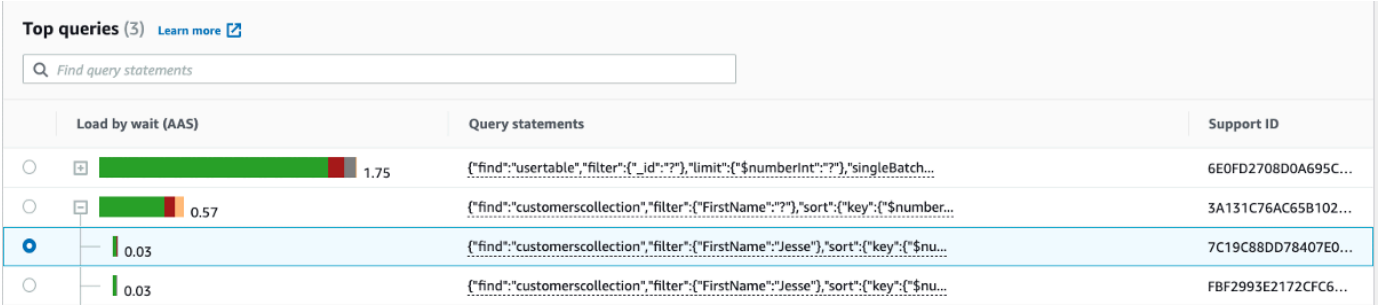
Viewing and downloading statement query text

In the Performance Insights dashboard, you can view or download query text.

To view more query text in the Performance Insights dashboard

1. Open the Amazon DocumentDB console at: https://console.aws.amazon.com/docdb/
2. In the navigation pane, choose **Performance Insights**.
3. Choose a DB instance. The Performance Insights dashboard is displayed for that DB instance.

   Query statements with text larger than 500 bytes will look like the following image:



4. Examine the query information section to view more of the query text.



The Performance Insights dashboard can display up to 1 KB for each full query statement.

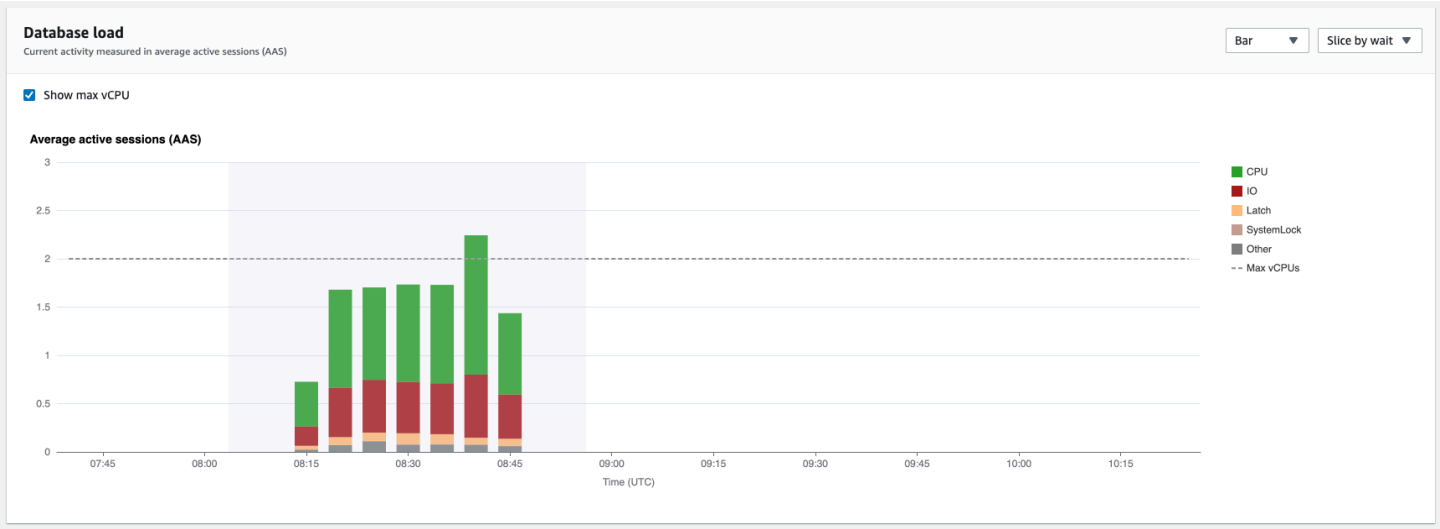
> **Note**
>
> To copy or download the query statement, disable any pop-up blockers.

Zooming in on the database load chart

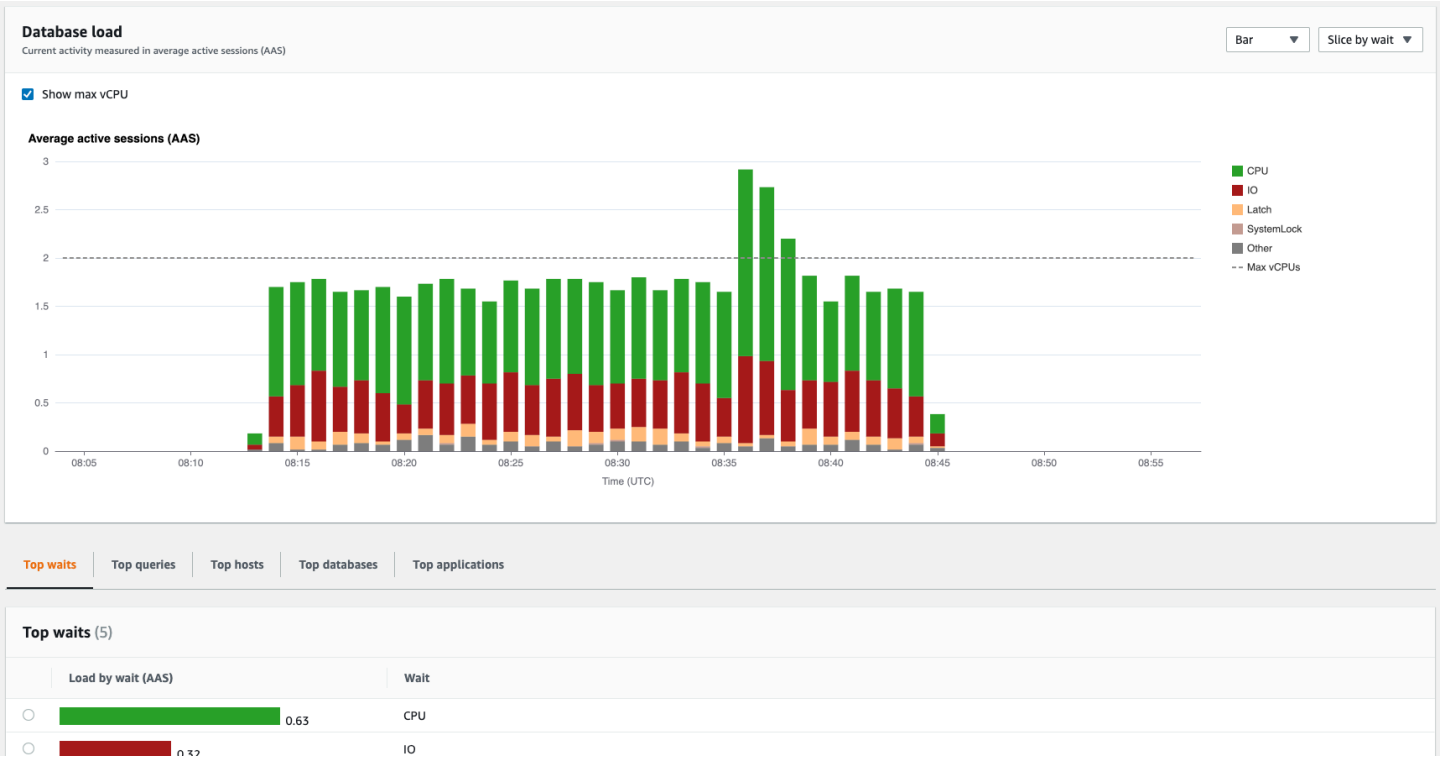
You can use other features of the Performance Insights user interface to help analyze performance data.

Click-and-Drag Zoom In

In the Performance Insights interface, you can choose a small portion of the load chart and zoom in on the detail.



To zoom in on a portion of the load chart, choose the start time and drag to the end of the time period you want. When you do this, the selected area is highlighted. When you release the mouse, the load chart zooms in on the selected area, and the **Top *items*** table is recalculated.



Retrieving metrics with the Performance Insights API

When Performance Insights is enabled, the API provides visibility into instance performance. Amazon CloudWatch Logs provides the authoritative source for vended monitoring metrics for AWS services.

Performance Insights offers a domain-specific view of database load measured as average active sessions (AAS). This metric appears to API consumers as a two-dimensional time-series dataset. The time dimension of the data provides DB load data for each time point in the queried time range. Each time point decomposes overall load in relation to the requested dimensions, such as `Query`, `Wait-state`, `Application`, or `Host`, measured at that time point.

Amazon DocumentDB Performance Insights monitors your Amazon DocumentDB DB instance so that you can analyze and troubleshoot database performance. One way to view Performance Insights data is in the AWS Management Console. Performance Insights also provides a public API so that you can query your own data. You can use the API to do the following:

- Offload data into a database
- Add Performance Insights data to existing monitoring dashboards
- Build monitoring tools

To use the Performance Insights API, enable Performance Insights on one of your Amazon DocumentDB instances. For information about enabling Performance Insights, see Enabling and disabling Performance Insights. For more information about the Performance Insights API, see the Performance Insights API Reference.

The Performance Insights API provides the following operations.

Performance Insights action	AWS CLI command	Description
DescribeDimensionKeys	aws pi describe-dimension-keys	Retrieves the top N dimension keys for a metric for a specific time period.
GetDimensionKeyDetails	aws pi get-dimension-key-details	Retrieves the attributes of the specified dimension group for a DB instance or data source. For example, if you specify a query ID, and if the dimension details are available, `GetDimensionKeyDetails` retrieves the full text of the dimension `db.query.statement` associated with this ID. This operation is useful because `GetResourceMetrics` and `DescribeDimensionKeys` don't support retrieval of large query statement text.
GetResourceMetadata	aws pi get-resource-metadata	Retrieve the metadata for different features. For example, the metadata might indicate that a feature is turned on or off on a specific DB instance.

Performance Insights action	AWS CLI command	Description
GetResourceMetrics	aws pi get-resource-metrics	Retrieves Performance Insights metrics for a set of data sources over a time period. You can provide specific dimension groups and dimensions, and provide aggregation and filtering criteria for each group.
ListAvailableResourceDimensions	aws pi list-available-resource-dimensions	Retrieve the dimensions that can be queried for each specified metric type on a specified instance.
ListAvailableResourceMetrics	aws pi list-available-resource-metrics	Retrieve all available metrics of the specified metric types that can be queried for a specified DB instance.

Topics

- AWS CLI for Performance Insights
- Retrieving time-series metrics
- AWS CLI examples for Performance Insights

AWS CLI for Performance Insights

You can view Performance Insights data using the AWS CLI. You can view help for the AWS CLI commands for Performance Insights by entering the following on the command line.

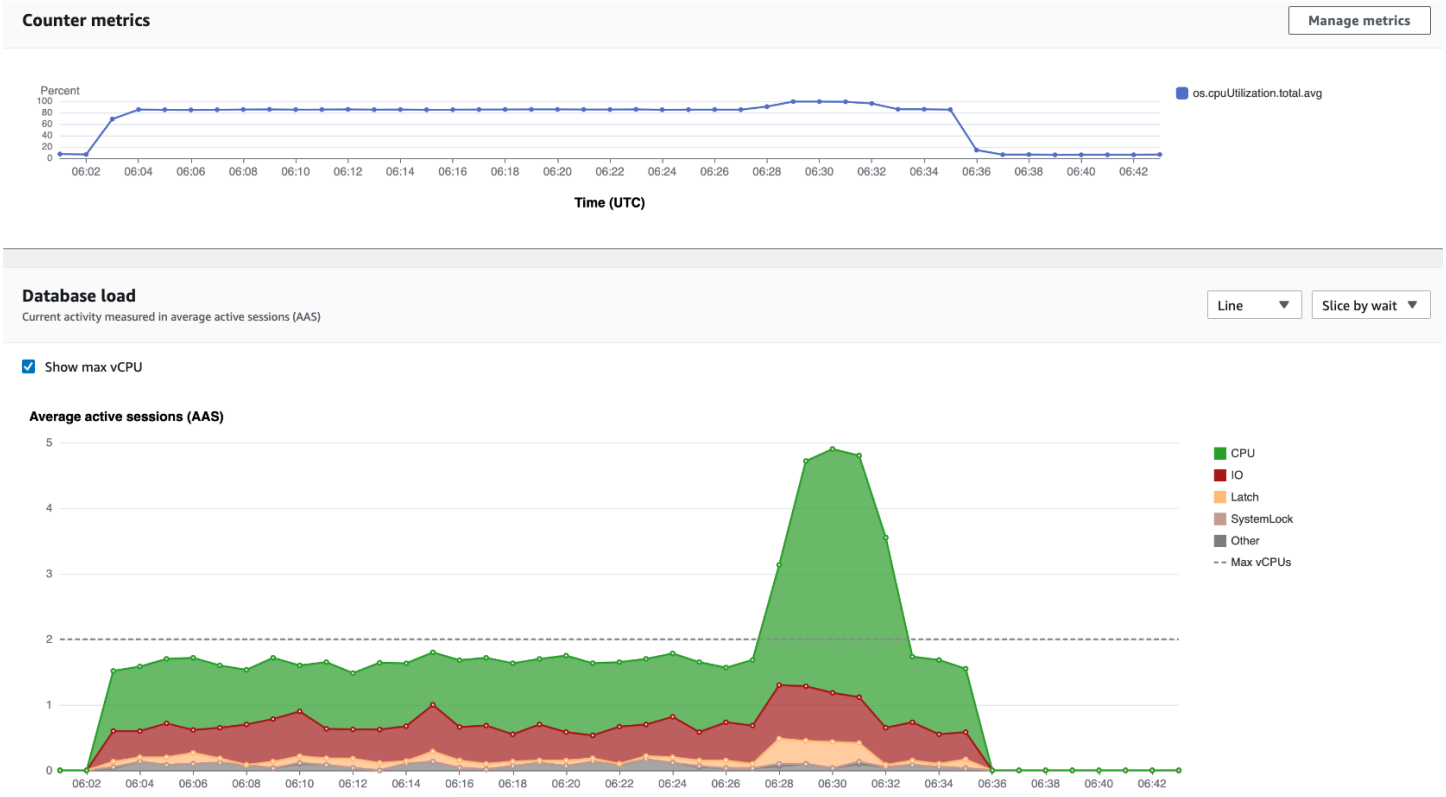
```
aws pi help
```

If you don't have the AWS CLI installed, see Installing the AWS Command Line Interface in the *AWS CLI User Guide* for information about installing it.

Retrieving time-series metrics

The `GetResourceMetrics` operation retrieves one or more time-series metrics from the Performance Insights data. `GetResourceMetrics` requires a metric and time period, and returns a response with a list of data points.

For example, the AWS Management Console uses `GetResourceMetrics` to populate the **Counter Metrics** chart and the **Database Load** chart, as seen in the following image.



All metrics returned by `GetResourceMetrics` are standard time-series metrics, with the exception of `db.load`. This metric is displayed in the **Database Load** chart. The `db.load` metric is different from the other time-series metrics because you can break it into subcomponents called *dimensions*. In the previous image, `db.load` is broken down and grouped by the waits states that make up the `db.load`.

> **Note**
>
> `GetResourceMetrics` can also return the `db.sampleload` metric, but the `db.load` metric is appropriate in most cases.

For information about the counter metrics returned by `GetResourceMetrics`, see Performance Insights for counter metrics.

The following calculations are supported for the metrics:

- Average – The average value for the metric over a period of time. Append `.avg` to the metric name.
- Minimum – The minimum value for the metric over a period of time. Append `.min` to the metric name.
- Maximum – The maximum value for the metric over a period of time. Append `.max` to the metric name.
- Sum – The sum of the metric values over a period of time. Append `.sum` to the metric name.
- Sample count – The number of times the metric was collected over a period of time. Append `.sample_count` to the metric name.

For example, assume that a metric is collected for 300 seconds (5 minutes), and that the metric is collected one time each minute. The values for each minute are 1, 2, 3, 4, and 5. In this case, the following calculations are returned:

- Average – 3
- Minimum – 1
- Maximum – 5
- Sum – 15
- Sample count – 5

For information about using the `get-resource-metrics` AWS CLI command, see `get-resource-metrics`.

For the `--metric-queries` option, specify one or more queries that you want to get results for. Each query consists of a mandatory `Metric` and optional `GroupBy` and `Filter` parameters. The following is an example of a `--metric-queries` option specification.

```
{
   "Metric": "string",
   "GroupBy": {
     "Group": "string",
```

```
    "Dimensions": ["string", ...],
    "Limit": integer
  },
  "Filter": {"string": "string"
    ...}
```

AWS CLI examples for Performance Insights

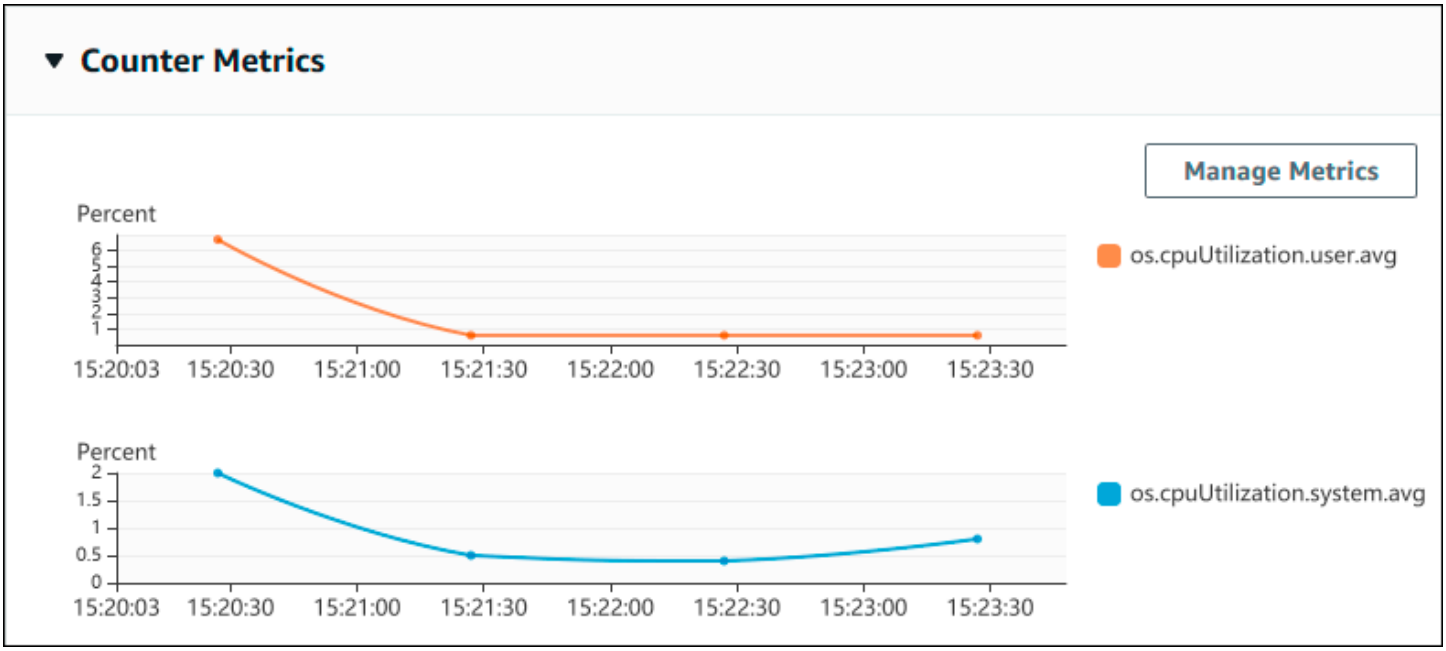
The following examples show how to use the AWS CLI for Performance Insights.

Topics

- Retrieving counter metrics
- Retrieving the DB load average for top wait states
- Retrieving the DB load average for top Query
- Retrieving the DB load average filtered by Query

Retrieving counter metrics

The following screenshot shows two counter metrics charts in the AWS Management Console.



The following example shows how to gather the same data that the AWS Management Console uses to generate the two counter metric charts.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type DOCDB \
   --identifier db-ID \
   --start-time 2022-03-13T8:00:00Z \
   --end-time   2022-03-13T9:00:00Z \
   --period-in-seconds 60 \
   --metric-queries '[{"Metric": "os.cpuUtilization.user.avg"  },
                      {"Metric": "os.cpuUtilization.idle.avg"}]'
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type DOCDB ^
   --identifier db-ID ^
   --start-time 2022-03-13T8:00:00Z ^
   --end-time   2022-03-13T9:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries '[{"Metric": "os.cpuUtilization.user.avg"  },
                      {"Metric": "os.cpuUtilization.idle.avg"}]'
```

You can also make a command easier to read by specifying a file for the `--metrics-query` option. The following example uses a file called query.json for the option. The file has the following contents.

```
[
    {
        "Metric": "os.cpuUtilization.user.avg"
    },
    {
        "Metric": "os.cpuUtilization.idle.avg"
    }
]
```

Run the following command to use the file.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type DOCDB \
   --identifier db-ID \
   --start-time 2022-03-13T8:00:00Z \
   --end-time   2022-03-13T9:00:00Z \
```

```
   --period-in-seconds 60 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type DOCDB ^
   --identifier db-ID ^
   --start-time 2022-03-13T8:00:00Z ^
   --end-time   2022-03-13T9:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries file://query.json
```

The preceding example specifies the following values for the options:

- `--service-type` – DOCDB for Amazon DocumentDB
- `--identifier` – The resource ID for the DB instance
- `--start-time` and `--end-time` – The ISO 8601 `DateTime` values for the period to query, with multiple supported formats

It queries for a one-hour time range:

- `--period-in-seconds` – 60 for a per-minute query
- `--metric-queries` – An array of two queries, each just for one metric.

  The metric name uses dots to classify the metric in a useful category, with the final element being a function. In the example, the function is `avg` for each query. As with Amazon CloudWatch, the supported functions are `min`, `max`, `total`, and `avg`.

The response looks similar to the following.

```
{
    "AlignedStartTime": "2022-03-13T08:00:00+00:00",
    "AlignedEndTime": "2022-03-13T09:00:00+00:00",
    "Identifier": "db-NQF3TTMFQ3GTOKIMJODMC3KQQ4",
    "MetricList": [
        {
            "Key": {
                "Metric": "os.cpuUtilization.user.avg"
```

```
            },
            "DataPoints": [
                {
                    "Timestamp": "2022-03-13T08:01:00+00:00", //Minute1
                    "Value": 3.6
                },
                {
                    "Timestamp": "2022-03-13T08:02:00+00:00", //Minute2
                    "Value": 2.6
                },
                //.... 60 datapoints for the os.cpuUtilization.user.avg metric
        {
            "Key": {
                "Metric": "os.cpuUtilization.idle.avg"
            },
            "DataPoints": [
                {
                    "Timestamp": "2022-03-13T08:01:00+00:00",
                    "Value": 92.7
                },
                {
                    "Timestamp": "2022-03-13T08:02:00+00:00",
                    "Value": 93.7
                },
                //.... 60 datapoints for the os.cpuUtilization.user.avg metric
            ]
        }
    ] //end of MetricList
} //end of response
```

The response has an `Identifier`, `AlignedStartTime`, and `AlignedEndTime`. B the `--period-in-seconds` value was `60`, the start and end times have been aligned to the minute. If the `--period-in-seconds` was `3600`, the start and end times would have been aligned to the hour.

The `MetricList` in the response has a number of entries, each with a `Key` and a `DataPoints` entry. Each `DataPoint` has a `Timestamp` and a `Value`. Each `Datapoints` list has 60 data points because the queries are for per-minute data over an hour, with `Timestamp1/Minute1`, `Timestamp2/Minute2`, and so on, up to `Timestamp60/Minute60`.

Because the query is for two different counter metrics, there are two elements in the response `MetricList`.

Retrieving the DB load average for top wait states

The following example is the same query that the AWS Management Console uses to generate a stacked area line graph. This example retrieves the `db.load.avg` for the last hour with load divided according to the top seven wait states. The command is the same as the command in Retrieving counter metrics. However, the query.json file has the following contents.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.wait_state", "Limit": 7 }
    }
]
```

Run the following command.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type DOCDB \
   --identifier db-ID \
   --start-time 2022-03-13T8:00:00Z \
   --end-time   2022-03-13T9:00:00Z \
   --period-in-seconds 60 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type DOCDB ^
   --identifier db-ID ^
   --start-time 2022-03-13T8:00:00Z ^
   --end-time   2022-03-13T9:00:00Z ^
   --period-in-seconds 60 ^
   --metric-queries file://query.json
```

The example specifies the metric of `db.load.avg` and a `GroupBy` of the top seven wait states. For details about valid values for this example, see DimensionGroup in the *Performance Insights API Reference.*

The response looks similar to the following.

```
{
    "AlignedStartTime": "2022-04-04T06:00:00+00:00",
    "AlignedEndTime": "2022-04-04T06:15:00+00:00",
    "Identifier": "db-NQF3TTMFQ3GTOKIMJODMC3KQQ4",
    "MetricList": [
        {//A list of key/datapoints
            "Key": {
                //A Metric with no dimensions. This is the total db.load.avg
                "Metric": "db.load.avg"
            },
            "DataPoints": [
                //Each list of datapoints has the same timestamps and same number of
 items
                {
                    "Timestamp": "2022-04-04T06:01:00+00:00",//Minute1
                    "Value": 0.0
                },
                {
                    "Timestamp": "2022-04-04T06:02:00+00:00",//Minute2
                    "Value": 0.0
                },
                //... 60 datapoints for the total db.load.avg key
                ]
        },
        {
            "Key": {
                //Another key. This is db.load.avg broken down by CPU
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.wait_state.name": "CPU"
                }
            },
            "DataPoints": [
                {
                    "Timestamp": "2022-04-04T06:01:00+00:00",//Minute1
                    "Value": 0.0
                },
                {
                    "Timestamp": "2022-04-04T06:02:00+00:00",//Minute2
                    "Value": 0.0
                },
                //... 60 datapoints for the CPU key
            ]
```

```
        },//... In total we have 3 key/datapoints entries, 1) total, 2-3) Top Wait
 States
    ] //end of MetricList
} //end of response
```

In this response, there are three entries in the `MetricList`. There is one entry for the total `db.load.avg`, and three entries each for the `db.load.avg` divided according to one of the top three wait states. Since there was a grouping dimension (unlike the first example), there must be one key for each grouping of the metric. There can't be only one key for each metric, as in the basic counter metric use case.

Retrieving the DB load average for top Query

The following example groups `db.wait_state` by the top 10 query statements. There are two different groups for query statements:

- `db.query` – The full query statement, such as `{"find":"customers","filter": {"FirstName":"Jesse"},"sort":{"key":{"$numberInt":"1"}}}`
- `db.query_tokenized` – The tokenized query statement, such as `{"find":"customers","filter":{"FirstName":"?"},"sort":{"key": {"$numberInt":"?"}},"limit":{"$numberInt":"?"}}`

When analyzing database performance, it can be useful to consider query statements that only differ by their parameters as one logic item. So, you can use `db.query_tokenized` when querying. However, especially when you're interested in `explain()`, sometimes it's more useful to examine full query statements with parameters. There is a parent-child relationship between tokenized and full queries, with multiple full queries (children) grouped under the same tokenized query (parent).

The command in this example is the similar to the command in Retrieving the DB load average for top wait states. However, the query.json file has the following contents.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.query_tokenized", "Limit": 10 }
    }
]
```

The following example uses `db.query_tokenized`.

For Linux, macOS, or Unix:

```
aws pi get-resource-metrics \
   --service-type DOCDB \
   --identifier db-ID \
   --start-time 2022-03-13T8:00:00Z \
   --end-time   2022-03-13T9:00:00Z \
   --period-in-seconds 3600 \
   --metric-queries file://query.json
```

For Windows:

```
aws pi get-resource-metrics ^
   --service-type DOCDB ^
   --identifier db-ID ^
   --start-time 2022-03-13T8:00:00Z ^
   --end-time   2022-03-13T9:00:00Z  ^
   --period-in-seconds 3600 ^
   --metric-queries file://query.json
```

This example queries over 1 hour, with a one minute period-in-seconds.

The example specifies the metric of `db.load.avg` and a `GroupBy` of the top seven wait states. For details about valid values for this example, see DimensionGroup in the *Performance Insights API Reference*.

The response looks similar to the following.

```
{
    "AlignedStartTime": "2022-04-04T06:00:00+00:00",
    "AlignedEndTime": "2022-04-04T06:15:00+00:00",
    "Identifier": "db-NQF3TTMFQ3GTOKIMJODMC3KQQ4",
    "MetricList": [
        {//A list of key/datapoints
            "Key": {
                "Metric": "db.load.avg"
            },
            "DataPoints": [
                //... 60 datapoints for the total db.load.avg key
                ]
```

```
        },
               {
            "Key": {//Next key are the top tokenized queries
                "Metric": "db.load.avg",
                "Dimensions": {
                    "db.query_tokenized.db_id": "pi-1064184600",
                    "db.query_tokenized.id": "77DE8364594EXAMPLE",
                    "db.query_tokenized.statement": "{\"find\":\"customers\",\"filter
\":{\"FirstName\":\"?\"},\"sort\":{\"key\":{\"$numberInt\":\"?\"}},\"limit\"
:{\"$numberInt\":\"?\"},\"$db\":\"myDB\",\"$readPreference\":{\"mode\":\"primary\"}}"
                }
            },
            "DataPoints": [
            //... 60 datapoints
            ]
        },
        // In total 11 entries, 10 Keys of top tokenized queries, 1 total key
    ] //End of MetricList
} //End of response
```

This response has 11 entries in the `MetricList` (1 total, 10 top tokenized query), with each entry having 24 per-hour `DataPoints`.

For tokenized queries, there are three entries in each dimensions list:

- `db.query_tokenized.statement` – The tokenized query statement.
- `db.query_tokenized.db_id` – The synthetic ID that Performance Insights generates for you. This example returns the `pi-1064184600` synthetic ID.
- `db.query_tokenized.id` – The ID of the query inside Performance Insights.

  In the AWS Management Console, this ID is called the Support ID. It's named this because the ID is data that AWS Support can examine to help you troubleshoot an issue with your database. AWS takes the security and privacy of your data extremely seriously, and almost all data is stored encrypted with your AWS KMS key. Therefore, nobody inside AWS can look at this data. In the example preceding, both the `tokenized.statement` and the `tokenized.db_id` are stored encrypted. If you have an issue with your database, AWS Support can help you by referencing the Support ID.

When querying, it might be convenient to specify a `Group` in `GroupBy`. However, for finer-grained control over the data that's returned, specify the list of dimensions. For example, if all that is

needed is the `db.query_tokenized.statement`, then a `Dimensions` attribute can be added to the query.json file.

```
[
    {
        "Metric": "db.load.avg",
        "GroupBy": {
            "Group": "db.query_tokenized",
            "Dimensions":["db.query_tokenized.statement"],
            "Limit": 10
        }
    }
]
```

Retrieving the DB load average filtered by Query

The corresponding API query in this example is similar to the command in Retrieving the DB load average for top Query. However, the query.json file has the following contents.

```
[
 {
        "Metric": "db.load.avg",
        "GroupBy": { "Group": "db.wait_state", "Limit": 5  },
        "Filter": { "db.query_tokenized.id": "AKIAIOSFODNN7EXAMPLE" }
    }
]
```

In this response, all values are filtered according to the contribution of tokenized query AKIAIOSFODNN7EXAMPLE specified in the query.json file. The keys also might follow a different order than a query without a filter, because it's the top five wait states that affected the filtered query.

Amazon CloudWatch metrics for Performance Insights

Performance Insights automatically publishes metrics to Amazon CloudWatch. The same data can be queried from Performance Insights, but having the metrics in CloudWatch makes it easy to add CloudWatch alarms. It also makes it easy to add the metrics to existing CloudWatch Dashboards.

Metric	Description
DBLoad	The number of active sessions for Amazon DocumentDB. Typically, you want the data for the average number of active sessions. In Performance Insights, this data is queried as `db.load.avg` .
DBLoadCPU	The number of active sessions where the wait state type is CPU. In Performance Insights, this data is queried as `db.load.avg` , filtered by the wait state type CPU.
DBLoadNonCPU	The number of active sessions where the wait state type is not CPU.

Note

These metrics are published to CloudWatch only if there is load on the DB instance.

You can examine these metrics using the CloudWatch console, the AWS CLI, or the CloudWatch API.

For example, you can get the statistics for the `DBLoad` metric by running the get-metric-statistics command.

```
aws cloudwatch get-metric-statistics \
    --region ap-south-1 \
    --namespace AWS/DocDB \
    --metric-name DBLoad  \
    --period 360 \
    --statistics Average \
    --start-time 2022-03-14T8:00:00Z \
    --end-time 2022-03-14T9:00:00Z \
    --dimensions Name=DBInstanceIdentifier,Value=documentdbinstance
```

This example generates output similar to the following.

```
{
```

```
    "Datapoints": [
        {
            "Timestamp": "2022-03-14T08:42:00Z",
            "Average": 1.0,
            "Unit": "None"
        },
        {
            "Timestamp": "2022-03-14T08:24:00Z",
            "Average": 2.0,
            "Unit": "None"
        },
        {
            "Timestamp": "2022-03-14T08:54:00Z",
            "Average": 6.0,
            "Unit": "None"
        },
        {
            "Timestamp": "2022-03-14T08:36:00Z",
            "Average": 5.7,
            "Unit": "None"
        },
        {
            "Timestamp": "2022-03-14T08:06:00Z",
            "Average": 4.0,
            "Unit": "None"
        },
        {
            "Timestamp": "2022-03-14T08:00:00Z",
            "Average": 5.2,
            "Unit": "None"
        }
    ],
    "Label": "DBLoad"
}
```

You can use the DB_PERF_INSIGHTS metric math function in the CloudWatch console to query Amazon DocumentDB Performance Insights counter metrics. The DB_PERF_INSIGHTS function also includes the DBLoad metric at sub-minute intervals. You can set CloudWatch alarms on these metrics. For more details on how to create an alarm, see Create an alarm on Performance Insights counter metrics from an AWS database.

For more information about CloudWatch, see What is Amazon CloudWatch? in the *Amazon CloudWatch User Guide*.

Performance Insights for counter metrics

Counter metrics are operating system metrics in the Performance Insights dashboard. To help identify and analyze performance problems, you can correlate counter metrics with DB load.

Performance Insights operating system counters

The following operating system counters are available with Amazon DocumentDB Performance Insights.

Counter	Type	Metric
active	memory	os.memory.active
buffers	memory	os.memory.buffers
cached	memory	os.memory.cached
dirty	memory	os.memory.dirty
free	memory	os.memory.free
inactive	memory	os.memory.inactive
mapped	memory	os.memory.mapped
pageTables	memory	os.memory.pageTables
slab	memory	os.memory.slab
total	memory	os.memory.total
writeback	memory	os.memory.writeback
idle	cpuUtilization	os.cpuUtilization.idle
system	cpuUtilization	os.cpuUtilization.system
total	cpuUtilization	os.cpuUtilization.total

Counter	Type	Metric
user	cpuUtilization	os.cpuUtilization.user
wait	cpuUtilization	os.cpuUtilization.wait
one	loadAverageMinute	os.loadAverageMinute.one
fifteen	loadAverageMinute	os.loadAverageMinute.fifteen
five	loadAverageMinute	os.loadAverageMinute.five
cached	swap	os.swap.cached
free	swap	os.swap.free
in	swap	os.swap.in
out	swap	os.swap.out
total	swap	os.swap.total
rx	network	os.network.rx
tx	network	os.network.tx
numVCPUs	general	os.general.numVCPUs

Using Amazon DocumentDB serverless

Amazon DocumentDB serverless is an on-demand, auto scaling configuration that automates the processes of dynamically adjusting the capacity of your Amazon DocumentDB database based on application demand. You're charged only for the resources that your clusters consume. Thus, DocumentDB serverless can help you to stay within budget and avoid paying for computer resources that you don't use.

This type of automation is especially valuable for environments with highly variable and unpredictable workloads, such as multitenant databases, distributed databases, and development and test systems.

Topics

- DocumentDB serverless use cases
- Advantages of Amazon DocumentDB serverless
- How Amazon DocumentDB serverless works
- Requirements and limitations for DocumentDB serverless
- Creating a cluster that uses Amazon DocumentDB serverless
- Migrating to Amazon DocumentDB serverless
- Managing Amazon DocumentDB serverless
- Amazon DocumentDB serverless instance limits
- Amazon DocumentDB serverless scaling configuration
- Monitoring Amazon DocumentDB serverless

DocumentDB serverless use cases

Both Amazon DocumentDB provisioned clusters and DocumentDB serverless support many types of database workloads, from development and testing environments to the most demanding, business-critical applications that require high scale and availability. But DocumentDB serverless adds another dimension to customer workloads, namely the ability to support websites and applications that have unpredictable workloads.

DocumentDB serverless is especially useful for the following use cases:

- **Variable workloads** — You're running workloads that have sudden and unpredictable increases in activity. An example is a traffic site that sees a surge of activity when it starts raining. Another is an e-commerce site with increased traffic when you offer sales or special promotions. With DocumentDB serverless, your database automatically scales capacity to meet the needs of the application's peak load and scales back down when the surge of activity is over. With DocumentDB serverless, you no longer need to provision for peak or average capacity. You can specify an upper capacity limit to handle the worst-case situation, and that capacity isn't used unless it's needed.
  - The granularity of scaling in DocumentDB serverless helps you to match capacity closely to your database's needs. For a provisioned cluster, scaling up requires adding a whole new instance. DocumentDB serverless can add half a DCU when only a little more capacity is needed. It can add 0.5, 1, 1.5, 2, or additional half-DCUs based on the additional capacity needed to handle an increase in workload. And it can remove 0.5, 1, 1.5, 2, or additional half-DCUs when the workload decreases and that capacity is no longer needed.
- **Multi-tenant applications** — With DocumentDB serverless, you don't have to individually manage database capacity for each application in your fleet. DocumentDB serverless manages individual database capacity for you.
  - You can create a cluster for each tenant. That way, you can use features such as cloning and snapshot restore to enhance high availability and disaster recovery as appropriate for each tenant.
  - Each tenant might have specific busy and idle periods depending on the time of day, time of year, promotional events, and so on. Each cluster can have a wide capacity range. That way, clusters with low activity incur minimal instance charges. Any cluster can quickly scale up to handle periods of high activity.
- **New applications** — You're deploying a new application and you're unsure about the instance size you need. By using DocumentDB serverless, you can set up a cluster with one or many instances and have the database autoscale to the capacity requirements of your application.
- **Mixed-use applications** — Suppose that you have an online transaction processing (OLTP) application, but you periodically experience spikes in query traffic. By specifying promotion tiers for the DocumentDB serverless instances in a cluster, you can configure your cluster so that the reader instances can scale independently of the writer instance to handle the additional load. When the usage spike subsides, the reader instances scale back down to match the capacity of the writer instance.
- **Capacity planning** — Suppose that you usually adjust your database capacity, or verify the optimal database capacity for your workload, by modifying the instance classes of all the

instances in a cluster. With DocumentDB serverless, you can avoid this administrative overhead. You can determine the appropriate minimum and maximum capacity by running the workload and checking how much the instances actually scale.

  - You can modify existing instances from provisioned to DocumentDB serverless or from DocumentDB serverless to provisioned. You don't need to create a new cluster or a new instance in such cases.
- **Development and testing** — In addition to running your most demanding applications, you can also use DocumentDB serverless for development and testing environments. With DocumentDB serverless, you can create instances with a low minimum capacity instead of using burstable db.t* instance classes. You can set the maximum capacity high enough that those instances can still run substantial workloads without running low on memory. When the database isn't in use, all of the instances scale down to avoid unnecessary charges.

Using Amazon DocumentDB serverless for existing provisioned workloads

Suppose that you already have an DocumentDB application running on a provisioned cluster. You can check how the application would work with DocumentDB serverless by adding one or more DocumentDB serverless instances to the existing cluster as reader instances. You can check how often the reader instances scale up and down. You can use the DocumentDB failover mechanism to promote an DocumentDB serverless instance to be the writer and check how it handles the read/write workload. That way, you can switch over with minimal downtime and without changing the endpoint that your client applications use. For details on the procedure to convert existing clusters to DocumentDB serverless, see Migrating to Amazon DocumentDB serverless.

Advantages of Amazon DocumentDB serverless

DocumentDB serverless is intended for variable or "spiky" workloads. With such unpredictable workloads, you might have difficulty planning when to change your database capacity. You might also have trouble making capacity changes quickly enough using the familiar mechanisms such as adding instances or changing instance classes. DocumentDB serverless provides the following advantages to help with such use cases:

- **Simpler capacity management than provisioned** — DocumentDB serverless reduces the effort for planning instance sizes and resizing instances as the workload changes. It also reduces the effort for maintaining consistent capacity for all the instances in a cluster.

- **Faster and easier scaling during periods of high activity** — DocumentDB serverless scales compute and memory capacity as needed, with no disruption to client transactions or your overall workload. The ability to use reader instances with DocumentDB serverless helps you to take advantage of horizontal scaling in addition to vertical scaling. The ability to use DocumentDB global clusters means that you can spread your DocumentDB serverless read workload across multiple AWS Regions. This capability is more convenient than the scaling mechanisms for provisioned clusters.

- **Cost-effective during periods of low activity** — DocumentDB serverless helps you to avoid overprovisioning your instances. DocumentDB serverless adds resources in granular increments when instances scale up. You pay only for the database resources that you consume. DocumentDB serverless resource usage is measured on a per-second basis. That way, when a instance scales down, the reduced resource usage is registered right away.

- **Feature parity with provisioned** — You can use all DocumentDB features with DocumentDB serverless. For example, with DocumentDB serverless you can use reader instances, AWS Identity and Access Management (IAM) database authentication, and Performance Insights.

  In particular, with DocumentDB serverless you can take advantage of the following features from provisioned clusters:

  - **Reader instances** — DocumentDB serverless can take advantage of reader instances to scale horizontally. When a cluster contains one or more reader instances, the cluster can fail over immediately in case of problems with the writer instance.
  - **Multi-AZ clusters** — You can distribute the DocumentDB serverless instances of a cluster across multiple Availability Zones (AZs). Setting up a Multi-AZ cluster helps to ensure business continuity even in the rare case of issues that affect an entire AZ.
  - **Global clusters** — You can use DocumentDB serverless in combination with DocumentDB global clusters to create additional read-only copies of your cluster in other AWS Regions for disaster recovery purposes.

How Amazon DocumentDB serverless works

Topics

- Overview
- Configurations for Amazon DocumentDB clusters

- Amazon DocumentDB serverless scaling capacity
- Amazon DocumentDB serverless scaling
- Idle state (0.5 DCUs)

Overview

Amazon DocumentDB serverless is suitable for the most demanding, highly variable workloads. For example, your database usage might be heavy for a short period of time, followed by long periods of light activity or no activity at all. Some examples are retail, gaming, or sports websites with periodic promotional events, and databases that produce reports when needed. Others are development and testing environments, and new applications where usage might ramp up quickly. For cases such as these and many others, configuring capacity correctly in advance isn't always possible with the provisioned model. It can also result in higher costs if you overprovision and have capacity that you don't use.

In contrast, DocumentDB provisioned clusters are suitable for steady workloads. With provisioned clusters, you choose a instance class that has a predefined amount of memory, CPU power, I/O bandwidth, and so on. If your workload changes, you manually modify the instance class of your writer and readers. The provisioned model works well when you can adjust capacity in advance of expected consumption patterns and it's acceptable to have brief outages while you change the instance class of the writer and readers in your cluster.

DocumentDB serverless is architected from the ground up to support serverless clusters that are instantly scalable. DocumentDB serverless is engineered to provide the same degree of security and isolation as with provisioned writers and readers. These aspects are crucial in multitenant serverless cloud environments. The dynamic scaling mechanism has very little overhead so that it can respond quickly to changes in the database workload. It's also powerful enough to meet dramatic increases in processing demand.

By using DocumentDB serverless, you can create an DocumentDB cluster without being locked into a specific database capacity for each writer and reader. You specify the minimum and maximum capacity range. DocumentDB scales each DocumentDB serverless writer or reader in the cluster within that capacity range. By using a Multi-AZ cluster where each writer or reader can scale dynamically, you can take advantage of dynamic scaling and high availability.

DocumentDB serverless scales the database resources automatically based on your minimum and maximum capacity specifications. Scaling is fast because most scaling events operations keep

the writer or reader on the same host. In the rare cases that an DocumentDB serverless writer or reader is moved from one host to another, DocumentDB serverless manages the connections automatically. You don't need to change your database client application code or your database connection strings.

With DocumentDB serverless, as with provisioned clusters, storage capacity and compute capacity are separate. When we refer to DocumentDB serverless capacity and scaling, it's always compute capacity that's increasing or decreasing. Thus, your cluster can contain many terabytes of data even when the CPU and memory capacity scale down to low levels.

Instead of provisioning and managing database servers, you specify database capacity. The actual capacity of each DocumentDB serverless writer or reader varies over time, depending on your workload. For details about that mechanism, see Amazon DocumentDB serverless scaling.

Configurations for Amazon DocumentDB clusters

For each of your Amazon DocumentDB clusters, you can choose any combination of DocumentDB serverless capacity, provisioned capacity, or both.

You can set up a cluster that contains both DocumentDB serverless and provisioned capacity, called a mixed-configuration cluster. For example, suppose that you need more read/write capacity than is available for an DocumentDB serverless writer. In this case, you can set up the cluster with a very large provisioned writer. Then you can still use DocumentDB serverless for the readers. Or suppose that the write workload for your cluster varies but the read workload is steady. In this case, you can set up your cluster with an DocumentDB serverless writer and one or more provisioned readers.

You can also set up a cluster where all the capacity is managed by DocumentDB serverless. To do this, you can create a new cluster and use DocumentDB serverless from the start. Or you can replace all the provisioned capacity in an existing cluster with DocumentDB serverless. For the procedures to create a new cluster with DocumentDB serverless or to switch an existing cluster to DocumentDB serverless, see Creating a cluster that uses Amazon DocumentDB serverless and Migrating to Amazon DocumentDB serverless.

If you don't use DocumentDB serverless at all in a cluster, all the writers and readers in the cluster are provisioned. This is the most common kind of cluster that most users are familiar with. Provisioned capacity is constant. The charges are relatively easy to forecast. However, you have to predict in advance how much capacity you need. In some cases, your predictions might be inaccurate or your capacity needs might change. In these cases, your cluster can become underprovisioned (slower than you want) or overprovisioned (more expensive than you want).

Amazon DocumentDB serverless scaling capacity

The unit of measure for Amazon DocumentDB serverless is the DocumentDB Capacity Unit (DCU). DocumentDB serverless scaling capacity isn't tied to the instance classes that you use for provisioned clusters.

Each DCU is a combination of approximately 2 gibibytes (GiB) of memory, corresponding CPU, and networking. You specify the database capacity range using this unit of measure. The `ServerlessDatabaseCapacity` and `DCUUtilization` CloudWatch metrics help you to determine how much capacity your database is actually using and where that capacity falls within the specified range.

At any moment in time, each DocumentDB serverless writer or reader has a capacity. The capacity is a floating-point number representing DCUs. The capacity increases or decreases whenever the writer or reader scales. This value is measured every second. For each cluster where you intend to use DocumentDB serverless, you define a capacity range: the minimum and maximum capacity values that each DocumentDB serverless writer or reader can scale between. The capacity range is the same for each DocumentDB serverless writer or reader in a cluster. Each DocumentDB serverless writer or reader has its own capacity, falling somewhere in that range.

DocumentDB serverless is supported on DocumentDB 5.0 and 8.0 with a capacity range of 0.5 - 256 DCUs.

Platform versions in DocumentDB serverless represent improvements in performance, scaling capabilities, or features. Amazon DocumentDB automatically manages platform version assignments at the cluster level. All new clusters launch with the latest platform version available in your AWS Region. When a new platform version becomes available, you can upgrade existing clusters on previous platform versions directly to the latest platform version by applying the pending maintenance action. For more information, see Checking and upgrading the platform version of a serverless cluster. Upgrade to the latest platform version to benefit from all the latest improvements.

The following table shows the DocumentDB serverless platform versions with their DCU ranges and performance.

DocumentDB serverless platform version	DCU range	Performance
3	0.5–256	Baseline performance

DocumentDB serverless platform version	DCU range	Performance
4	0.5–256	Up to 35% higher throughput compared to platform version 3

Platform version 3 is available in all AWS Regions where DocumentDB serverless is supported. Platform version 4 is available in the following AWS Regions: US East (N. Virginia), US East (Ohio), US West (Oregon), Africa (Cape Town), Asia Pacific (Hong Kong), Asia Pacific (Hyderabad), Asia Pacific (Jakarta), Asia Pacific (Malaysia), Asia Pacific (Mumbai), Asia Pacific (New Zealand), Asia Pacific (Osaka), Asia Pacific (Seoul), Asia Pacific (Singapore), Asia Pacific (Sydney), Asia Pacific (Thailand), Asia Pacific (Tokyo), Canada (Central), China (Beijing), China (Ningxia), Europe (Frankfurt), Europe (Ireland), Europe (London), Europe (Milan), Europe (Paris), Europe (Spain), Europe (Stockholm), Europe (Zurich), Mexico (Central), South America (São Paulo), AWS GovCloud (US-East), and AWS GovCloud (US-West).

You can determine the platform version of your cluster on the **Configuration** tab of the AWS Management Console, or through the `ServerlessV2PlatformVersion` field in the `DescribeDBClusters` API response. For more information, see DBCluster.

The smallest DocumentDB serverless capacity that you can define is 0.5 DCUs. You can specify a higher number if it's less than or equal to the maximum supported capacity value. Setting the minimum capacity to a small number lets lightly loaded clusters consume minimal compute resources. At the same time, they stay ready to accept connections immediately and scale up when they become busy.

Set the minimum to a value that allows each writer or reader to hold the working set of the application in the buffer pool. That way, the contents of the buffer pool aren't discarded during idle periods. For all the considerations when choosing the scaling capacity range, see Choosing the scaling capacity range for a DocumentDB serverless cluster.

Depending on how you configure the readers in a Multi-AZ deployment, their capacities can be tied to the capacity of the writer or independently. For details about how to do that, see Viewing and modifying the promotion tier of serverless readers.

Monitoring DocumentDB serverless involves measuring the capacity values for the writer and readers in your cluster over time. If your database doesn't scale down to the minimum capacity, you

can take actions such as adjusting the minimum and optimizing your database application. If your database consistently reaches its maximum capacity, you can take actions such as increasing the maximum. You can also optimize your database application and spread the query load across more readers.

The charges for DocumentDB serverless capacity are measured in terms of DCU-hours. For information about how DocumentDB serverless charges are calculated, see Amazon DocumentDB pricing. Suppose that the total number of writers and readers in your cluster is n. In that case, the cluster consumes approximately n x minimum DCUs when you aren't running any database operations. Amazon DocumentDB itself might run monitoring or maintenance operations that cause some small amount of load. That cluster consumes no more than n x maximum DCUs when the database is running at full capacity.

For more details about choosing appropriate minimum and maximum DCU values, see Choosing the scaling capacity range for a DocumentDB serverless cluster. The minimum and maximum DCU values that you specify also affect some Amazon DocumentDB instance limits. For details about the interaction between the capacity range and instance limits, see Amazon DocumentDB serverless instance limits.

Amazon DocumentDB serverless scaling

For each DocumentDB serverless writer or reader, Amazon DocumentDB continuously tracks utilization of resources such as CPU, memory, and network. These measurements collectively are called the load. The load includes the database operations performed by your application. It also includes background processing for the database server and Amazon DocumentDB administrative tasks. When capacity is constrained by any of these, DocumentDB serverless scales up. DocumentDB serverless also scales up when it detects performance issues that it can resolve by doing so. You can monitor resource utilization and how it affects DocumentDB serverless scaling by using the procedures in Monitoring Amazon DocumentDB serverless.

The load can vary across the writer and readers in your cluster. The writer handles write operations, and performs all the data modifications to the cluster volume. Readers can process read-only requests.

Scaling is the operation that increases or decreases DocumentDB serverless capacity for your database. With DocumentDB serverless, each writer and reader has its own current capacity value, measured in DCUs. DocumentDB serverless scales a writer or reader up to a higher capacity when its current capacity is too low to handle the load. It scales the writer or reader down to a lower capacity when its current capacity is higher than needed.

DocumentDB serverless can increase capacity incrementally. When your workload demand begins to reach the current database capacity of a writer or reader, DocumentDB serverless increases the number of DCUs for that writer or reader. DocumentDB serverless scales capacity in the increments required to provide the best performance for the resources consumed. Scaling happens in increments as small as 0.5 DCUs. The larger the current capacity, the larger the scaling increment and thus the faster scaling can happen.

Because DocumentDB serverless scaling is so frequent, granular, and nondisruptive, it doesn't cause discrete events in the AWS Management Console. Instead, you can measure the Amazon CloudWatch metrics such as `serverlessDatabaseCapacity` and `DCUUtilization`, and track their minimum, maximum, and average values over time. To learn more about monitoring DocumentDB serverless, see Monitoring Amazon DocumentDB serverless.

Scaling up or down can be caused by the following:

- Memory utilization
- CPU utilization
- Network utilization
- Storage utilization

You can monitor these causes of scaling up/down on DocumentDB serverless instances. For more information, see Monitoring Amazon DocumentDB serverless.

You can choose to make a reader scale at the same time as the associated writer, or independently from the writer. You do so by specifying the promotion tier for that reader.

- DocumentDB serverless readers, in promotion tiers 0 and 1, scale at the same time as the writer. That scaling behavior makes readers in priority tiers 0 and 1 ideal for availability. That's because they are always sized to the right capacity to take over the workload from the writer in case of failover.
- Readers in promotion tiers 2–15 scale independently from the writer. Each reader remains within the minimum and maximum DCU values that you specified for your cluster. When a reader scales independently of the associated writer DB, it can become idle and scale down while the writer continues to process a high volume of transactions. It's still available as a failover target, if no other readers are available in lower promotion tiers. However, if it's promoted to be the writer, it might need to scale up to handle the full workload of the writer.

For details about viewing and changing promotion tiers of serverless instances, see Viewing and modifying the promotion tier of serverless readers.

DocumentDB serverless scaling can happen while database connections are open, while transactions are in process, etc. DocumentDB serverless doesn't wait for a quiet point to begin scaling. Scaling doesn't disrupt any database operations that are underway.

If your workload requires more read capacity than is available with a single writer and a single reader, you can add multiple DocumentDB serverless readers to the cluster. Each DocumentDB serverless reader can scale within the range of minimum and maximum capacity values that you specified for your cluster. You can use the cluster's reader endpoint to direct read-only sessions to the readers and reduce the load on the writer.

Whether DocumentDB serverless performs scaling, and how fast scaling occurs once it starts, also depends on the minimum and maximum DCU settings for the cluster. In addition, it depends on whether a reader is configured to scale along with the writer or independently from it. For details about the scaling configuration, see Amazon DocumentDB serverless scaling configuration.

Idle state (0.5 DCUs)

When Amazon DocumentDB serverless writers or readers are idle, DocumentDB serverless instances support scaling down to an idle state of 0.5 DCUs if the cluster's MinCapacity is configured to be 0.5.

In the idle state, the DocumentDB serverless instances do not have sufficient CPU compute capacity to support most production workloads, but are ready to quickly scale up to support a new workload. In a non-idle state, DocumentDB serverless instances typically require at least 1.0 - 2.5 DCUs. Therefore, when DocumentDB serverless instances scale up from an idle state to a non-idle state, they will scale up directly to 1.0 - 2.5 DCUs (or the value of MaxCapacity if it is lower).

In order to support scaling down to 0.5 DCUs when idle, instance limits are capped if the MinCapacity is configured to be less than or equal to 1.0 DCUs. For more information on how the limits are affected by the MinCapacity configuration, see Amazon DocumentDB serverless instance limits.

Requirements and limitations for DocumentDB serverless

Feature requirements

Region availability

For more information about AWS Regions where the Amazon DocumentDB serverless instance type is available, see the Serverless column in Supported instance classes by Region.

The following AWS CLI command can be used to verify the exact DocumentDB serverless instance options offered in a particular Region:

```
aws docdb describe-orderable-db-instance-options \
        --region my_region \
        --db-instance-class db.serverless \
        --engine docdb
```

Engine version availability

DocumentDB serverless is supported by Amazon DocumentDB 5.0 and 8.0 engine versions. It is not available on engine versions 3.6 or 4.0.

> **Note**
>
> DocumentDB serverless is supported only on newer patch versions of 5.0.0. Ensure that your cluster is updated to a recent engine patch version. For more information on maintaining Amazon DocumentDB clusters, see Maintaining Amazon DocumentDB

Cluster configuration

Before adding any Amazon DocumentDB serverless instances to an Amazon DocumentDB cluster, the cluster must also have the `ServerlessV2ScalingConfiguration` parameter set. This defines the capacity range of DocumentDB serverless instances in the cluster. For more information on scaling configuration, see Amazon DocumentDB serverless scaling configuration.

The available scaling range for a cluster is influenced by both its engine version and its platform version. For the capacity ranges of the DocumentDB serverless platform versions, see Amazon DocumentDB serverless scaling capacity.

Minimum scaling capacity range settings for certain Amazon DocumentDB features

Some Amazon DocumentDB features work with DocumentDB serverless, but might cause issues if your capacity range is lower than needed for the memory requirements for those features with your specific workload. In that case, your database might not perform as well as usual, or might encounter out-of-memory errors.

The following features require configuration of a higher `MinCapacity` and/or `MaxCapacity` value for best operation:

- Performance Insights
- Global clusters (applies only to the primary AWS Region)
- Serverless instance creation on a cluster with a large data volume

  This includes serverless instance creation as part of a cluster restore.

For recommendations about setting the appropriate capacity range (if you are using this feature), see Choosing the scaling capacity range for a DocumentDB serverless cluster. For troubleshooting information if your database encounters out-of-memory errors due to a misconfigured capacity range, see Avoiding out-of-memory errors.

Creating a cluster that uses Amazon DocumentDB serverless

Creating an Amazon DocumentDB serverless cluster

With Amazon DocumentDB serverless, your clusters are interchangeable with provisioned clusters. You can have clusters where some instances use serverless and some instances are provisioned.

Verify that your desired Region and engine version support DocumentDB serverless. See Requirements and limitations for DocumentDB serverless.

To create an Amazon DocumentDB cluster where you can add serverless instances, follow the same procedure as in Creating an Amazon DocumentDB cluster. The only difference is that the `ServerlessV2ScalingConfiguration` argument must also be provided.

The `ServerlessV2ScalingConfiguration` argument specifies the scaling capacity range of your DocumentDB serverless instances. It consists of the minimum and maximum DocumentDB capacity unit (DCU) values that apply to all the DocumentDB serverless instances in the cluster:

- The `MinCapacity` value specifies the minimum scaling capacity.
- The `MaxCapacity` value specifies the maximum scaling capacity.

For more information on scaling, see Amazon DocumentDB serverless scaling configuration.

Using the AWS Management Console

The following AWS Management Console configuration example shows how to create a DocumentDB serverless cluster.

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

   The **Clusters** table appears.
3. Choose **Create**.

   The **Create Amazon DocumentDB cluster** page appears.
4. On the Create Amazon DocumentDB cluster page, in the **Cluster type** section, choose **Instance-based cluster** (this is the default option).
5. In the **Cluster configuration** section:

   a. For **Cluster identifier**, enter a unique name, such as **`myserverlesscluster`**. Note that the console will change all cluster names into lower-case regardless of how they are entered.

   b. For **Engine version**, keep the default version, or choose another available version. For more information, see Amazon DocumentDB cluster settings.

6. In the **Cluster storage configuration** section, choose **Amazon DocumentDB Standard** (this is the default option).

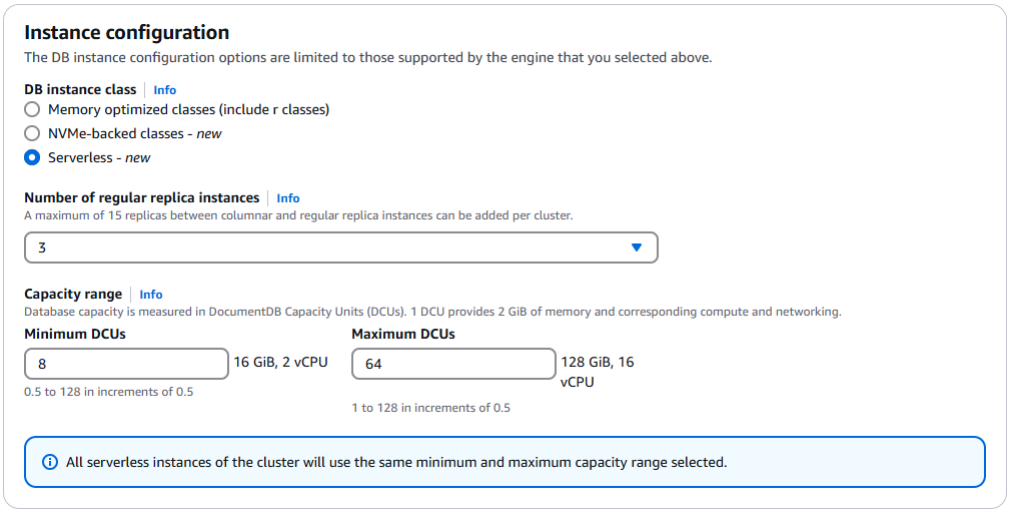
> **Note**
>
> The other option in this category is **Amazon DocumentDB I/O-Optimized**. To learn more about either option, see Amazon DocumentDB cluster storage configurations

7. In the **Instance configuration** section:

   a. For **DB instance class**, choose **Serverless**.

   b. For **Number of regular replica instances**, choose **3** (this is the default option).

   c. In the **Capacity range** section, leave the default values for **Minimum DCUs** and **Maximum DCUs**. For information on setting these parameters, see Amazon DocumentDB serverless instance limits.



8. In the **Connectivity** section, leave the default setting of **Don't connect to an EC2 compute resource**.

9. In the **Authentication** section, enter a username for the primary user, and then choose **Self managed**. Enter a password, then confirm it.

   If you instead chose **Managed in AWS Secrets Manager**, see Password management with Amazon DocumentDB and AWS Secrets Manager for more information.

10. Leave all other options as default and choose **Create cluster**.

Using the AWS CLI

In the following example, replace each *user input placeholder* with your own information or configuration parameters.

To create a cluster compatible with DocumentDB serverless instances using the AWS CLI, see Creating a cluster using the AWS CLI.

Include the following additional parameters in your `create-db-cluster` command:

```
--serverless-v2-scaling-configuration
     MinCapacity=minimum_capacity,MaxCapacity=maximum_capacity
```

Example:

```
aws docdb create-db-cluster \
      --db-cluster-identifier sample-cluster \
      --engine docdb \
      --engine-version 5.0.0 \
      --serverless-v2-scaling-configuration MinCapacity=0.5,MaxCapacity=16 \
      --master-username user-name \
      --master-user-password password
```

Adding an Amazon DocumentDB serverless instance

To add an DocumentDB serverless instance, follow the same procedure in Adding an Amazon DocumentDB instance to a cluster, making sure to specify db.serverless as the instance class.

Adding a serverless instance using the AWS Management Console.

To add an DocumentDB serverless instances using the console, see Adding an Amazon DocumentDB instance to a cluster and choose the **Using the AWS Management Console** tab.

Adding a serverless instance using the AWS CLI

To add an DocumentDB serverless instances using the AWS CLI, see Adding an Amazon DocumentDB instance to a cluster and choose the **Using the AWS CLI** tab.

Use the following instance class CLI argument:

```
--db-instance-class db.serverless
```

Example:

```
aws docdb create-db-instance \
      --db-cluster-identifier sample-cluster \
      --db-instance-identifier sample-instance \
      --db-instance-class db.serverless \
      --engine docdb
```

Migrating to Amazon DocumentDB serverless

Topics

- Migrating existing DocumentDB clusters to serverless
- Migrating from MongoDB to DocumentDB serverless

Migrating existing DocumentDB clusters to serverless

Upgrading a cluster's engine version

If your provisioned cluster is running a lower engine version that does not support DocumentDB serverless, you need to first upgrade the cluster to a supported engine version. For more information, see Amazon DocumentDB in-place major version upgrade.

Migrating a provisioned cluster to DocumentDB serverless

To switch a provisioned cluster to use DocumentDB serverless, follow these steps:

1. Check if the provisioned cluster's engine version needs to be upgraded to be used with DocumentDB serverless. See Requirements and limitations for DocumentDB serverless.

   > **Note**
   >
   > If the provisioned cluster is running an engine version that isn't available for DocumentDB serverless, upgrade the engine version of the cluster. See Amazon DocumentDB in-place major version upgrade.

2. Configure the scaling configuration for the cluster. For details on choosing the scaling configuration, see Choosing the scaling capacity range for a DocumentDB serverless cluster.

To modify the scaling configuration for a cluster, see Viewing and modifying a cluster's scaling capacity range configuration.

3. Configure any other cluster properties to account for the DocumentDB serverless requirements and limitations from Requirements and limitations for DocumentDB serverless.
4. Add one or more DocumentDB serverless instances to the cluster. Follow the procedure in Adding an Amazon DocumentDB serverless instance.

> **Note**
>
> In some cases, you might already have one or more provisioned reader instances in the cluster. If so, you can choose to convert one of the readers to a DocumentDB serverless instance instead of creating a new instance. To do so, follow the procedure in Changing an instance's class.

5. (Optional) Perform a failover operation to make a DocumentDB serverless instance the cluster writer. See Amazon DocumentDB failover.
6. (Optional) Convert any remaining provisioned Amazon DocumentDB instances to DocumentDB serverless instances (see Changing an instance's class) or remove them from the cluster (see Deleting an Amazon DocumentDB instance).

Using the AWS Management Console

The following AWS Management Console configuration example shows the migration process using an Amazon DocumentDB provisioned cluster that is running Amazon DocumentDB 5.0.0, which does not require an engine version upgrade to start using DocumentDB serverless. The cluster is named `sample`, and starts with three provisioned instances named `sample`, `sample2`, and `sample3`. In this example, these three instances will be replaced by three serverless instances.

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

The **Clusters** table appears.

3. In the **Clusters** table, select the checkbox of the cluster to which you want to add a serverless instance.
4. Choose **Actions**, and then choose **Add instances**.
5. In the **Add instances** dialog box, select **Serverless** in the **DB instance class** section for each new serverless instance you want to create.
6. For **Serverless capacity settings**, set the scaling configuration based on the capacity description in the dialog box.
7. (Optional) To add another instance, choose **Add instance**. Continue adding instances until you have reached the desired number of new instances.

   In this example, three new serverless instances are created.
8. Choose **Create**.

   It takes several minutes to create an instance. You can use the console or the AWS CLI to view the instance's status. For more information, see Monitoring an Amazon DocumentDB cluster's status.
9. Returning to the **Clusters** table, select the checkboxes of all three of the original provisioned instances.
10. Choose **Actions**, and then choose **Delete**.

During the deletion, a failover is automatically performed to promote one of the remaining instances to be the writer. After a few minutes, the delete process is completed. The existing cluster now contains three DocumentDB serverless instances (as defined in the **Size** column).

Using the AWS CLI

The following AWS CLI configuration example shows the migration process using an Amazon DocumentDB provisioned cluster that is running Amazon DocumentDB 5.0.0, which does not

require an engine version upgrade to start using DocumentDB serverless. The cluster is named `sample`, and starts with three provisioned instances named `sample`, `sample2`, and `sample3`. In this example, these three instances will be replaced by three serverless instances. The cluster is named `sample-cluster`, and starts with two provisioned instances named `sample-provisioned-instance-1` and `sample-provisioned-instance-2`, a writer instance and a reader instance.

In the following example, replace each *`user input placeholder`* with your own information or configuration parameters.

Use the `aws docdb describe-db-clusters` operation to determine the status of a cluster. The following code finds the status of the cluster `sample-cluster` and outputs the results in a table:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].DBClusterMembers' \
    --output table
```

```
-------------------------------------------------------------------------------------------
|                                        DescribeDBClusters
                  |
+---------------------------------+---------------------------------
+------------------+----------------+
|  DBClusterParameterGroupStatus |      DBInstanceIdentifier       | IsClusterWriter
  | PromotionTier  |
+---------------------------------+---------------------------------
+------------------+----------------+
|  in-sync                       |  sample-provisioned-instance-2  |  False
  |  1             |
|  in-sync                       |  sample-provisioned-instance-1  |  True
  |  1             |
+---------------------------------+---------------------------------
+------------------+----------------+
```

Add the scaling configuration for the cluster:

```
aws docdb modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --serverless-v2-scaling-configuration MinCapacity=0.5,MaxCapacity=16
```

Add the serverless instances. In this example, new serverless instances named `sample-serverless-instance-1` and `sample-serverless-instance-2` are added:

```
aws docdb create-db-instance \
    --db-cluster-identifier sample-cluster \
    --db-instance-identifier sample-serverless-instance-1 \
    --db-instance-class db.serverless \
    --engine docdb

aws docdb create-db-instance \
    --db-cluster-identifier sample-cluster \
    --db-instance-identifier sample-serverless-instance-2 \
    --db-instance-class db.serverless \
    --engine docdb
```

Enter the following to wait for the serverless instances to be available before proceeding:

```
aws docdb wait db-instance-available \
    --db-instance-identifier sample-serverless-instance-1

aws docdb wait db-instance-available \
    --db-instance-identifier sample-serverless-instance-2
```

Perform a failover to make the new `sample-serverless-instance-1` instance the cluster writer:

```
aws docdb failover-db-cluster \
    --db-cluster-identifier sample-cluster \
    --target-db-instance-identifier sample-serverless-instance-1
```

The failover takes a few seconds to complete, after which sample-serverless-instance-1 becomes the cluster writer. Verify this with the following input:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[*].DBClusterMembers' \
    --output table
```

```
-------------------------------------------------------------------------------------------
|                                        DescribeDBClusters
                  |
```

```
+--------------------------------+---------------------------------
+------------------+----------------+
|  DBClusterParameterGroupStatus |      DBInstanceIdentifier       | IsClusterWriter
  | PromotionTier  |
+--------------------------------+---------------------------------
+------------------+----------------+
|  in-sync                       |  sample-provisioned-instance-2  |  False
  |  1             |
|  in-sync                       |  sample-provisioned-instance-1  |  False
  |  1             |
|  in-sync                       |  sample-serverless-instance-2   |  False
  |  1             |
|  in-sync                       |  sample-serverless-instance-1   |  True
  |  1             |
+--------------------------------+---------------------------------
+------------------+----------------+
```

Finally, delete the original provisioned instances:

```
aws docdb delete-db-instance \
    --db-instance-identifier sample-provisioned-instance-1

aws docdb delete-db-instance \
    --db-instance-identifier sample-provisioned-instance-2
```

Migrating from MongoDB to DocumentDB serverless

You can migrate your MongoDB databases to DocumentDB serverless, just as with provisioned Amazon DocumentDB. For more information, see Migrating and upgrading Amazon DocumentDB.

Managing Amazon DocumentDB serverless

Viewing and modifying a cluster's scaling capacity range configuration

The `ServerlessV2ScalingConfiguration` argument specifies the scaling capacity range of your DocumentDB serverless instances. It consists of the minimum and maximum DocumentDB capacity unit (DCU) values that apply to all the DocumentDB serverless instances in the cluster.

- **`MinCapacity`** — The minimum scaling capacity of any DocumentDB serverless instances in the cluster.

- **`MaxCapacity`** — The maximum scaling capacity of any DocumentDB serverless instances in the cluster.

> **Note**
>
> The following scaling configuration modifications require an instance restart to reflect the new `MinCapacity` and `MaxCapacity` values:
>
> - Any changes to the `MaxCapacity` value
> - Changing the `MinCapacity` value to 1.0 or lower from a higher value
> - Changing the `MinCapacity` value to greater than 1.0 from a lower value

For more information on scaling configuration and how to select appropriate scaling capacity limits, see Amazon DocumentDB serverless scaling configuration.

Using the AWS Management Console

The following AWS Management Console configuration example shows how to view and edit a DocumentDB serverless cluster's scaling configuration settings.

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

   The **Clusters** table appears.
3. In the **Clusters** table, select the checkbox of the cluster to which you want to modify the scaling capacity.
4. Choose **Actions**, and then choose **Modify**.

   The **Modify cluster** dialog box appears.

5. Locate the **Serverless capacity setting** section and set the scaling configuration (**Capacity range**) based on the capacity description in the dialog box.

   For more information about scaling and capacity ranges, see Amazon DocumentDB serverless scaling configuration.

6. Choose **Continue**.
7. For **Scheduling of modifications**, choose **Apply immediately**.
8. Choose **Modify cluster**.
9. Once the modification has completed, each serverless instance should be rebooted. To minimize writer unavailability, perform the following sequence of operations:

   a. Reboot each serverless reader instance.

      i. Select the reader instance, choose **Actions**, and then **Reboot**.

      ii. Wait for the instance status to return to **Available**.

   b. Perform a failover to a rebooted serverless instance.

      i. Select the cluster, choose **Actions**, and then **Failover**.

      ii. Wait for the failover operation to complete.

   c. Reboot the remaining serverless instance.

      i. Select the remaining instance, choose **Actions**, and then **Reboot**.

      ii. Wait for the instance status to return to **Available**.

Using the AWS CLI

The following AWS CLI configuration example shows the current scaling configuration.

In the following example, replace each *`user input placeholder`* with your own information or configuration parameters.

The cluster's current scaling configuration can be viewed using the `describe-db-clusters` AWS CLI command:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[0].ServerlessV2ScalingConfiguration'
```

The following is the output of this command:

```
{
    "MinCapacity": 0.5,
    "MaxCapacity": 16.0
}
```

The cluster's scaling configuration can be modified using the `modify-db-cluster` command:

```
aws docdb modify-db-cluster \
    --db-cluster-identifier sample-cluster \
    --serverless-v2-scaling-configuration MinCapacity=0.5,MaxCapacity=32
```

Once complete, each serverless instance should be rebooted. To minimize writer unavailability, we can perform the following sequence of operations:

```
aws docdb reboot-db-instance \
    --db-instance-identifier sample-serverless-instance-reader

aws docdb wait db-instance-available \
    --db-instance-identifier sample-serverless-instance-reader

aws docdb failover-db-cluster \
   --db-cluster-identifier sample-cluster \
   --target-db-instance-identifier sample-serverless-instance-reader

aws docdb reboot-db-instance \
    --db-instance-identifier sample-serverless-instance-writer

aws docdb wait db-instance-available \
    --db-instance-identifier sample-serverless-instance-writer

aws docdb failover-db-cluster \
   --db-cluster-identifier sample-cluster \
   --target-db-instance-identifier sample-serverless-instance-writer
```

Viewing and modifying the promotion tier of serverless readers

For clusters containing multiple DocumentDB serverless instances or a mixture of provisioned and serverless instances, pay attention to the promotion tier setting for each serverless instance. This setting controls more behavior for serverless instances than for provisioned instances.

For provisioned instances, the choice of tier 0–15 determines only the order in which Amazon DocumentDB chooses which reader instance to promote to the writer during a failover operation. However, for serverless instances, the tier number also determines whether the instance scales up to match the capacity of the writer instance or scales independently based on its own workload. Serverless reader instances in tier 0 or 1 are kept at a minimum capacity at least as high as the writer instance. That way, they are ready to take over from the writer instance in case of a failover. If the writer instance is a provisioned instance, Amazon DocumentDB estimates the equivalent DocumentDB serverless capacity. It uses that estimate as the minimum capacity for the serverless reader instance.

DocumentDB serverless reader instances in tiers 2–15 don't have the same constraint on their minimum capacity. When they are idle, they can scale down to the minimum DocumentDB capacity unit (DCU) value specified in the cluster's capacity range.

Using the AWS Management Console

The following AWS Management Console configuration example shows how to view and modify a DocumentDB serverless instance reader's promotional tier settings.

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.

   The promotion tiers of each instance is displayed in the **Promotion tier** column in the AWS Management Console.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

   The **Clusters** table appears.

3. In the **Clusters** table, select the checkbox of the instance to which you want to modify the promotion tier.
4. Choose **Actions**, and then choose **Modify**.

   The **Modify instance** dialog box appears.
5. Locate the **Failover** section and set the **Promotion tier** to the desired level.
6. Choose **Continue**.
7. For **Scheduling of modifications**, choose **Apply immediately**.
8. Choose **Modify instance**.

Using the AWS CLI

The following AWS CLI configuration example shows the current scaling configuration.

In the following example, replace each *`user input placeholder`* with your own information or configuration parameters.

The promotion tiers of all instances in a cluster can be viewed using the `describe-db-clusters` AWS CLI command:

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[0].DBClusterMembers' \
    --output table
```

The following is the output of this command:

```
-------------------------------------------------------------------------------------------
|                                          DescribeDBClusters
                  |
+---------------------------------+---------------------------------
+------------------+----------------+
|  DBClusterParameterGroupStatus |      DBInstanceIdentifier       | IsClusterWriter
  | PromotionTier  |
+---------------------------------+---------------------------------
+------------------+----------------+
|  in-sync                       |  sample-serverless-instance-2   |  False
  |  1             |
```

```
|  in-sync                      |  sample-serverless-instance-1   |  True
  |  1             |
+-------------------------------+---------------------------------
+-----------------+---------------+
```

The promotion tier of a specific instance can be modified using the `modify-db-instance` command:

```
aws docdb modify-db-instance \
    --db-instance-identifier sample-serverless-instance-2 \
    --promotion-tier 3
```

Checking and upgrading the platform version of a serverless cluster

Important

After you upgrade to a newer platform version, you can't downgrade to a previous version.

Checking the platform version for an existing cluster

Using the AWS Management Console

The platform version is displayed on the **Configuration** tab of the cluster details page and the serverless instance details page.

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

   Tip

   If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

3. In the list of clusters, choose the name of your serverless cluster.
4. Choose the **Configuration** tab. The **Platform version** field shows the platform version of the cluster.

Using the AWS CLI

You can use the `describe-db-clusters` AWS CLI command to check the platform version for an existing cluster.

In the following example, replace each *`user input placeholder`* with your own information or configuration parameters.

```
aws docdb describe-db-clusters \
    --db-cluster-identifier sample-cluster \
    --query 'DBClusters[0].ServerlessV2PlatformVersion'
```

The following is the output of this command:

```
"4"
```

Checking the default platform version

You can use the `describe-serverless-v2-platform-versions` AWS CLI command to check the details of a platform version, or to check the default platform version that you get when you create a new cluster.

```
aws docdb describe-serverless-v2-platform-versions \
    --engine docdb \
    --default-only
```

The following is the output of this command:

```
{
    "ServerlessV2PlatformVersions": [
        {
            "ServerlessV2PlatformVersion": "4",
            "ServerlessV2PlatformVersionDescription": "Version 4 offering scaling up to
 256 DCUs, and performance improvement up to 35% compared to version 3",
            "Engine": "docdb",
            "ServerlessV2FeaturesSupport": {
                "MinCapacity": 0.5,
                "MaxCapacity": 256.0
            },
            "Status": "enabled",
            "IsDefault": true
```

```
        }
    ]
}
```

Checking for pending platform version upgrades

You can use the `describe-pending-maintenance-actions` AWS CLI command to check whether there are any pending platform version upgrades for your DocumentDB serverless clusters. A pending platform version upgrade appears as the `serverless-platform-version-update` maintenance action.

To check for pending upgrades on a specific cluster, use the `--resource-identifier` parameter to specify the Amazon Resource Name (ARN) of your cluster:

```
aws docdb describe-pending-maintenance-actions \
    --resource-identifier arn:aws:rds:us-east-1:111122223333:cluster:sample-cluster
```

Output from this operation looks something like the following (JSON format).

```
{
    "PendingMaintenanceActions": [
        {
            "ResourceIdentifier": "arn:aws:rds:us-east-1:111122223333:cluster:sample-
cluster",
            "PendingMaintenanceActionDetails": [
                {
                    "Action": "serverless-platform-version-update",
                    "Description": "Upgrade to platform version 4"
                }
            ]
        }
    ]
}
```

Scheduling a platform version upgrade

To schedule when a pending platform version upgrade occurs, use the `apply-pending-maintenance-action` AWS CLI command after you identify the pending upgrade.

To schedule a platform version upgrade, specify the following parameters:

- `--resource-identifier` — The ARN of your DocumentDB serverless cluster.

- `--apply-action` — Use `serverless-platform-version-update` to specify a platform version upgrade.
- `--opt-in-type` — Choose when to apply the upgrade. Use one of the following values:
  - `immediate` — Apply the upgrade immediately.
  - `next-maintenance` — Apply the upgrade during your next scheduled maintenance window.

Example

For Linux, macOS, or Unix:

```
aws docdb apply-pending-maintenance-action \
    --resource-identifier arn:aws:rds:us-east-1:111122223333:cluster:sample-cluster \
    --apply-action serverless-platform-version-update \
    --opt-in-type immediate
```

For Windows:

```
aws docdb apply-pending-maintenance-action ^
    --resource-identifier arn:aws:rds:us-east-1:111122223333:cluster:sample-cluster ^
    --apply-action serverless-platform-version-update ^
    --opt-in-type immediate
```

For more information about maintenance actions, see Maintaining Amazon DocumentDB.

Amazon DocumentDB serverless instance limits

For DocumentDB serverless instances, the following limits per instance are dependent upon the instance's current scaling capacity:

- Instance Memory (GiB)
- Connections (all)
- Cursor Limit
- Open Transactions
- Connections (active)

The following tables describe how the per-instance limits of serverless instances scale in relation to the instance's current scaling capacity.

Note that the limit values are different depending on whether the cluster's `MinCapacity` scaling configuration has been set to greater than two (2). If `MinCapacity` is set to less than or equal to 2, then the limit values for cursor limit, open transactions, and connections (active) are capped to a lower maximum value. (The technical reason for the cap is to support scaling down to an "idle" state. For more information, see Idle state (0.5 DCUs).

db.serverless instance limits if MinCapacity ≤1

Current Capacity (DCU)	Instance Memory (GiB)	Connections (all)	Cursor Limit	Open Transactions	Connections (active)
0.5	1	250	6	12	39
1	2	500	12	24	79
2	4	1000	24	48	173
4	8	2000	48	96	416
8	16	4000	96	192	1071
16	32	8000	132	264	1550
32	64	16000	132	264	1550
64	128	32000	132	264	1550
128	256	60000	132	264	1550
256	512	60000	132	264	1550

db.serverless instance limits if MinCapacity >1

Current Capacity (DCU)	Instance Memory (GiB)	Connections (all)	Cursor Limit	Open Transactions	Connections (active)
1.5	3	750	18	36	124

Current Capacity (DCU)	Instance Memory (GiB)	Connections (all)	Cursor Limit	Open Transactions	Connections (active)
2	4	1000	24	48	173
4	8	2000	48	96	416
8	16	4000	96	192	1071
16	32	8000	192	384	2709
32	64	16000	384	768	4500
64	128	32000	768	1536	4500
128	256	60000	1536	3072	4500
256	512	60000	3072	6144	4500

You can monitor and alarm on the per instance limits using the following CloudWatch metrics. For more on Amazon DocumentDB CloudWatch metrics, see Monitoring Amazon DocumentDB with CloudWatch.

Resource	CloudWatch limit metric	CloudWatch usage metric (1-min. max)	CloudWatch usage metric
Instance Memory	-	-	FreeableMemory
Connections (all)	DatabaseConnectionsLimit	DatabaseConnectionsMax	DatabaseConnections
Cursors	DatabaseCursorsLimit	DatabaseCursorsMax	DatabaseCursors
Transactions	TransactionsOpenLimit	TransactionsOpenMax	TransactionsOpen

Amazon DocumentDB serverless scaling configuration

Topics

- Choosing the scaling capacity range for a DocumentDB serverless cluster
- Choosing the MinCapacity setting for a DocumentDB serverless cluster
- Choosing the MaxCapacity setting for a DocumentDB serverless cluster
- Avoiding out-of-memory errors
- Why is my serverless instance not scaling down?

Choosing the scaling capacity range for a DocumentDB serverless cluster

Before adding any DocumentDB serverless instances to an Amazon DocumentDB cluster, the cluster must also have the `ServerlessV2ScalingConfiguration` parameter set.

The `ServerlessV2ScalingConfiguration` parameter consists of two values which define the serverless scaling capacity range of any serverless instance in the cluster:

- **`MinCapacity`** — The minimum scaling capacity of any DocumentDB serverless instances in the cluster.
- **`MaxCapacity`** — The maximum scaling capacity of any DocumentDB serverless instances in the cluster.

Choosing the `MinCapacity` setting for a DocumentDB serverless cluster

It's tempting to always choose 0.5 for `MinCapacity`. That value allows the instance to scale down to the smallest capacity when it's completely idle, while remaining active. However, depending on how you use that cluster and the other settings that you configure, a different minimum capacity might be the most effective. Consider the following factors when choosing the minimum capacity setting:

- The scaling rate for a DocumentDB serverless instance depends on its current capacity. The higher the current capacity, the faster it can scale up. If you need the instance to quickly scale up

to a very high capacity, consider setting the minimum capacity to a value where the scaling rate meets your requirement.

- If you typically modify the instance class of your instances in anticipation of especially high or low workload, you can use that experience to make a rough estimate of the equivalent DocumentDB serverless capacity range. To determine the memory size of a provisioned Amazon DocumentDB instance type, see Instance quotas.

  For example, suppose that you use the `db.r6g.xlarge` instance class when your cluster has a low workload. That instance class has 32 GiB of memory. Thus, you can specify a `MinCapacity` of 16 to set up a serverless instance that can scale down to approximately that same capacity. That's because each DCU corresponds to approximately 2 GiB of memory. You might specify a somewhat lower value to let the instance scale down further in case your `db.r6g.xlarge` instance was sometimes underutilized.

- If your application works most efficiently when the instances have a certain amount of data in the buffer cache, consider specifying a minimum DCU setting where the memory is large enough to hold the frequently accessed data. Otherwise, some data is evicted from the buffer cache when the serverless instances scale down to a lower memory size. Then when the instances scale back up, the information is read back into the buffer cache over time. If the amount of I/O to bring the data back into the buffer cache is substantial, it might be more effective to choose a higher minimum DCU value. For more information, see Instance sizing.
- If your DocumentDB serverless instances run most of the time at a particular capacity, consider specifying a minimum capacity setting that's lower than that baseline, but not too much lower. Serverless instances can most effectively estimate how much and how fast to scale up when the current capacity isn't drastically lower than the required capacity.
- If your provisioned workload has memory requirements that are too high for small instance classes such as T3 or T4g, choose a minimum DCU setting that provides memory comparable to an R5 or R6g instance.
- Use the following minimum `MinCapacity` for the specified features (subject to change):
  - Performance Insights — 2 DCUs
  - Global clusters — 8 DCUs (applies only to the primary AWS Region)
- In Amazon DocumentDB, replication occurs at the storage layer, so reader capacity doesn't directly affect replication. However, for DocumentDB serverless reader instances that scale independently, make sure that the minimum capacity is sufficient to handle workloads during write-intensive periods to avoid query latency. If reader instances in promotion tiers 2–15 experience performance issues, consider increasing the cluster's minimum capacity. For details on

changing whether reader instances scale along with the writer or independently, see Viewing and modifying the promotion tier of serverless readers.

If you have a cluster with DocumentDB serverless reader instances, the readers don't scale along with the writer instance when the promotion tier of the readers isn't 0 or 1. In that case, setting a low minimum capacity can result in excessive replication lag. That's because the readers might not have enough capacity to apply changes from the writer when the database is busy. Set the minimum capacity to a value that represents a comparable amount of memory and CPU to the writer instance.

- The time it takes for a DocumentDB serverless instance to scale from its minimum capacity to its maximum capacity depends on the difference between its minimum and maximum DCU values. When the current capacity of the instance is large, DocumentDB serverless scales up in larger increments than when the instance starts from a small capacity. Thus, if you specify a relatively large maximum capacity and the instance spends most of its time near that capacity, consider increasing the minimum DCU setting. That way, an idle instance can scale back up to maximum capacity more quickly.
- Certain instance limits are determined by the serverless instance's current capacity, such as connections limit, cursor limit, and open transactions limit. If the instance's current capacity is small, then the limits will correspondingly also be small. If these limits are an issue when your serverless instance is scaled down to its `MinCapacity` value, then consider increasing `MinCapacity` to a higher value. For more information, see Amazon DocumentDB serverless instance limits.
- Furthermore, certain instance limits are capped at a lower maximum value if the `MinCapacity` is set to less than or equal to 1.0 DCUs, such as active connections limit, cursor limit, and open transactions limit. If these capped limits are insufficient for your workload, use a `MinCapacity` value of at least 1.5 DCUs. For more information, see Amazon DocumentDB serverless instance limits.

For instructions on how to modify a cluster's scaling configuration, see Managing Amazon DocumentDB serverless.

Choosing the `MaxCapacity` setting for a DocumentDB serverless cluster

It's tempting to always choose some high value for the maximum DocumentDB serverless capacity setting. A large maximum capacity allows the instance to scale up the most when it's running an

intensive workload. A low value avoids the possibility of unexpected charges. Depending on how you use that cluster and the other settings that you configure, the most effective value might be higher or lower than you originally thought. Consider the following factors when choosing the maximum capacity setting:

- The maximum capacity must be at least as high as the minimum capacity. You can set the minimum and maximum capacity to be identical. However, in that case the capacity never scales up or down. Thus, using identical values for minimum and maximum capacity isn't appropriate outside of testing situations.
- The maximum capacity must be at least 1.0 DCUs and must be at most 256 DCUs.
- Monitor the scaling and resource usage of your serverless instances. If your serverless instance is frequently scaling to maximum capacity and hitting resource constraints (for example, when the `DCUUtilization` metric is at 100.0), select a higher `MaxCapacity` value. For more information, see Monitoring Amazon DocumentDB serverless.
- If you typically modify the instance class of your provisioned instances in anticipation of especially high or low workload, you can use that experience to estimate the equivalent DocumentDB serverless capacity range. To determine the memory size of provisioned Amazon DocumentDB instances, see Instance quotas.

  For example, suppose that you use the `db.r6g.4xlarge` instance class when your cluster has a high workload. That instance class has 128 GiB of memory. Thus, you can specify a maximum DCU setting of 64 to set up a serverless instance that can scale up to approximately that same capacity. That's because each DCU corresponds to approximately 2 GiB of memory. You might specify a somewhat higher value to let the instance scale up farther in case your `db.r6g.4xlarge` instance sometimes doesn't have enough capacity to handle the workload effectively.
- If you have a budgetary cap on your database usage, choose a value that stays within that cap even if all your serverless instances run at maximum capacity all the time. Remember that when you have n serverless instances in your cluster, the theoretical maximum serverless capacity that the cluster can consume at any moment is n times the maximum DCU setting for the cluster. (The actual amount consumed might be less, for example if some readers scale independently from the writer.)
- If you make use of serverless reader instances to offload some of the read-only workload from the writer instance, you might be able to choose a lower maximum capacity setting. You do this to reflect that each reader instance doesn't need to scale as high as if the cluster contains only a single instance.

- Suppose that you want to protect against excessive usage due to misconfigured database parameters or inefficient queries in your application. In that case, you might avoid accidental overuse by choosing a maximum capacity setting that's lower than the absolute highest that you could set.
- If spikes due to real user activity are rare but do happen, you can take those occasions into account when choosing the maximum capacity setting. If the priority is for the application to keep running with full performance and scalability, you can specify a maximum capacity setting that's higher than you observe in normal usage. If it's OK for the application to run with reduced throughput during very extreme spikes in activity, you can choose a slightly lower maximum capacity setting. Make sure that you choose a setting that still has enough memory and CPU resources to keep the application running.
- If you turn on settings in your cluster that increase the memory usage for each instance, take that memory into account when deciding on the maximum DCU value. Such settings include those for Performance Insights and global clusters. Make sure that the maximum DCU value allows the serverless instances to scale up enough to handle the workload when those feature are being used. For information about troubleshooting problems caused by the combination of a low maximum DCU setting and Amazon DocumentDB features that impose memory overhead, see Avoiding out-of-memory errors (below).
- Use the following minimum `MaxCapacity` for the specified features (subject to change):
  - Serverless instance creation on a cluster with a large data volume – 2 DCUs (this includes serverless instance creation as part of a cluster restore.)
- Certain instance limits are determined by the instance's current capacity, such as connections limit, cursor limit, and open transactions limit. When choosing the `MaxCapacity` value for your workload, make sure to keep these instance limits in mind to avoid being bottlenecked by one of these limits. For more information, see Amazon DocumentDB serverless instance limits.

For instructions on how to modify a cluster's scaling configuration, see Managing Amazon DocumentDB serverless.

Avoiding out-of-memory errors

If one of your DocumentDB serverless instances consistently reaches the limit of its maximum capacity, Amazon DocumentDB indicates this condition by setting the instance to a status of **incompatible-parameters**. While the instance has the **incompatible-parameters** status, some operations are blocked. For example, you can't upgrade the engine version. For more information

on the status of an Amazon DocumentDB instance, see Monitoring an Amazon DocumentDB instance's status.

Typically, your instance goes into this status when it restarts frequently due to out-of-memory errors. Amazon DocumentDB records an event when this type of restart happens. To view resource events, see Viewing Amazon DocumentDB events. Unusually high memory usage can happen because of overhead from turning on settings such as Performance Insights. It can also come from a heavy workload on your instance or from managing the metadata associated with a large number of schema objects.

If the memory pressure becomes lower so that the instance doesn't reach its maximum capacity very often, Amazon DocumentDB automatically changes the instance status back to available.

To recover from this condition, you can take some or all of the following actions:

- Increase the lower limit on capacity for serverless instances by changing the minimum DocumentDB capacity unit (DCU) value for the cluster. Doing so avoids issues where an idle database scales down to a capacity with less memory than is needed for the features that are turned on in your cluster. After changing the DCU settings for the cluster, reboot the serverless instance. Doing so evaluates whether Amazon DocumentDB can reset the status back to available.
- Increase the upper limit on capacity for serverless instances by changing the maximum DCU value for the cluster. Doing so avoids issues where a busy database can't scale up to a capacity with enough memory for the features that are turned on in your cluster and the database workload. After changing the DCU settings for the cluster, reboot the serverless instance. Doing so evaluates whether Amazon DocumentDB can reset the status back to available.
- Turn off configuration settings that require memory overhead. For example, suppose that you have a feature such as Performance Insights turned on but don't use it. If so, you can turn it off. Or you can adjust the minimum and maximum capacity values for the cluster higher to account for the memory used by those types of features. For guidelines about choosing minimum and maximum capacity settings, see Choosing the scaling capacity range for a DocumentDB serverless cluster.
- Reduce the workload on the instance. For example, you can add reader instances to the cluster to spread the load from read-only queries across more instances.

Why is my serverless instance not scaling down?

In some cases, DocumentDB serverless doesn't scale down to the minimum capacity, even with no load on the database. This can happen for the following reasons:

- Performance Insights can increase resource usage and prevent the database from scaling down to minimum capacity. These features include the following:
- If a reader instance isn't scaling down to the minimum and stays at the same or higher capacity than the writer instance, then check the priority tier of the reader instance. DocumentDB serverless reader instances in tier 0 or 1 are kept at a minimum capacity at least as high as the writer instance. Change the priority tier of the reader to 2 or higher so that it scales up and down independently of the writer. For more information, see Amazon DocumentDB serverless scaling.
- Heavy database workloads can increase resource usage.
- Large database volumes can increase resource usage. Amazon DocumentDB uses memory and CPU resources for cluster management. Amazon DocumentDB requires more CPU and memory to manage clusters with larger database volumes. If your cluster's minimum capacity is less than the minimum required for cluster management, your cluster won't scale down to the minimum capacity.
- Background maintenance activity can periodically increase resource usage.

If the database still doesn't scale down to the minimum capacity configured, stop and restart the database to reclaim any memory fragments that might have built up over time.

> **Note**
>
> Stopping and starting a database results in downtime. Only do this when necessary.

Monitoring Amazon DocumentDB serverless

To learn more about monitoring in Amazon DocumentDB, see Monitoring and logging in Amazon DocumentDB.

Topics

- Out of Memory: incompatible-parameters status
- Amazon CloudWatch metrics for DocumentDB serverless

- Monitoring DocumentDB serverless performance with Performance Insights

Out of Memory: incompatible-parameters status

If one of your serverless instances consistently reaches the limit of its maximum capacity, Amazon DocumentDB indicates this condition by setting the instance to a status of incompatible-parameters. For more information, see Avoiding out-of-memory errors.

Amazon CloudWatch metrics for DocumentDB serverless

To learn more about using CloudWatch with Amazon DocumentDB, see Monitoring Amazon DocumentDB with CloudWatch.

You can view your serverless instances in CloudWatch to monitor the capacity consumed by each instance with the `ServerlessDatabaseCapacity` metric. You can also monitor all of the standard DocumentDB CloudWatch metrics, such as `DatabaseConnections` and Queries. For the full list of CloudWatch metrics that you can monitor for Amazon DocumentDB, see Monitoring Amazon DocumentDB with CloudWatch. The following CloudWatch instance-level metrics are important to monitor for you to understand how your DocumentDB serverless instances are scaling up and down. All of these metrics are calculated every second. That way, you can monitor the current status of your serverless instances. You can set alarms to notify you if any serverless instance approaches a threshold for metrics related to capacity. You can determine if the minimum and maximum capacity settings are appropriate, or if you need to adjust them. You can determine where to focus your efforts for optimizing the efficiency of your database.

- **`ServerlessDatabaseCapacity`** — As an instance-level metric, it reports the number of DCUs represented by the current instance capacity. As a cluster-level metric, it represents the average of the `ServerlessDatabaseCapacity` values of all the DocumentDB serverless instances in the cluster.
- **`DCUUtilization.`** — This metric is new in DocumentDB serverless. This value is represented as a percentage. It's calculated as the value of the `ServerlessDatabaseCapacity` metric divided by the maximum DCU value of the cluster. Consider the following guidelines to interpret this metric and take action:
  - If this metric approaches a value of `100.0`, the instance has scaled up as high as it can. Consider increasing the maximum DCU setting for the cluster. That way, both writer and reader instances can scale to a higher capacity.

- Suppose that a read-only workload causes a reader instance to approach a `DCUUtilization` of `100.0`, while the writer instance isn't close to its maximum capacity. In this case, consider adding additional reader instances to the cluster. That way, you can spread the read-only part of the workload across more instances, reducing the load on each reader instance.
- Suppose that you are running a production application, where performance and scalability are the primary considerations. In this case, you can set the maximum DCU value for the cluster to a high number. Your goal is for the `DCUUtilization` metric to always be below 100.0. With a high maximum DCU value, you can be confident that there's enough room in case there are unexpected spikes in database activity. You are only charged for the database capacity that's actually consumed.

- **`CPUUtilization`** — This metric is interpreted differently in DocumentDB serverless than in provisioned instances. For DocumentDB serverless, this value is a percentage that's calculated as the amount of CPU currently being used, divided by the CPU capacity that's available under the maximum DCU value of the cluster. Amazon DocumentDB monitors this value automatically and scales up your serverless instance when the instance consistently uses a high proportion of its CPU capacity.

  If this metric approaches a value of `100.0`, the instance has reached its maximum CPU capacity. Consider increasing the maximum DCU setting for the cluster. If this metric approaches a value of `100.0` on a reader instance, consider adding additional reader instances to the cluster. That way, you can spread the read-only part of the workload spread across more instances, reducing the load on each reader instance.

- **`FreeableMemory`** — This value represents the amount of unused memory that is available when the DocumentDB serverless instance is scaled to its maximum capacity. For every DCU that the current capacity is below the maximum capacity, this value increases by approximately 2 GiB. Thus, this metric doesn't approach zero until the instance is scaled up as high as it can.

  If this metric approaches a value of zero, the instance has scaled up as much as it can and is nearing the limit of its available memory. Consider increasing the maximum DCU setting for the cluster. If this metric approaches a value of zero on a reader instance, consider adding additional reader instances to the cluster. That way, the read-only part of the workload can be spread across more instances, reducing the memory usage on each reader instance.

- **`TempStorageIops`** — The number of IOPS done on local storage attached to the instance. It includes the IOPS for both reads and writes. This metric represents a count and is measured once per second. This is a new metric for DocumentDB serverless. For details, see Monitoring Amazon DocumentDB with CloudWatch.

- **`TempStorageThroughput`** — The amount of data transferred to and from local storage associated with the instance. This metric represents bytes and is measured once per second. This is a new metric for DocumentDB serverless. For details, see Monitoring Amazon DocumentDB with CloudWatch.

Typically, most scaling up for DocumentDB serverless instances is caused by memory usage and CPU activity. The `TempStorageIops` and `TempStorageThroughput` metrics can help you to diagnose the rare cases where network activity for transfers between your instance and local storage devices is responsible for unexpected capacity increases. To monitor other network activity, you can use these existing metrics:

- `NetworkReceiveThroughput`
- `NetworkThroughput`
- `NetworkTransmitThroughput`
- `StorageNetworkReceiveThroughput`
- `StorageNetworkThroughput`
- `StorageNetworkTransmitThroughput`

How DocumentDB serverless CloudWatch metrics apply to your AWS bill

The DocumentDB serverless charges on your AWS bill are calculated based on the same `ServerlessDatabaseCapacity` metric that you can monitor. The billing mechanism can differ from the computed CloudWatch average for this metric in cases where you use DocumentDB serverless capacity for only part of an hour. It can also differ if system issues make the CloudWatch metric unavailable for brief periods. Thus, you might see a slightly different value of DCU-hours on your bill than if you compute the number yourself from the `ServerlessDatabaseCapacity` average value.

Examples of Amazon CloudWatch CLI commands for DocumentDB serverless metrics

The following AWS CLI examples demonstrate how you can monitor the most important CloudWatch metrics related to DocumentDB serverless. In each case, replace the `Value=` string for the `--dimensions` parameter with the identifier of your own DocumentDB serverless instance.

The following Linux example displays the minimum, maximum, and average capacity values for an instance, measured every 10 minutes over one hour. The Linux date commands specifies the start and end times relative to the current date and time. The `sort_by` function in the `--query` parameter sorts the results chronologically based on the `Timestamp` field.

```
aws cloudwatch get-metric-statistics \
    --metric-name "ServerlessDatabaseCapacity" \
    --start-time "$(date -d '1 hour ago')" \
    --end-time "$(date -d 'now')" \
    --period 600 \
    --namespace "AWS/DocDB" \
    --statistics Minimum Maximum Average \
    --dimensions Name=DBInstanceIdentifier,Value=my_instance \
    --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' \
    --output table
```

The following Linux example demonstrates monitoring the capacity of an instance in a cluster. It measures the minimum, maximum, and average capacity utilization of an instance. The measurements are taken once each hour over a three-hour period. These examples use the `DCUUtilization` metric representing a percentage of the upper limit on DCUs, instead of `ServerlessDatabaseCapacity` representing a fixed number of DCUs. That way, you don't need to know the actual numbers for the minimum and maximum DCU values in the capacity range. You can see percentages ranging from 0 to 100.

```
aws cloudwatch get-metric-statistics \
    --metric-name "DCUUtilization" \
    --start-time "$(date -d '3 hours ago')" \
    --end-time "$(date -d 'now')" \
    --period 3600 \
    --namespace "AWS/DocDB" \
    --statistics Minimum Maximum Average \
    --dimensions Name=DBInstanceIdentifier,Value=my_instance \
    --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' \
    --output table
```

The following Linux example does similar measurements as the previous ones. In this case, the measurements are for the `CPUUtilization` metric. The measurements are taken every 10

minutes over a 1-hour period. The numbers represent the percentage of available CPU used, based on the CPU resources available to the maximum capacity setting for the instance.

```
aws cloudwatch get-metric-statistics \
    --metric-name "CPUUtilization" \
    --start-time "$(date -d '1 hour ago')" \
    --end-time "$(date -d 'now')" \
    --period 600 \
    --namespace "AWS/DocDB" \
    --statistics Minimum Maximum Average \
    --dimensions Name=DBInstanceIdentifier,Value=my_instance \
    --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' \
    --output table
```

The following Linux example does similar measurements as the previous ones. In this case, the measurements are for the `FreeableMemory` metric. The measurements are taken every 10 minutes over a 1-hour period.

```
aws cloudwatch get-metric-statistics \
    --metric-name "FreeableMemory" \
    --start-time "$(date -d '1 hour ago')" \
    --end-time "$(date -d 'now')" \
    --period 600 \
    --namespace "AWS/DocDB" \
    --statistics Minimum Maximum Average \
    --dimensions Name=DBInstanceIdentifier,Value=my_instance \
    --query 'sort_by(Datapoints[*].
{min:Minimum,max:Maximum,avg:Average,ts:Timestamp},&ts)' \
    --output table
```

Monitoring DocumentDB serverless performance with Performance Insights

You can use Performance Insights to monitor the performance of DocumentDB serverless instances. For Performance Insights procedures, see Monitoring with Performance Insights.

The following new Performance Insights counters apply to DocumentDB serverless instances:

- **`os.general.serverlessDBCapacity`** — The current capacity of the instance in DCUs. The value corresponds to the `ServerlessDatabaseCapacity` CloudWatch metric for the instance.

- **`os.general.dcuUtilization`** — The percentage of current capacity out of the maximum configured capacity. The value corresponds to the `DCUUtilization` CloudWatch metric for the instance.
- **`os.general.maxConfiguredDcu`** — The maximum capacity that you configured for this DocumentDB serverless instance. It's measured in DCUs.
- **`os.general.minConfiguredDcu`** — The minimum capacity that you configured for this DocumentDB serverless instance. It's measured in DCUs.

For the full list of Performance Insights counters, see Performance Insights for counter metrics.

When vCPU values are shown for a DocumentDB serverless instance in Performance Insights, those values represent estimates based on the DCU value for the instance. At the default interval of one minute, any fractional vCPU values are rounded up to the nearest whole number. For longer intervals, the vCPU value shown is the average of the integer vCPU values for each minute.

Using Amazon DocumentDB Elastic Clusters

Amazon DocumentDB elastic clusters support workloads with millions of reads/writes per second and petabytes of storage capacity. Elastic clusters also simplify how developers interact with Amazon DocumentDB by eliminating the need to choose, manage or upgrade instances.

> **Note**
>
> Elastic clusters use a MongoDB 5.0-compatible wire protocol. They are not supported on Amazon DocumentDB engine version 8.0, and are not available in China Regions or AWS GovCloud.

Amazon DocumentDB elastic clusters were created to:

- Provide a solution for customers looking for a database that provides virtually limitless scale with rich query capabilities and MongoDB API compatibility.
- Give customers higher connection limits, and to reduce downtime from patching.
- Continue investing in a cloud-native, elastic, and class leading architecture for JSON workloads.

Topics

- Elastic cluster use cases
- Advantages of elastic clusters
- Elastic cluster Region and version availability
- Limitations
- Amazon DocumentDB elastic clusters: how it works
- Get started with Amazon DocumentDB elastic clusters
- Amazon DocumentDB elastic cluster best practices
- Managing Amazon DocumentDB elastic clusters
- Data encryption at rest for Amazon DocumentDB elastic clusters
- Service-linked roles in elastic clusters

Elastic cluster use cases

Document databases are useful for workloads that require a flexible schema for fast, iterative development. For example Amazon DocumentDB use cases, see Document database use cases.

The following are some examples of use cases for which elastic clusters can provide significant advantages:

User profiles

Because document databases have a flexible schema, they can store documents that have different attributes and data values at scale. Elastic clusters are a practical solution to online profiles in which different users provide different types of information. Suppose that your applications support hundreds of millions of user profiles. You can use elastic clusters to support such applications because they can be scaled up and out to support millions of writes and reads to these user profiles. You can also scale down for off-peak hours to reduce cost.

Content management and historical records

To effectively manage content, you must be able to collect and aggregate content from a variety of sources, and then deliver it to the customer. Due to their flexible schema, document databases are perfect for collecting and storing any type of data. You can use them to create and incorporate new types of content, including user-generated content such as images, comments, and videos. Over time, your database may require more storage. With elastic clusters, you can distribute your data over more storage volumes enabling you to store petabytes of data in a single cluster.

Advantages of elastic clusters

AWS service integration

Amazon DocumentDB elastic clusters integrate with other AWS services in the same way Amazon DocumentDB does:

- **Migration** — You can use AWS Database Migration Service (DMS) to migrate from MongoDB and other relational databases to Amazon DocumentDB elastic clusters.
- **Monitoring** — You can monitor the health and performance of your elastic cluster using Amazon CloudWatch.

- **Security** — You can set up authentication and authorization through AWS Identity and Access Management (IAM) to manage your elastic cluster resources such as creating and updating an elastic cluster, and use Amazon VPC for secure VPC-only connections. IAM authentication login for elastic clusters is not supported.
- **Data management** — You can use AWS Glue to import and export data from/to other AWS services such as Amazon S3, Amazon Redshift and Amazon OpenSearch Service.

Elastic cluster Region and version availability

Region availability for elastic clusters

The following table shows the AWS Regions where Amazon DocumentDB elastic clusters are currently available and the endpoint for each Region.

Region name	Region	Availability zones
US East (N. Virginia)	`us-east-1`	5
US East (Ohio)	`us-east-2`	3
US West (Oregon)	`us-west-2`	3
Asia Pacific (Hong Kong)	`ap-east-1`	3
Asia Pacific (Mumbai)	`ap-south-1`	3
Asia Pacific (Seoul)	`ap-northeast-2`	3
Asia Pacific (Singapore)	`ap-southeast-1`	3
Asia Pacific (Sydney)	`ap-southeast-2`	3
Asia Pacific (Tokyo)	`ap-northeast-1`	3
Canada (Central)	`ca-central-1`	3
South America (São Paulo)	`sa-east-1`	3
Europe (Frankfurt)	`eu-central-1`	3

Region name	Region	Availability zones
Europe (Ireland)	eu-west-1	3
Europe (London)	eu-west-2	3
Europe (Milan)	eu-south-1	3
Europe (Paris)	eu-west-3	3

Version availability

Elastic clusters support the MongoDB 5.0-compatable wire protocol. For differences between Amazon DocumentDB 4.0 instance-based clusters and elastic clusters, see Functional differences between Amazon DocumentDB 4.0 and elastic clusters. Note: Elastic Clusters are not supported on Amazon DocumentDB 8.0

Limitations

Elastic cluster management

The following cluster management features and capabilities are not supported in this release:

- Custom port (only 27017 is supported)
- Ability to use IAM users and roles to authenticate into the database
- Ability to create global clusters
- Existing Amazon DocumentDB events and subscribing to events
- Range sharding
- Shard existing collection
- Multi-field shard key
- Change shard key
- Point-in-time restore
- Cloning
- Performance Insights

> **Note**
>
> For information about elastic cluster quotas, see Amazon DocumentDB quotas.

Query and write operations

The following query and write operation commands and capabilities are not supported in this release:

- DDL commands during scaling operations
- Profiler
- Parameter groups
- AWS Config
- AWS Backup

Collection and index management

The following collection and index management features are not supported in this release:

- Unique indexes
- Partial indexes
- Text indexes
- Vector indexes
- Document compression

Administration and diagnostics

The following administration and diagnostic commands and capabilities are not supported in this release:

- AWS Secrets Manager
- Role-based-access-control (RBAC) custom roles.
- When connecting, write concern of 0 is not supported.

- Changing subnets belonging to an VPC that is not currently assigned to an existing elastic cluster.

Opt-in features

The following Amazon DocumentDB opt-in features are not supported in this release:

- ACID transactions
- DDL/DML auditing
- Change streams
- Session commands

Amazon DocumentDB elastic clusters: how it works

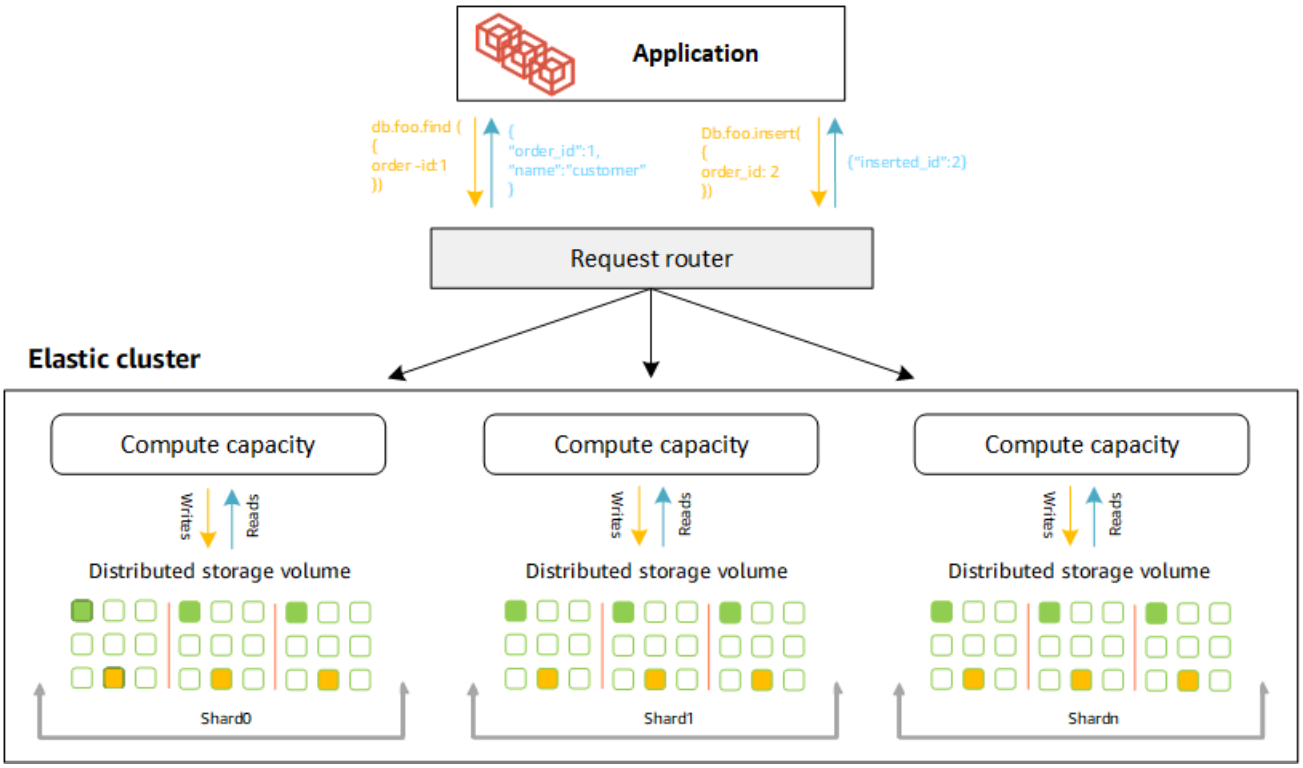
The topics in this section provide information about the mechanisms and functions that power Amazon DocumentDB elastic clusters.

Topics

- Amazon DocumentDB elastic cluster sharding
- Elastic cluster migration
- Elastic cluster scaling
- Elastic cluster reliability
- Elastic cluster storage and availability
- Functional differences between Amazon DocumentDB 4.0 and elastic clusters

Amazon DocumentDB elastic cluster sharding

Amazon DocumentDB elastic clusters use hash-based sharding to partition data across a distributed storage system. Sharding, also known as partitioning, splits large data sets into small data sets across multiple nodes enabling you to scale out your database beyond vertical scaling limits. Elastic clusters use the separation, or “decoupling,” of compute and storage in Amazon DocumentDB, enabling you to scale independently of each other. Rather than re-partitioning collections by moving small chunks of data between compute nodes, elastic clusters copy data efficiently within the distributed storage system.



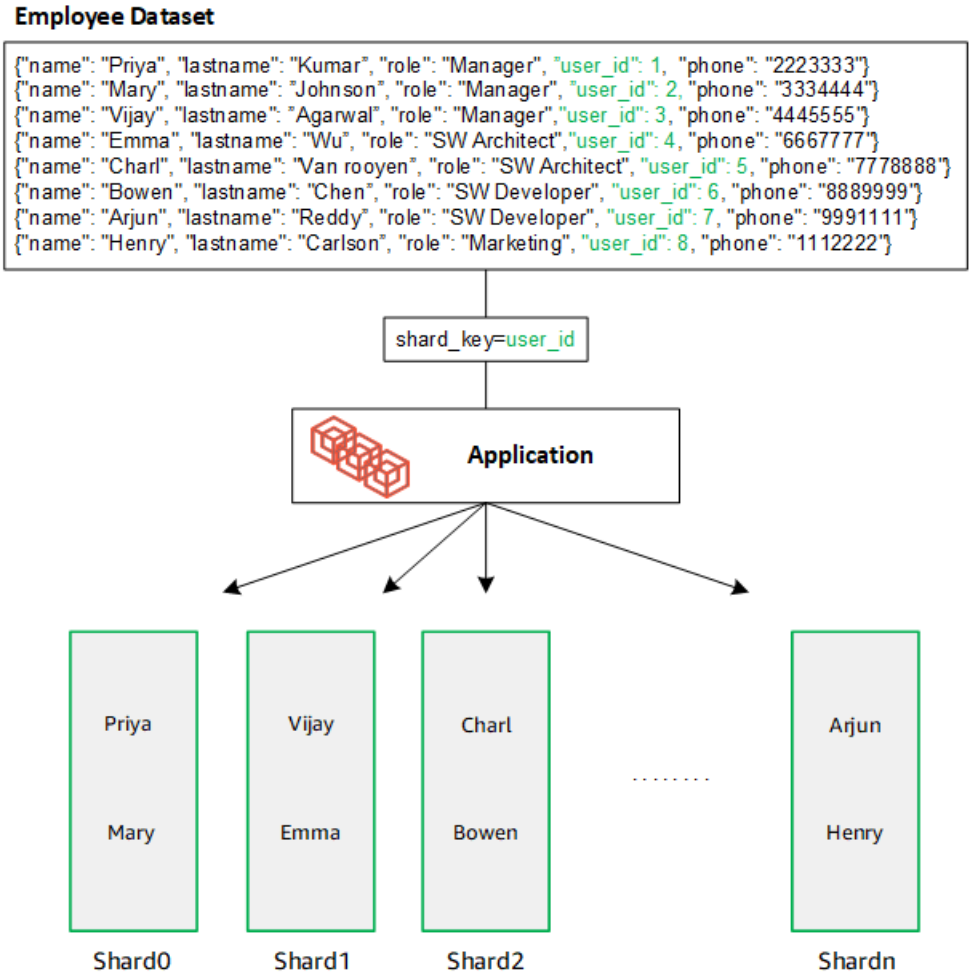
Shard definitions

Definitions of shard nomenclature:

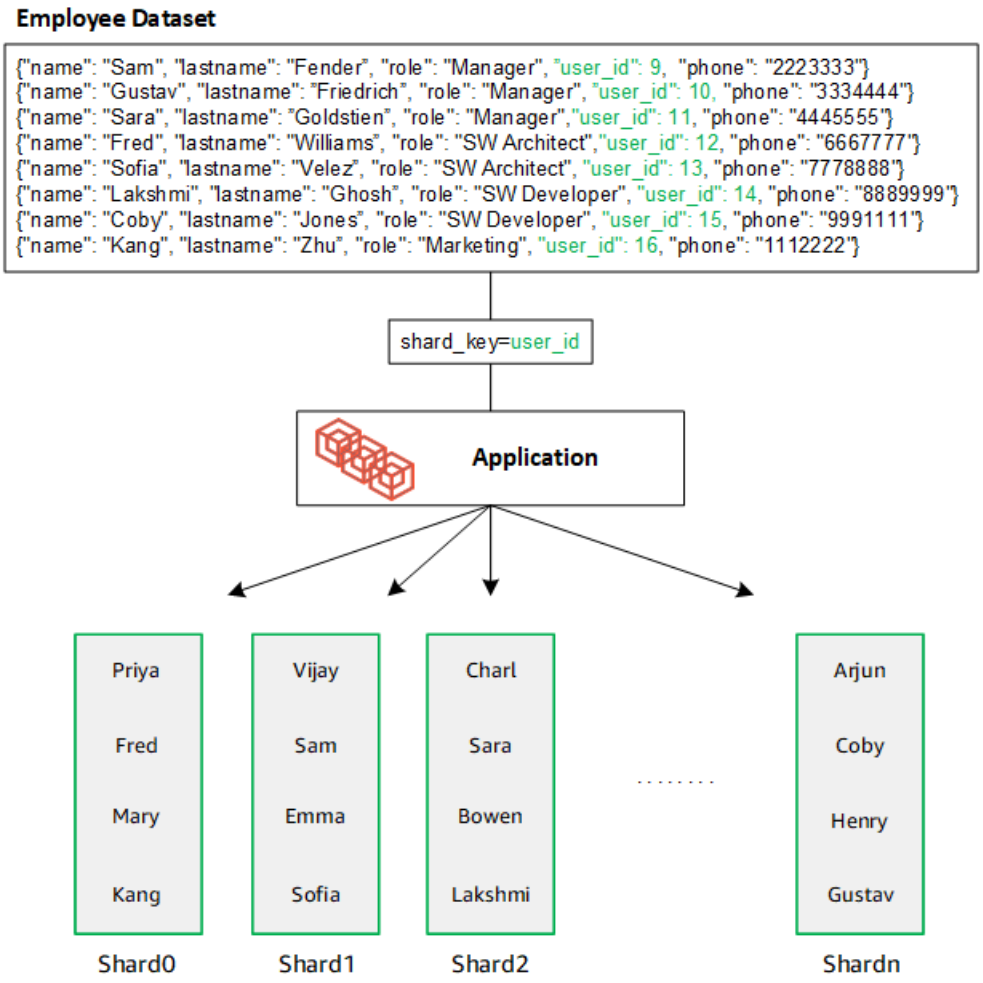
- **Shard** — A shard provides compute for an elastic cluster. It will have a single writer instance and 0–15 read replicas. By default, a shard will have two instances: a writer and a single read replica. You can configure a maximum of 32 shards and each shard instance can have a maximum of 64 vCPUs.
- **Shard key** — A shard key is a required field in your JSON documents in sharded collections that elastic clusters use to distribute read and write traffic to the matching shard.
- **Sharded collection** — A sharded collection is a collection whose data is distributed across an elastic cluster in data partitions.
- **Partition** — A partition is a logical portion of sharded data. When you create a sharded collection, the data is organized into partitions within each shard automatically based on the shard key. Each shard has multiple partitions.

Distributing data across configured shards

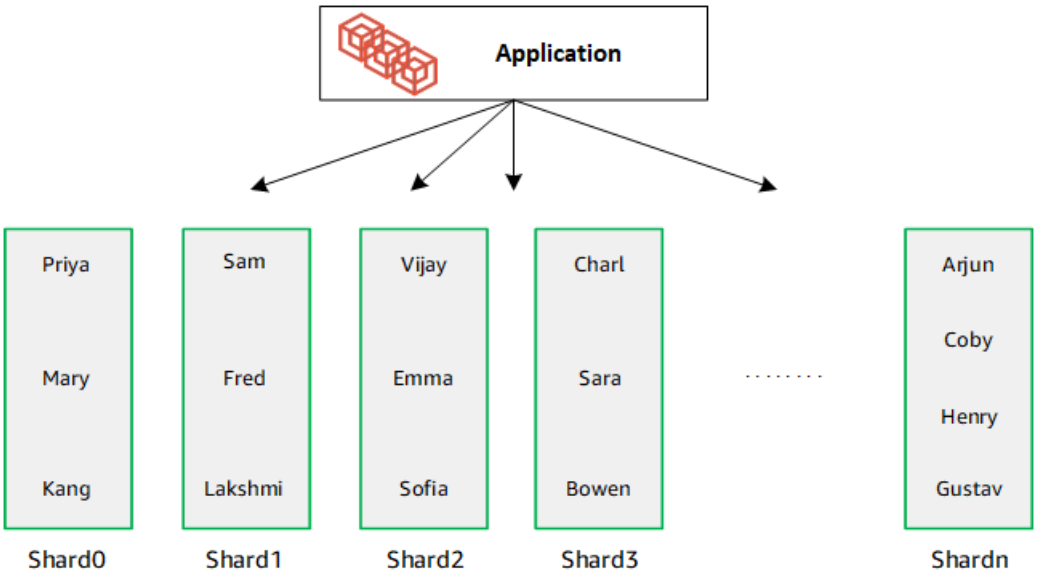
Create a shard key that has many unique values. A good shard key will evenly partition your data across the underlying shards, giving your workload the best throughput and performance. The following example is employee name data that uses a shard key named "user_id":



DocumentDB uses hash sharding to partition your data across underlying shards. Additonal data is inserted and distributed the same way:



When you scale out your database by adding additional shards, Amazon DocumentDB automatically redistributes the data:



Elastic cluster migration

Amazon DocumentDB supports migrating MongoDB sharded data to elastic clusters. Offline, online, and hybrid migration methods are supported. For more information, see Migrating and upgrading Amazon DocumentDB.

Elastic cluster scaling

Amazon DocumentDB elastic clusters provide the ability to increase the number of shards (scale out) in your elastic cluster, and the number of vCPUs applied to each shard (scale up). You can also reduce the number of shards and compute capacity (vCPUs) as needed.

For scaling best practices, see Scaling elastic clusters.

> **Note**
>
> Cluster-level scaling is also available. For more information, see Scaling Amazon DocumentDB clusters.

Elastic cluster reliability

Amazon DocumentDB is designed to be reliable, durable, and fault-tolerant. To improve availability, elastic clusters deploy two nodes per shard placed across different Availability Zones. Amazon DocumentDB includes several automatic features that make it a reliable database solution. For more information, see Amazon DocumentDB reliability.

Elastic cluster storage and availability

Amazon DocumentDB data is stored in a cluster volume, which is a single, virtual volume that uses solid state drives (SSDs). A cluster volume consists of six copies of your data, which are replicated automatically across multiple Availability Zones in a single AWS Region. This replication helps ensure that your data is highly durable, with less possibility of data loss. It also helps ensure that your cluster is more available during a failover because copies of your data already exist in other Availability Zones. For more details on storage, high availability, and replication see Amazon DocumentDB: how it works.

Functional differences between Amazon DocumentDB 4.0 and elastic clusters

The following functional differences exist between Amazon DocumentDB 4.0 and elastic clusters.

- Results from `top` and `collStats` are partitioned by shards. For sharded collections, data is distributed among multiple partitions and `collStats` reports aggregated `collScans` from the partitions.
- Collection statistics from `top` and `collStats` for sharded collections are reset when the cluster shard count is changed.
- The backup built-in role now supports `serverStatus`. Action - Developers and applications with backup role can collect statistics about the state of the Amazon DocumentDB cluster.
- The `SecondaryDelaySecs` field replaces `slaveDelay` in `replSetGetConfig` output.
- The `hello` command replaces `isMaster` - `hello` returns a document that describes the role of the elastic cluster.
- The `$elemMatch` operator in elastic clusters only matches documents in the first nesting level of an array. In Amazon DocumentDB 4.0, the operator traverses all levels before returning matched documents. For example:

```
db.foo.insert(
[
    {a: {b: 5}},
    {a: {b: [5]}},
    {a: {b: [3, 7]}},
    {a: [{b: 5}]},
    {a: [{b: 3}, {b: 7}]},
    {a: [{b: [5]}]},
    {a: [{b: [3, 7]}]},
    {a: [[{b: 5}]]},
    {a: [[{b: 3}, {b: 7}]]},
    {a: [[{b: [5]}]]},
    {a: [[{b: [3, 7]}]]}
]);
// Elastic clusters
> db.foo.find({a: {$elemMatch: {b: {$elemMatch: {$lt: 6, $gt: 4}}}}}, {_id: 0})
{ "a" : [ { "b" : [ 5 ] } ] }
```

```
// Docdb 4.0: traverse more than one level deep
> db.foo.find({a: {$elemMatch: {b: {$elemMatch: {$lt: 6, $gt: 4}}}}}, {_id: 0})
{ "a" : [ { "b" : [ 5 ] } ] }
{ "a" : [ [ { "b" : [ 5 ] } ] ] }
```

- The "$" projection in Amazon DocumentDB 4.0 returns all documents with all fields. With elastic clusters, the `find` command with a "$" projection returns documents that match the query parameter containing only the field that matched the "$" projection.
- In elastic clusters, the `find` commands with `$regex` and `$options` query parameters return an error: "Cannot set options in both $regex and $options."

- With elastic clusters, `$indexOfCP` now returns "-1" when:
  - the substring is not found in the `string expression`, or
  - `start` is a number greater than end, or
  - `start` is a number greater than the byte length of the string.

  In Amazon DocumentDB 4.0, `$indexOfCP` returns "0" when the `start` position is a number greater than end or the byte length of the string.
- With elastic clusters, projection operations in `_id fields`, e.g., `{"_id.nestedField" : 1}`, return documents that only include the projected field. Meanwhile, in Amazon DocumentDB 4.0, nested field projection commands do not filter out any document.

Get started with Amazon DocumentDB elastic clusters

This getting started section walks you through on how you can create and query your first elastic cluster.

There are many ways to connect and get started with Amazon DocumentDB. The following procedure is the quickest, simplest, and easiest way for users to get started using our powerful document database. This guide uses AWS CloudShell to connect and query your Amazon DocumentDB cluster directly from the AWS Management Console. New customers who are eligible for the AWS Free Tier can use Amazon DocumentDB and CloudShell for free. If your AWS CloudShell environment or Amazon DocumentDB elastic cluster makes use of resources beyond the free tier, you are charged the normal AWS rates for those resources. This guide will get you started with Amazon DocumentDB in less than 5 minutes.

Topics

- Prerequisites
- Step 1: Create an elastic cluster
- Step 2: Connect to your elastic cluster
- Step 3: Shard your collection, insert and query data
- Step 4: Explore

Prerequisites

Before you create your first Amazon DocumentDB cluster, you must do the following:

Create an Amazon Web Services (AWS) account

Before you can begin using Amazon DocumentDB, you must have an Amazon Web Services (AWS) account. The AWS account is free. You pay only for the services and resources that you use.

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://signin.aws.amazon.com/signup?request_type=register.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Set up the needed AWS Identity and Access Management (IAM) permissions.

Access to manage Amazon DocumentDB resources such as clusters, instances, and cluster parameter groups requires credentials that AWS can use to authenticate your requests. For more information, see Identity and Access Management for Amazon DocumentDB.

1. In the search bar of the AWS Management Console, enter **`IAM`** and select **IAM** in the drop down menu.
2. Once you're in the IAM console, select **Users** from the navigation pane.
3. Select your username.
4. Choose **Add permissions**.
5. Select **Attach policies directly**.
6. Type `AmazonDocDBElasticFullAccess` in the search bar and select it once it appears in the search results.
7. Choose **Next**.
8. Choose **Add permissions**.

> **Note**
>
> Your AWS account includes a default VPC in each Region. If you choose to use an Amazon VPC, complete the steps in the Create a Amazon VPC topic in the *Amazon VPC User Guide*.

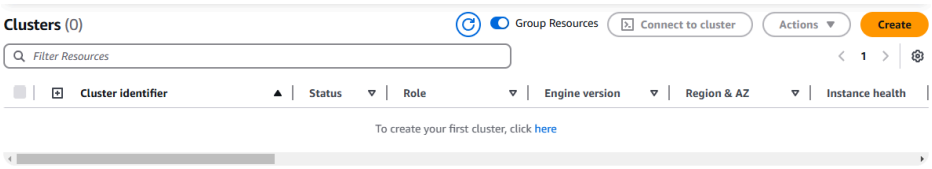
Step 1: Create an elastic cluster

In this section we explain how to create a brand new elastic cluster, using either the AWS Management Console or AWS CLI with the following instructions.

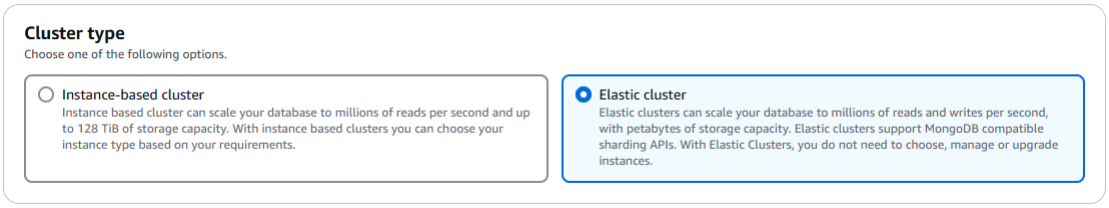
Using the AWS Management Console

To create an elastic cluster configuration using the AWS Management Console:

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. On the **Amazon DocumentDB Management Console**, under **Clusters**, choose **Create**.



3. On the **Create Amazon DocumentDB cluster** page, in the **Cluster type** section, choose **Elastic cluster**.



4. In the **Configuration** section, configure the following:

   a. In the **Cluster name** field, enter a unique cluster identifier (following the naming requirements below the field).

   b. In the **Shard count** field, enter the number of shards you want in your cluster. The maximum number of shards per cluster is 32.

> **Note**
>
> Two nodes will be deployed for each shard. Both nodes will have the same shard capacity.

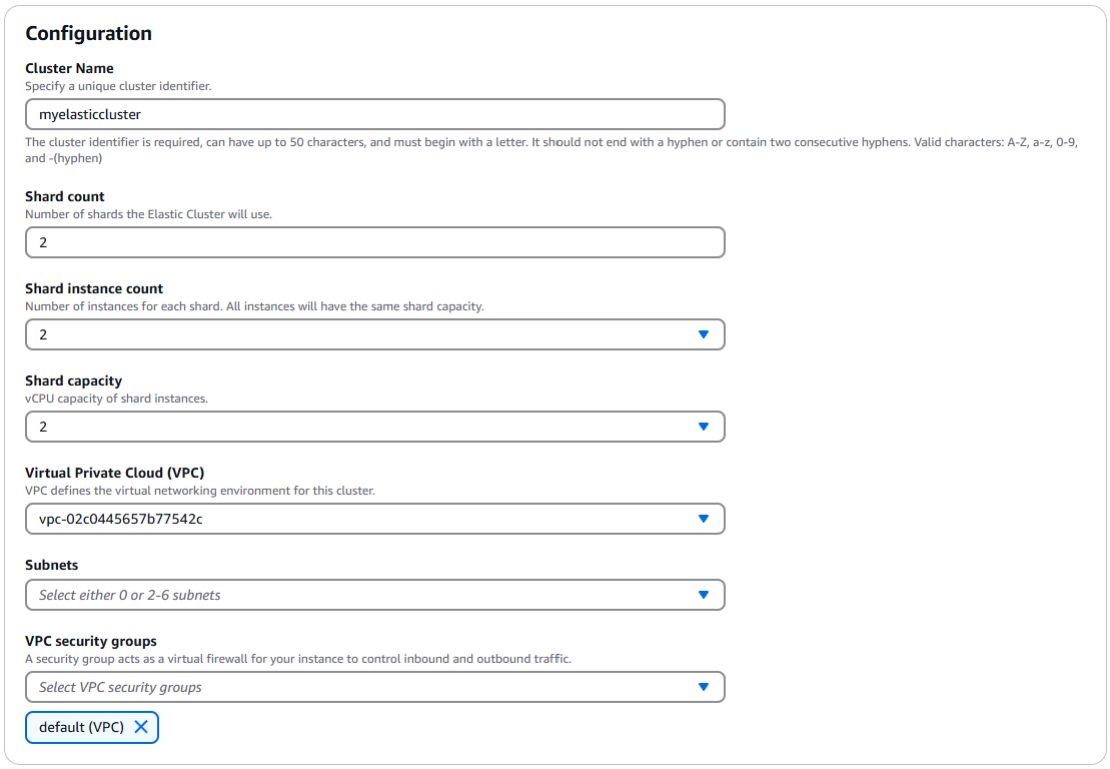
   c. In the **Shard instance count** field, choose the number of replica instances you want associated with each shard. The maximum number of shard instances is 16, in increments of 1. All replica instances have the same shard capacity as defined in the following field. For testing purposes, the default value of 2 should suffice.

> **Note**
>
> The number of replica instances applies to all shards in the elastic cluster. A shard instance count value of 1 means there is one writer instance, and any additional instances are replicas that can be used for reads and to improve availability. For testing purposes, the default value of 2 should suffice.

   d. In the **Shard capacity** field, choose the number of virtual CPUs (vCPUs) you want associated with each shard instance. The maximum number of vCPUs per shard instance is 64. Allowed values are 2, 4, 8, 16, 32, 64. For testing purposes, the default value of 2 should suffice.

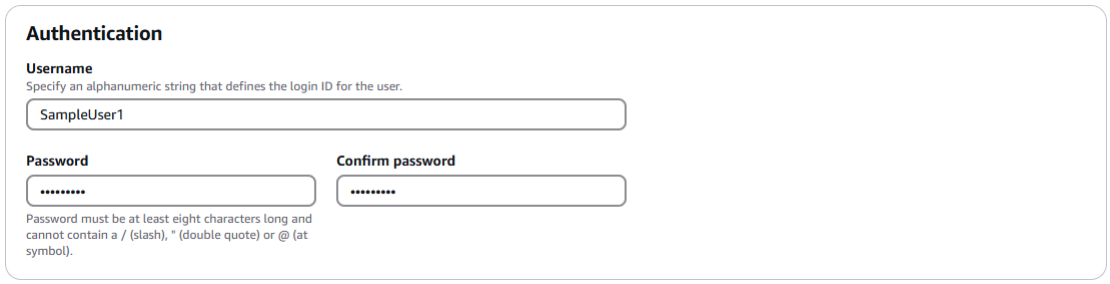
   e. In the **Virtual Private Cloud (VPC)** field, choose a VPC from the dropdown list.

f. For **Subnets** and **VPC security groups**, you can use the defaults or select three subnets of your choice and up to three **VPC security groups** (minimum one).



5. In the **Authentication** section, enter a string that identifies the login name of the primary user in the **Username** field.

   In the **Password** field, enter a unique password that complies with the instructions, then confirm it.



6. In the **Encryption** section, keep the default setting (**Default Key**).

   Optionally, you can enter a AWS KMS key ARN you created. For more information, see Data encryption at rest for Amazon DocumentDB elastic clusters.

> **Important**
>
> Encryption must be enabled for elastic clusters.

7. In the **Backup** section, edit the fields according to your backup requirements. For test purposes, you can retain the default settings.



   a. **Backup retention period**—In the list, choose the number of days to keep automatic backups of this cluster before deleting them.

   b. **Backup window**—Set the daily time and duration during which Amazon DocumentDB is to make backups of this cluster.

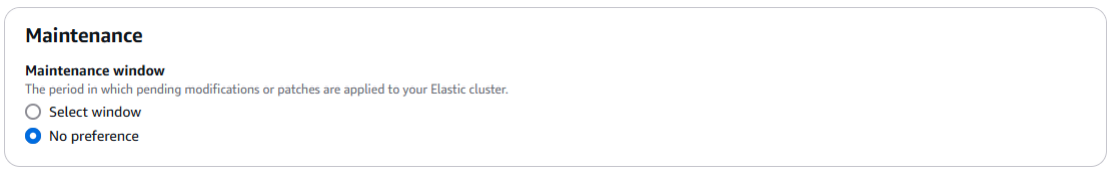
      i. Choose **Select window** if you want to configure the time and duration when backups are created.

         Start time—In the first list, choose the start time hour (UTC) for starting your automatic backups. In the second list, choose the minute of the hour that you want automatic backups to begin.

         Duration—In the list, choose the number of hours to be allocated to creating automatic backups.

      ii. Choose **No preference** if you want Amazon DocumentDB to choose the time and duration when backups are created.

8. In the **Maintenance** section, choose the day, time, and duration when modifications or patches are applied to your cluster. For test purposes, you can retain the default settings.



9. Choose **Create cluster**.

The elastic cluster is now provisioning. This can take up to a few minutes to finish. You can connect to your cluster when the elastic cluster status shows as **Available** in the **Clusters** list.

Using the AWS CLI

To create an elastic cluster using the AWS CLI, use the `create-cluster` operation with the following parameters:

- **--cluster-name**—Required. The current name of the elastic scale cluster as entered during creation or last modified.
- **--shard-capacity**—Required. The number of vCPUs assigned to each shard. Maximum is 64. Allowed values are 2, 4, 8, 16, 32, 64.
- **--shard-count**—Required. The number of shards assigned to the cluster. Maximum is 32.
- **--shard-instance-count**—Optional. The number of replica instances applying to all shards in this cluster. Maximum is 16.
- **--admin-user-name**—Required. The username associated with the admin user.
- **--admin-user-password**—Required. The password associated with the admin user.
- **--auth-type**—Required. The authentication type used to determine where to fetch the password used for accessing the elastic cluster. Valid types are `PLAIN_TEXT` or `SECRET_ARN`.
- **--vpc-security-group-ids**—Optional. Configure a list of EC2 VPC security groups to associate with this cluster.
- **--preferred-maintenance-window**—Optional. Configure the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC).

  The format is: `ddd:hh24:mi-ddd:hh24:mi`. Valid days (ddd): Mon, Tue, Wed, Thu, Fri, Sat, Sun

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Web Services Region, occurring on a random day of the week.

  Minimum 30-minute window.
- **--kms-key-id**—Optional. Configure the KMS key identifier for an encrypted cluster.

  The KMS key identifier is the Amazon Resource Name (ARN) for the AWS KMS encryption key. If you are creating a cluster using the same Amazon Web Services account that owns the KMS encryption key that is used to encrypt the new cluster, you can use the KMS key alias instead of the ARN for the KMS encryption key.

If an encryption key is not specified in KmsKeyId and if the `StorageEncrypted` parameter is true, Amazon DocumentDB uses your default encryption key.

- **--preferred-backup-window**—Optional. The daily preferred time range during which automated backups are created. The default is a 30-minute window selected at random from an 8-hour block of time for each AWS Region.
- **--backup-retention-period**—Optional. The number of days for which automated backups are retained. The default value is 1.
- **--storage-encrypted**—Optional. Configues whether the cluster is encrypted or not encrypted.

  `--no-storage-encrypted` specifies that the cluster is not encrypted.
- **--subnet-ids**—Optional. Configure network subnet Ids.

In the following example, replace each *`user input placeholder`* with your own information.

> **Note**
>
> The following examples include creation of a specific KMS key. To use the default KMS key, do not include the `--kms-key-id` parameter.

For Linux, macOS, or Unix:

```
aws docdb-elastic create-cluster \
     --cluster-name sample-cluster-123 \
     --shard-capacity 8 \
     --shard-count 4 \
     --shard-instance-count 3 \
     --auth-type PLAIN_TEXT \
     --admin-user-name testadmin \
     --admin-user-password testPassword \
     --vpc-security-group-ids ec-65f40350 \
     --kms-key-id arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 \
     --subnet-ids subnet-9253c6a3, subnet-9f1b5af9 \
     --preferred-backup-window 18:00-18:30 \
     --backup-retention-period 7
```

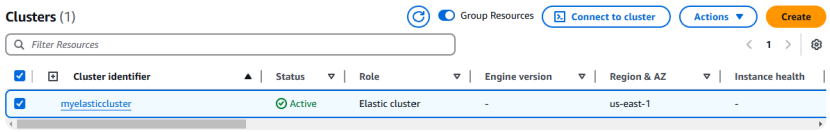
For Windows:

```
aws docdb-elastic create-cluster ^
     --cluster-name sample-cluster-123 ^
     --shard-capacity 8 ^
     --shard-count 4 ^
     --shard-instance-count 3 ^
     --auth-type PLAIN_TEXT ^
     --admin-user-name testadmin ^
     --admin-user-password testPassword ^
     --vpc-security-group-ids ec-65f40350 ^
     --kms-key-id arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 ^
     --subnet-ids subnet-9253c6a3, subnet-9f1b5af9 \
     --preferred-backup-window 18:00-18:30 \
     --backup-retention-period 7
```

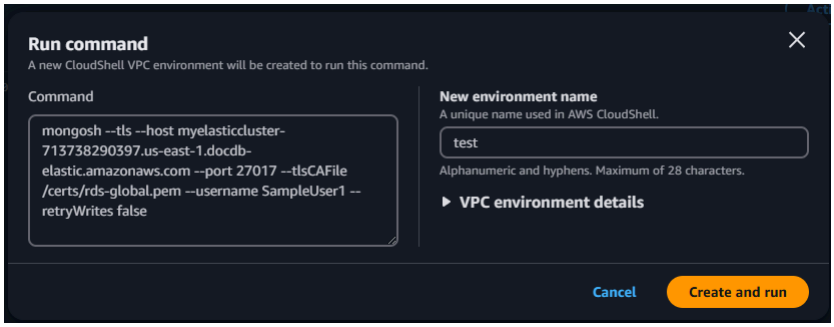
Step 2: Connect to your elastic cluster

Connect to your Amazon DocumentDB elastic cluster using AWS CloudShell.

1. On the Amazon DocumentDB management console, under **Clusters**, locate the elastic cluster you created. Choose your cluster by clicking the check box next to it.



2. Choose **Connect to cluster** (which is next to the **Actions** dropdown menu. This button is enabled only after you have clicked the checkbox next to your cluster, and the status of the cluster shows as **Available**. The CloudShell **Run command** screen appears.
3. In the **New environment name** field, enter a unique name, such as "test" and choose **Create and run**. VPC environment details are automatically configured for your Amazon DocumentDB database.



4. When prompted, enter the password you created in Step 1: Create an Amazon DocumentDB elastic cluster (sub-step 5).



   After you enter your password and your prompt becomes `direct: mongos] <env-name>>`, you are successfully connected to your Amazon DocumentDB cluster

> **Note**
>
> For information about troubleshooting, see Troubleshooting Amazon DocumentDB.

Step 3: Shard your collection, insert and query data

Elastic clusters add support for sharding in Amazon DocumentDB. Now that you are connected to your cluster, you can shard the cluster, insert data and run a few queries.

1. To shard a collection, enter the following:

```
sh.shardCollection("db.Employee1" , { "Employeeid" : "hashed" })
```

2. To insert a single document, enter the following:

```
db.Employee1.insertOne({"Employeeid":1, "Name":"Joe", "LastName": "Bruin", "level":
 1 })
```

The following output is displayed:

```
WriteResult({ "nInserted" : 1 })
```

3. To read the document that you wrote, enter the `findOne()` command (it returns a single document):

```
db.Employee1.findOne()
```

The following output is displayed:

```
{
"_id" : ObjectId("61f344e0594fe1a1685a8151"),
"EmployeeID" : 1,
"Name" : "Joe",
"LastName" : "Bruin",
"level" : 1
}
```

4. To perform a few more queries, consider a gaming profile use case. First, insert a few entries into a collection titled "Employee". Enter the following:

```
db.profiles.insertMany([ { "_id": 1, "name": "Matt", "status": "active", "level":
 12, "score": 202 },
    { "_id": 2, "name": "Frank", "status": "inactive", "level": 2, "score": 9 },
    { "_id": 3, "name": "Karen", "status": "active", "level": 7, "score": 87 },
    { "_id": 4, "name": "Katie", "status": "active", "level": 3, "score": 27 }
])
```

The following output is displayed:

```
{ acknowledged: true,
     insertedIds: {
        '0': ObjectId('679d02cd6b5a0581be78bcbd'),
        '1': ObjectId('679d02cd6b5a0581be78bcbe'),
        '2': ObjectId('679d02cd6b5a0581be78bcbf'),
        '3': ObjectId('679d02cd6b5a0581be78bcc0')
```

```
    }
}
```

5. To return all the documents in the profiles collection, enter the `find()` command:

```
db.Employee.find()
```

   The data you entered in step 4 is displayed.

6. To query a single document, include a filter (for example: "Katie"). Enter the following:

```
db.Employee.find({name: "Katie"})
```

   The following output is displayed:

```
[

   {

     _id: ObjectId('679d02cd6b5a0581be78bcc0'),

     Employeeid: 4,

     name: 'Katie',

     lastname: 'Schaper',

     level: 3

   }

]
```

7. To find a profile and modify it, enter the `findAndModify` command. In this example, the employee "Matt" is given a higher level of "14":

```
db.Employee.findAndModify({
    query: { "Employeeid" : 1, "name" : "Matt"},
    update: { "Employeeid" : 1, "name" : "Matt", "lastname" : "Winkle", "level" :
 14 }
})
```

The following output is displayed (note that the level has not changed yet):

```
{

   _id: ObjectId('679d02cd6b5a0581be78bcbd'),

   Employeeid: 1,

   name: 'Matt',

   lastname: 'Winkle',

   level: 12

}
```

8. To verify the level increase, enter the following query:

```
db.Employee.find({name: "Matt"})
```

The following output is displayed:

```
[
   {

   _id: ObjectId('679d02cd6b5a0581be78bcbd'),

   Employeeid: 1,

   name: 'Matt',

   lastname: 'Winkle',

   level: 14

   }
]
```

Step 4: Explore

Congratulations! You have successfully completed the Get started procedure for Amazon DocumentDB elastic clusters.

What's next? Learn how to fully leverage this database with some of its popular features:

- Amazon DocumentDB elastic cluster best practices
- Managing Amazon DocumentDB elastic clusters

> **Note**
>
> The elastic cluster you created from this get started procedure will continue to accrue costs unless you delete it. For directions, see Deleting an elastic cluster.

Amazon DocumentDB elastic cluster best practices

Learn best practices for working with Amazon DocumentDB elastic clusters. All best practices for instance-based Amazon DocumentDB clusters also apply for elastic clusters. This section is continually updated as new best practices are identified.

Topics

- Choosing shard keys
- Connection management
- Unsharded collections
- Scaling elastic clusters
- Monitoring elastic clusters

Choosing shard keys

The following list describes guidelines for creating shard keys.

- Use an evenly distributed hash key to distribute your data across all of the shards in your cluster (avoid hot keys).
- Use your shard key in all read/update/delete requests to avoid scatter gather queries.

- Avoid nested shard keys when doing read/update/delete operations.
- When making batch operations, set `ordered` to false so all shards can run in parallel and improve latencies.

Connection management

The following list describes guidelines for managing your connections to your database.

- Monitor your connection counts and how frequently new connections are opened closed.
- Distribute your connections across all of the subnets in your application's configuration. If your cluster is configured in multiple subnets but you only utilize a subset of the subnets, you may be bottlenecked on your maximum connections.

Unsharded collections

The following describes a guideline for unsharded collections.

- When working with unsharded collections, to distribute load, try keeping highly utilized unsharded collections on different databases. Amazon DocumentDB elastic clusters place databases across different shards and co-locates unsharded collections for the same database on the same shard.

Scaling elastic clusters

The following list describes guidelines for scaling your elastic clusters.

- Scaling operations may cause a brief period of intermittent database and network errors. When possible, avoid scaling during peak hours. Try to scale during maintenance windows.
- Scaling shard capacity up and down (changing vCPU count per shard) to increase compute is preferred over increasing or decreasing the shard-count as it is faster and has a shorter duration of intermittent database and network errors.
- When anticipating growth, favor increasing the shard count instead scaling the shard capacity. This enables you to scale your cluster by increasing the shard capacity for scenarios where you need to quickly scale.
- Monitor your client-side retry policies and retry with exponential backoff and jitter to avoid overloading your database when getting errors while scaling.

- Changing the number of shards can take time because data has to be safely moved from one shard to another. To minimize risk of errors, don't stop or make any other changes to the cluster state while the change is in progress.

Monitoring elastic clusters

The following list describes guidelines for monitoring your elastic clusters.

- Track the peak-to-average ratio of your per-shard metrics to determine if you are driving un-even traffic (have a hot-key/hot-spot). Key metrics to track peak-to-average ratios are:
  - `PrimaryInstanceCPUUtilization`
    - This can be monitored at the per-shard level.
    - At the cluster level you can monitor the average to p99 skew.
  - `PrimaryInstanceFreeableMemory`
    - This can be monitored at the per-shard level.
    - At the cluster level you can monitor the average to p99 skew.
  - `DatabaseCursorsMax`
    - This should be monitored at the per-shard level to determine skew.
  - `Documents-Inserted/Updated/Returned/Deleted`
    - This should be monitored at the per-shard level to determine skew.

Managing Amazon DocumentDB elastic clusters

To manage an Amazon DocumentDB elastic cluster, you must have an IAM policy with the appropriate Amazon DocumentDB control plane permissions. These permissions allow you to create, modify, and delete clusters. The Amazon DocumentDBFullAccess policy provides all the required permissions for administering an Amazon DocumentDB elastic cluster.

The following topics show how to perform various tasks when working with Amazon DocumentDB elastic clusters.

Topics

- Modifying elastic cluster configurations
- Monitoring an elastic cluster

- Deleting an elastic cluster
- Managing elastic cluster snapshots
- Stopping and starting an Amazon DocumentDB elastic cluster
- Maintaining Amazon DocumentDB elastic clusters

Modifying elastic cluster configurations

In this section we explain how to modify elastic cluster, using either the AWS Management Console or AWS CLI with the following instructions.

A primary use of modifying the cluster is to scale shards by increasing or decreasing the shard count and/or shard compute capacity.

Using the AWS Management Console

To modify an elastic cluster configuration using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the name of the cluster you want to modify in the **Cluster identifier** column.
4. Choose **Modify**.
5. Edit the fields you want changed and then select **Modify cluster**.

Configuration

Cluster identifier

SampleCluster

Shard count

Number of shards the Elastic Cluster will use.

2

Shard instance count

Number of instances for each shard. All instances will have the same shard capacity.

2

Shard capacity

vCPU capacity of each shard.

2

Maintenance

Maintenance window

The period in which pending modifications or patches are applied to your Elastic cluster.

- ○ Select window
- ◉ No preference

Authentication

Username

SampleUser

New password

Confirm new password

Password must be at least eight characters long and cannot contain a / (slash), " (double quote) or @ (at symbol).

Network settings

Subnets

Select either 0 or 2-6 subnets

subnet-0b2962f92a0f5a8fb ✕ subnet-08c6d849efd4dfe96 ✕

VPC security groups

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic.

Select VPC security groups

sg-021d234a0a3a2c2fe ✕ sg-0824c1992d6009b52 ✕

> **Note**
>
> Alternatively, you can access the **Modify cluster** dialog by going to the **Clusters** page, checking the box next to your cluster, choosing **Actions**, then **Modify**.

Using the AWS CLI

To modify an elastic cluster configuration using the AWS CLI, use the `update-cluster` operation with the following parameters:

- **`--cluster-arn`**—Required. The ARN identifier of the cluster that you want to modify.
- **`--shard-capacity`**—Optional. The number of vCPUs assigned to each shard. Maximum is 64. Allowed values are 2, 4, 8, 16, 32, 64.
- **`--shard-count`**—Optional. The number of shards assigned to the cluster. Maximum is 32.
- **`--shard-instance`**-count—Optional. The number of replica instances applying to all shards in this cluster. Maximum is 16.
- **`--auth-type`**—Optional. The authentication type used to determine where to fetch the password used for accessing the elastic cluster. Valid types are `PLAIN_TEXT` or `SECRET_ARN`.
- **`--admin-user-password`**—Optional. The password associated with the admin user.
- **`--vpc-security-group-ids`**—Optional. Configure a list of Amazon EC2 and Amazon Virtual Private Cloud (VPC) security groups to associate with this cluster.
- **`--preferred-maintenance-window`**—Optional. Configure the weekly time range during which system maintenance can occur, in Universal Coordinated Time (UTC)

  The format is: `ddd:hh24:mi-ddd:hh24:mi`. Valid days (ddd): Mon, Tue, Wed, Thu, Fri, Sat, Sun

  The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Web Services Region, occurring on a random day of the week.

  Minimum 30-minute window.
- **`--subnet-ids`**—Optional. Configure network subnet Ids.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic update-cluster \
    --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 \
    --shard-capacity 8 \
    --shard-count 4 \
    --shard-instance-count 3 \
    --admin-user-password testPassword \
    --vpc-security-group-ids ec-65f40350 \
    --subnet-ids subnet-9253c6a3, subnet-9f1b5af9
```

For Windows:

```
aws docdb-elastic update-cluster ^
    --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 ^
    --shard-capacity 8 ^
    --shard-count 4 ^
    --shard-instance-count 3 ^
    --admin-user-password testPassword ^
    --vpc-security-group-ids ec-65f40350 ^
    --subnet-ids subnet-9253c6a3, subnet-9f1b5af9
```

To monitor the status of the elastic cluster after your modification, see Monitoring an elastic cluster.

Monitoring an elastic cluster

In this section, we explain how to monitor your elastic cluster, using either the AWS Management Console or AWS CLI with the following instructions.

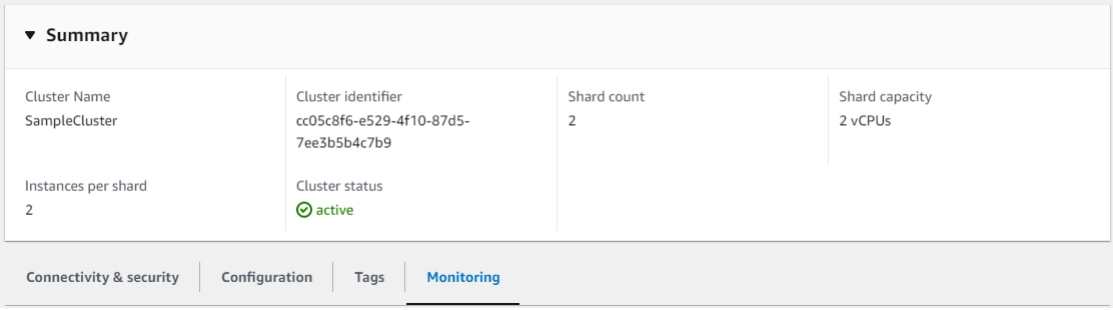
Using the AWS Management Console

To monitor an elastic cluster configuration using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the name of the cluster you want to monitor in the **Cluster identifier** column.
4. Choose the **Monitoring** tab.



A number of charts from Amazon CloudWatch are displayed for the following monitoring categories:

- Resource Utilization
- Throughput
- Operations
- System

You can also access Amazon CloudWatch through the AWS Management Console to set up your own monitoring environment for your elastic clusters.

Using the AWS CLI

To monitor a specific elastic cluster configuration using the AWS CLI, use the `get-cluster` operation with the following parameters:

- **`--cluster-arn`**—Required. The ARN identifier of the cluster for which you want information.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic get-cluster \
    --cluster-arn arn:aws:docdb-elastic:us-west-2:123456789012:cluster:/68ffcdf8-e3af-40a3-91e4-24736f2dacc9
```

For Windows:

```
aws docdb-elastic get-cluster ^
    --cluster-arn arn:aws:docdb:-elastic:us-west-2:123456789012:cluster:/68ffcdf8-e3af-40a3-91e4-24736f2dacc9
```

The output from this operation looks something like the following:

```
"cluster": {
        ...
        "clusterArn": "arn:aws:docdb-elastic:us-
west-2:123456789012:cluster:/68ffcdf8-e3af-40a3-91e4-24736f2dacc9",
        "clusterEndpoint": "stretch-11-477568257630.us-east-1.docdb-
elastic.amazonaws.com",
        "readerEndpoint": "stretch-11-477568257630-ro.us-east-1.docdb-
elastic.amazonaws.com",
        "clusterName": "stretch-11",
        "shardCapacity": 2,
        "shardCount": 3,
        "shardInstanceCount: 5,
        "status": "ACTIVE",
        ...
 }
```

For more information, see `DescribeClusterSnapshot` in the Amazon DocumentDB Resource Management API Reference.

To view the details of all elastic clusters using the AWS CLI, use the `list-clusters` operation with the following parameters:

- **`--next-token`**—Optional. If the number of items output (`--max-results`) is fewer than the total number of items returned by the underlying API calls, the output includes a `NextToken` that you can pass to a subsequent command to retrieve the next set of items.

- **`--max-results`**—Optional. The total number of items to return in the command's output. If more results exist than the specified `max-results` value, a pagination token (`next-token`) is included in the response so that the remaining results can be retrieved.
  - Default: 100
  - Minimum 20, maximum 100

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic list-clusters \
    --next-token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxfQ== \
    --max-results 2
```

For Windows:

```
aws docdb-elastic list-clusters ^
    --next-token eyJNYXJrZXIiOiBudWxsLCAiYm90b190cnVuY2F0ZV9hbW91bnQiOiAxfQ== ^
    --max-results 2
```

The output from this operation looks something like the following:

```
{
   "Clusters": [
      {
         "ClusterIdentifier":"mycluster-1",
         "ClusterArn":"arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-
cluster"
         "Status":"available",
         "ClusterEndpoint":"sample-cluster.sharded-cluster-corcjozrlsfc.us-
west-2.docdb.amazonaws.com"
       }
       {
         "ClusterIdentifier":"mycluster-2",
         "ClusterArn":"arn:aws:docdb:us-west-2:987654321098:sharded-cluster:sample-
cluster"
         "Status":"available",
         "ClusterEndpoint":"sample-cluster2.sharded-cluster-corcjozrlsfc.us-
west-2.docdb.amazonaws.com"
```

```
        }
    ]
}
```

Deleting an elastic cluster

In this section we explain how to delete an elastic cluster, using either the AWS Management Console or AWS CLI with the following instructions.

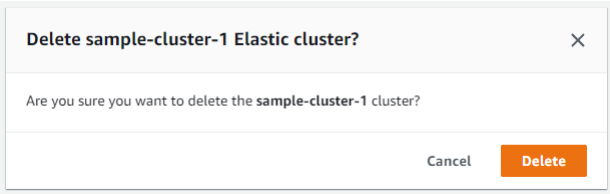
Using the AWS Management Console

To delete an elastic cluster configuration using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. In the cluster list table, select the check box to the left of the cluster name you want to delete and then choose **Actions**. From the dropdown menu, choose **Delete**.
4. In the **Delete "cluster-name" elastic cluster?** dialog box, choose **Delete**.



It takes several minutes for the cluster to be deleted. To monitor the status of the cluster, see Monitoring an Amazon DocumentDB cluster's status.

Using the AWS CLI

To delete an elastic cluster using the AWS CLI, use the `delete-cluster` operation with the following parameters::

- **`--cluster-arn`**—Required. The ARN identifier of the cluster that you want to delete.

- **`--no-skip-final-backup`**—Optional. If you want a final backup, you must include this parameter with a name for the final backup. You must include either `--final-backup-identifier` or `--skip-final-backup`.
- **`--skip-final-backup`**—Optional. Use this parameter only if you don't want to take a final backup before deleting your cluster. The default setting is to take a final snapshot.

The following AWS CLI code examples delete a cluster with an ARN of arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster with a final backup.

In the following example, replace each *`user input placeholder`* with your own information..

For Linux, macOS, or Unix:

```
aws docdb-elastic delete-cluster \
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster \
    --no-skip-final-backup \
    --final-backup-identifier finalArnBU-arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster
```

For Windows:

```
aws docdb-elastic delete-cluster ^
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster ^
    --no-skip-final-backup ^
    --final-backup-identifier finalArnBU-arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster
```

The following AWS CLI code examples delete a cluster with an ARN of arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-cluster without taking a final backup.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic delete-cluster \
```

```
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-
cluster \
    --skip-final-backup \
```

For Windows:

```
aws docdb-elastic delete-cluster ^
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-
cluster ^
    --skip-final-backup ^
```

The output of the `delete-cluster` operation is a display of the cluster you are deleting.

It takes several minutes for the cluster to be deleted. To monitor the status of the cluster, see Monitoring an Amazon DocumentDB cluster's status.

Managing elastic cluster snapshots

Manual snapshots can be taken after an elastic cluster has been created. Automated backups are created the moment the elastic cluster snapshot is created.

> **Note**
>
> Your elastic cluster must be in the `Available` state for a manual snapshot to be taken.

This section explains how you can create, view, restore from, and delete elastic cluster snapshots.

The following topics show how to perform various tasks when working with Amazon DocumentDB elastic cluster snapshots.

Topics

- Creating a manual elastic cluster snapshot
- Viewing an elastic cluster snapshot
- Restoring an elastic cluster from a snapshot
- Copying an elastic cluster snapshot
- Deleting an elastic cluster snapshot

- Managing an elastic cluster snapshot automatic backup

Creating a manual elastic cluster snapshot

In this section we explain how to create a manual elastic cluster snapshot, using either the AWS Management Console or AWS CLI with the following instructions.

Using the AWS Management Console

To create a manual elastic cluster snapshot using the AWS Management Console:

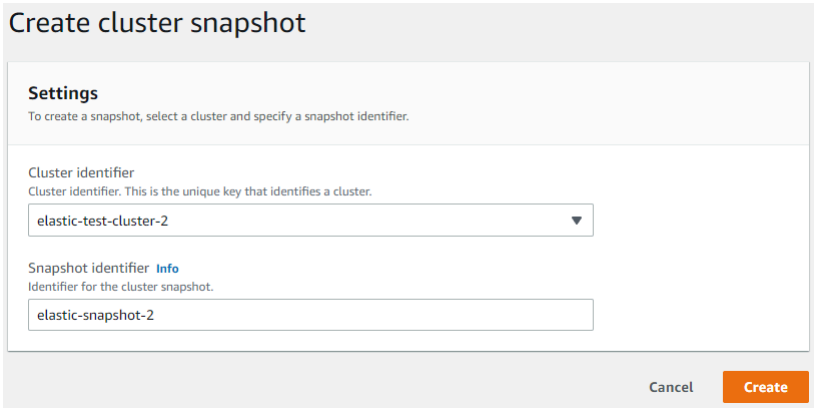
1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Snapshots**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. On the **Snapshots** page, choose **Create**.
4. On the **Create cluster snapshot** page, in the **Cluster identifier** field, choose your elastic cluster from the drop down list.

   In the **Snapshot identifier** field, enter a unique identifier for your elastic cluster.

   Choose **Create**.



> **Note**
>
> Alternatively, you can access the **Create cluster snapshot** dialog by going to the **Clusters** page, checking the box next to your cluster, then choosing **Actions**, then **Take snapshot**.

Your elastic cluster snapshot is now provisioning. This can take up to a few minutes to finish. You can view and restore from your snapshot when the status shows as `Available` in the **Snapshots** list.

Using the AWS CLI

To create a manual elastic cluster snapshot using the AWS CLI, use the `create-cluster-snapshot` operation with the following parameters:

- **`--snapshot-name`**—Required. The name of the cluster snapshot you want to create.
- **`--cluster-arn`**—Required. The ARN identifier of the cluster of which you want to create a snapshot.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic create-cluster-snapshot \
    --snapshot-name sample-snapshot-1 \
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-
cluster
```

For Windows:

```
aws docdb-elastic create-cluster-snapshot ^
    --snapshot-name sample-snapshot-1 ^
    --cluster-arn arn:aws:docdb:us-west-2:123456789012:sharded-cluster:sample-
cluster
```

Viewing an elastic cluster snapshot

In this section we explain how to view elastic cluster snapshot information, using either the AWS Management Console or AWS CLI with the following instructions.

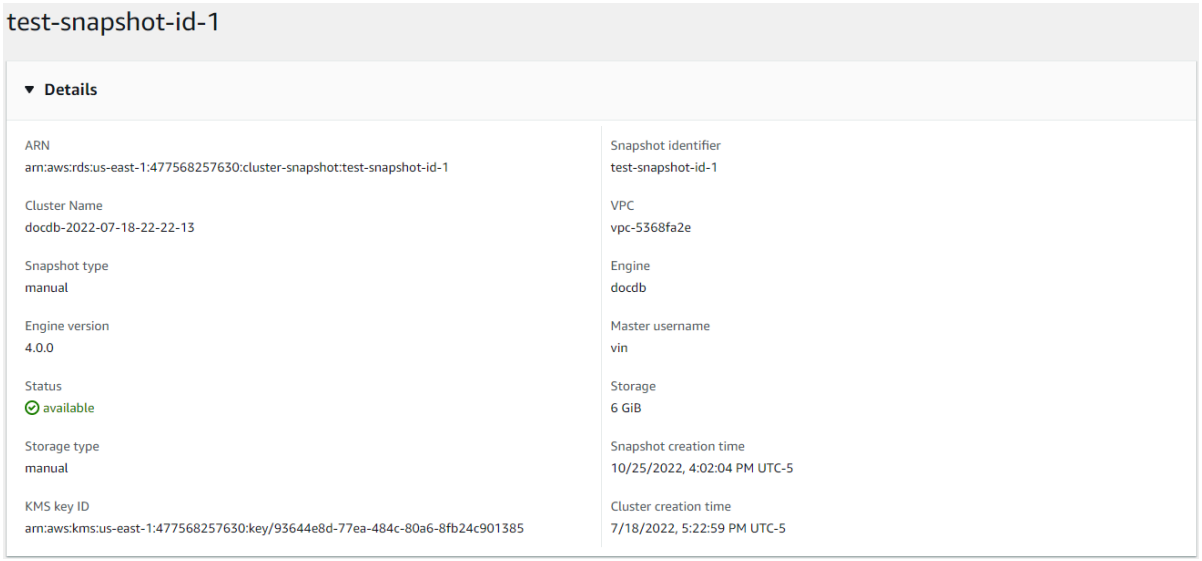
Using the AWS Management Console

To view information about a specific elastic cluster snapshot using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Snapshots**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. On the **Snapshots** page, choose your snapshot from the list by clicking on the name in **Snapshot identifier** column.
4. View your snapshot's information in **Details**.



Using the AWS CLI

To view information about a specific elastic cluster snapshot using the AWS CLI, use the `get-cluster-snapshot` operation with the following parameters:

- **`--snapshot-arn`**—Required. The ARN identifier of the snapshot for which you want information.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic get-cluster-snapshot \
    --snapshot-arn sampleResourceName
```

For Windows:

```
aws docdb-elastic get-cluster-snapshot ^
    --snapshot-arn sampleResourceName
```

To view information about a specific elastic cluster snapshot using the AWS CLI, use the `get-cluster-snapshot` operation with the following parameters:

- **`--snapshot-arn`**—Required. The ARN identifier of the snapshot for which you want information.

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic get-cluster-snapshot \
    --snapshot-arn sampleResourceName
```

For Windows:

```
aws docdb-elastic get-cluster-snapshot ^
    --snapshot-arn sampleResourceName
```

To view information about all elastic cluster snapshots using the AWS CLI, use the `list-cluster-snapshots` operation with the following parameters:

- **`--snapshot-type`**—Optional. The type of cluster snapshots to be returned. You can specify one of the following values:
  - `automated` - Return all cluster snapshots that Amazon DocumentDB has automatically created for your AWS account.
  - `manual` - Return all cluster snapshots that you have manually created for your AWS account.
  - `shared` - Return all manual cluster snapshots that have been shared to your AWS account.
  - `public` - Return all cluster snapshots that have been marked as public.
- **`--next-token`**—Optional. An optional pagination token provided by a previous request. If this parameter is specified, the response includes only records beyond the this token, up to the value specified by `max-results`.
- **`--max-results`**—Optional. The maximum number of results to include in the response. If more results exist than the specified `max-results` value, a pagination token (`next-token`) is included in the response so that the remaining results can be retrieved.
  - Default: 100
  - Minimum 20, maximum 100

In the following example, replace each *`user input placeholder`* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic list-cluster-snapshots \
    --snapshot-type  value \
    --next-token  value \
    --max-results 50
```

For Windows:

```
aws docdb-elastic list-cluster-snapshots ^
    --snapshot-type  value ^
    --next-token  value ^
    --max-results 50
```

Restoring an elastic cluster from a snapshot

In this section we explain how to restore an elastic cluster from a snapshot, using either the AWS Management Console or AWS CLI with the following instructions.

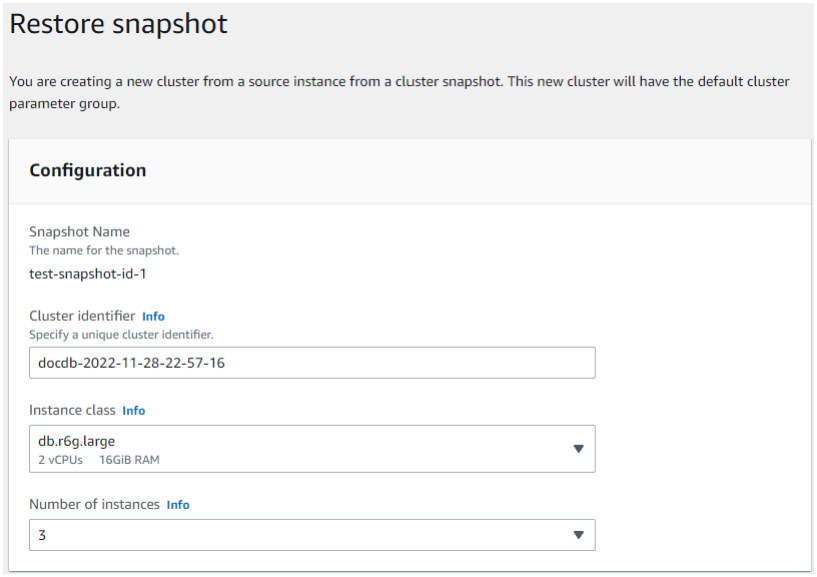
Using the AWS Management Console

To restore an elastic cluster from a snapshot using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Snapshots**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the button to the left of the snapshot, that you want to use to restore a cluster, in the **Snapshot identifier** column.
4. Choose **Actions**, then **Restore**.



5. On the **Restore snapshot** page, enter a name for the new cluster in the **Cluster identifier** field.

> **Note**
>
> For any manual snapshot restoration, you must create a new cluster.

6. In the **Virtual Private Cloud (VPC)** field, choose a VPC from the drop-down list.
7. For **Subnets** and **VPC security groups**, you can use the defaults or select three subnets of your choice and upto three VPC security groups (minimum one).
8. If you are satisfied with the cluster configuration, choose **Restore cluster** and wait while your cluster is restored.

Using the AWS CLI

To restore an elastic cluster from a snapshot using the AWS CLI, use the `restore-cluster-from-snapshot` operation with the following parameters:

- **`--cluster-name`**—Required. The current name of the elastic cluster as entered during creation or last modified.
- **`--snapshot-arn`**—Required. The ARN identifier of the snapshot being used to restore the cluster.
- **`--vpc-security-group-ids`**—Optional. One or more Amazon EC2 and Amazon Virtual Private Cloud (VPC) security groups to associate with the cluster.
- **`--kms-key-id`**—Optional. Configure the KMS key identifier for an encrypted cluster.

  The KMS key identifier is the Amazon Resource Name (ARN) for the AWS KMS encryption key. If you are creating a cluster using the same Amazon Web Services account that owns the KMS encryption key that is used to encrypt the new cluster, you can use the KMS key alias instead of the ARN for the KMS encryption key.

  If an encryption key is not specified in KmsKeyId and if the `StorageEncrypted` parameter is true, Amazon DocumentDB uses your default encryption key.
- **`--subnet-ids`**—Optional. Network subnet Ids.

In the following example, replace each *`user input placeholder`* with your own information..

For Linux, macOS, or Unix:

```
aws docdb-elastic restore-cluster-from-snapshot \
     --cluster-name elastic-sample-cluster \
     --snapshot-arn sampleResourceName \
     --vpc-security-group-ids value ec-65f40350 \
     --kms-key-id arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 \
     --subnet-ids subnet-9253c6a3, subnet-9f1b5af9
```

For Windows:

```
aws docdb-elastic restore-cluster-from-snapshot ^
     --cluster-name elastic-sample-cluster ^
     --snapshot-arn sampleResourceName ^
     --vpc-security-group-ids value ec-65f40350 ^
     --kms-key-id arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2 ^
     --subnet-ids subnet-9253c6a3, subnet-9f1b5af9
```

Copying an elastic cluster snapshot

In Amazon DocumentDB, you can copy manual and automatic elastic cluster snapshots within the same Region and within the same account. In this section we explain how to copy an elastic cluster snapshot, using either the AWS Management Console or AWS CLI.

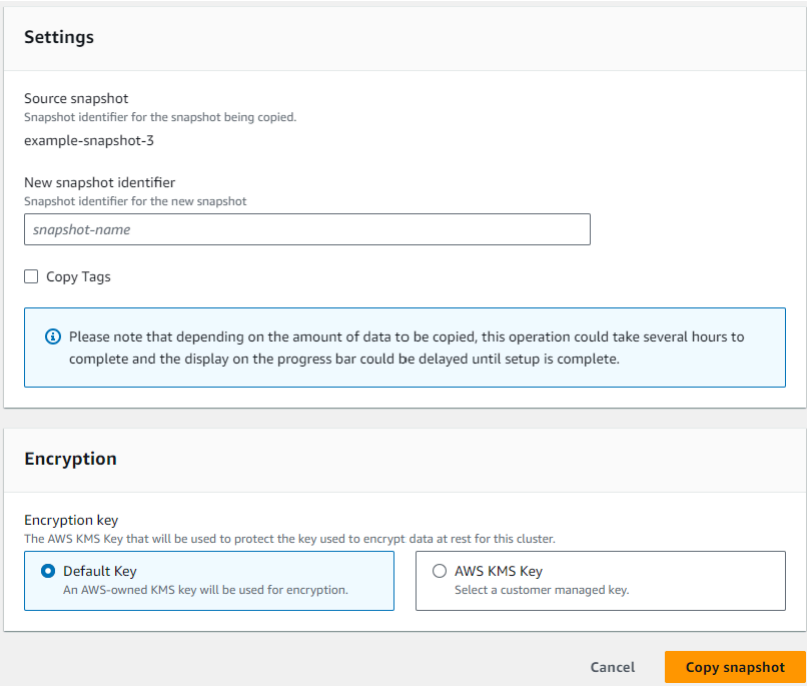
Using the AWS Management Console

To copy an elastic cluster snapshot using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Snapshots**.

   > **Tip**
   >
   > If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the button to the left of the snapshot, that you want to copy, in the **Snapshot identifier** column.
4. Choose **Actions**, then **Copy**.



5. For **New snapshot identifier**, enter the new snapshot's name.
6. For **Copy Tags**, check the box if you want to copy all tags from the source elastic cluster snapshot to the target elastic cluster snapshot.
7. For **Encryption**, choose either a default AWS KMS key or a KMS key of your choosing. The second option allows you to select an existing KMS key that you already created or allows you to create a new one.
8. Choose **Copy snapshot** when complete.

Using the AWS CLI

To copy an elastic cluster snapshot using the AWS CLI, use the `copy-cluster-snapshot` operation with the following parameters:

- **`--source-db-cluster-snapshot-identifier`**—Required. The identifier of the existing elastic cluster snapshot being copied. The elastic cluster snapshot must exist and be in the available state. This parameter is not case sensitive.
- **`--target-db-cluster-snapshot-identifier`**—Required. The identifier of the new elastic cluster snapshot to create from the existing cluster snapshot. This parameter is not case sensitive.

  Target snapshot name constraints:

- Cannot be the name of an existing snapshot.
- Length is [1—63] letters, numbers, or hyphens.
- First character must be a letter.
- Cannot end with a hyphen or contain two consecutive hyphens.

In the following example, replace each *user input placeholder* with your own information.

For Linux, macOS, or Unix:

```
aws docdb-elastic copy-cluster-snapshot \
     --source-cluster-snapshot-arn <sample ARN> \
     --target-cluster-snapshot-name my-target-copied-snapshot
```

For Windows:

```
aws docdb-elastic copy-cluster-snapshot ^
     --source-cluster-snapshot-arn <sample ARN> ^
     --target-cluster-snapshot-name my-target-copied-snapshot
```

Deleting an elastic cluster snapshot

In this section we explain how to delete an elastic cluster snapshot, using either the AWS Management Console or AWS CLI.

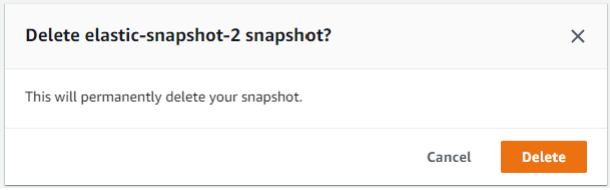
Using the AWS Management Console

To restore an elastic cluster from a snapshot using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Snapshots**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the button to the left of the snapshot, that you want to use to restore a cluster, in the **Snapshot identifier** column.
4. Choose **Actions**, then **Delete**.



5. On the **Delete "snapshot-name" snapshot** dialog, choose **Delete**.

Using the AWS CLI

To delete an elastic cluster snapshot using the AWS CLI, use the `delete-cluster-snapshot` operation with the following parameters:

- **`--snapshot-arn`**—Required. The ARN identifier of the snapshot being used to restore the cluster.

In the following example, replace each *`user input placeholder`* with your own information..

For Linux, macOS, or Unix:

```
aws docdb-elastic delete-cluster-snapshot \
     --snapshot-arn sampleResourceName
```

For Windows:

```
aws docdb-elastic delete-cluster-snapshot ^
     --snapshot-arn sampleResourceName
```

Managing an elastic cluster snapshot automatic backup

Amazon DocumentDB takes daily snapshots of your elastic clusters. You can specify the preferred backup window and the backup retention period in a new or existing elastic cluster snapshot configuration. In this section we explain how to set automatic backup parameters in an elastic cluster snapshot, using either the AWS Management Console or AWS CLI.

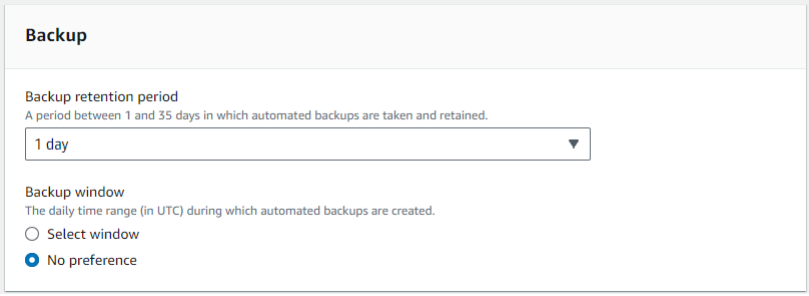
Using the AWS Management Console

To set an automatic backup for a new elastic cluster snapshot using the AWS Management Console:

1. Sign into the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon in the upper-left corner of the navigation pane.

3. Choose the button to the left of the cluster, that you want to change the backup settings for, in the **Cluster identifier** column.
4. Choose **Actions**, then **Modify**.
5. In the **Backup** section, edit the fields according to your backup requirements.



   a. **Backup retention period**—In the list, choose the number of days to keep automatic backups of this cluster before deleting them.

   b. **Backup window**—Set the daily time and duration during which Amazon DocumentDB is to make backups of this cluster.

      i. Choose **Select window** if you want to configure the time and duration when backups are created.

         Start time—In the first list, choose the start time hour (UTC) for starting your automatic backups. In the second list, choose the minute of the hour that you want automatic backups to begin.

         Duration—In the list, choose the number of hours to be allocated to creating automatic backups.

ii. Choose **No preference** if you want Amazon DocumentDB to choose the time and duration when backups are created.

6. Choose **Modify cluster** when complete.

Using the AWS CLI

To set an automatic backup for a new elastic cluster snapshot using the AWS CLI, use the `create-cluster-snapshot` operation with the following parameters:

- **`--preferred-backup-window`**—Optional. The daily preferred time range during which automated backups are created. The default is a 30-minute window selected at random from an 8-hour block of time for each AWS Region.

  Constraints:

  - Must be in the format `hh24:mi-hh24:mi`.
  - Must be in Universal Coordinated Time (UTC).
  - Must not conflict with the preferred maintenance window.
  - Must be at least 30 minutes.
- **`--backup-retention-period`**—Optional. The number of days for which automated backups are retained. The default value is 1.

  Constraints:

  - Must specify a minimum value of 1.
  - Range is from 1 to 35.

> **Note**
>
> Automated backups are only be taken when the cluster is in an ‘active’ state.

> **Note**
>
> You can also modify the `preferred-backup-window` and `backup-retention-period` parameters of an existing elastic cluster using the `aws docdb-elastic update-cluster` command.

In the following example, replace each *user input placeholder* with your own information.

The following `create-cluster` example creates the Amazon DocumentDB elastic cluster *sample-cluster* with the retention period for automatic backups of *7* days, and a preferred backup window of *18:00-18:30 UTC*.

For Linux, macOS, or Unix:

```
aws docdb-elastic create-cluster \
     --cluster-name sample-cluster \
     --shard-capacity 2 \
     --shard-count 2 \
     --admin-user-name SampleAdmin \
     --auth-type PLAIN_TEXT \
     --admin-user-password SamplePass123! \
     --preferred-backup-window 18:00-18:30 \
     --backup-retention-period 7
```

For Windows:

```
aws docdb-elastic create-cluster ^
     --cluster-name sample-cluster ^
     --shard-capacity 2 ^
     --shard-count 2 ^
     --admin-user-name SampleAdmin ^
     --auth-type PLAIN_TEXT ^
     --admin-user-password SamplePass123! ^
     --preferred-backup-window 18:00-18:30 ^
     --backup-retention-period 7
```

Stopping and starting an Amazon DocumentDB elastic cluster

Stopping and starting Amazon DocumentDB elastic clusters can help you manage costs for development and test environments. Instead of creating and deleting elastic clusters each time you use Amazon DocumentDB, you can temporarily stop your cluster when it isn't needed. You can then start it again when you resume your testing.

Topics

- Overview of stopping and starting an elastic cluster

- Operations you can perform on a stopped elastic cluster

Overview of stopping and starting an elastic cluster

During periods where you don't need an Amazon DocumentDB elastic cluster, you can stop the cluster. You can then start the cluster again anytime you need to use it. Starting and stopping simplifies the setup and teardown processes for elastic clusters that are used for development, testing, or similar activities that don't require continuous availability. You can stop and start an elastic cluster using the AWS Management Console or the AWS CLI with a single action.

While your elastic cluster is stopped, the cluster storage volume remains unchanged. You are charged only for storage, manual snapshots, and automated backup storage within your specified retention window. Amazon DocumentDB automatically starts your elastic cluster after seven days so that it doesn't fall behind any required maintenance updates. When your cluster starts after seven days, you will begin to be charged for the use of the elastic cluster again. While your cluster is stopped, you can't query your storage volume because querying requires that the cluster is in the available state.

When an Amazon DocumentDB elastic cluster is stopped, the cluster cannot be modified in any way. This includes deleting the cluster.

Using the AWS Management Console

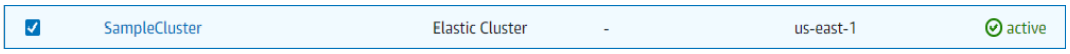
The following procedure shows you how to stop an elastic cluster in the available state, or start a stopped elastic cluster.

To stop or start an Amazon DocumentDB elastic cluster

1. Sign in to the AWS Management Console, and open the Amazon DocumentDB console at https://console.aws.amazon.com/docdb.
2. In the navigation pane, choose **Clusters**.

> **Tip**
>
> If you don't see the navigation pane on the left side of your screen, choose the menu icon (≡) in the upper-left corner of the page.

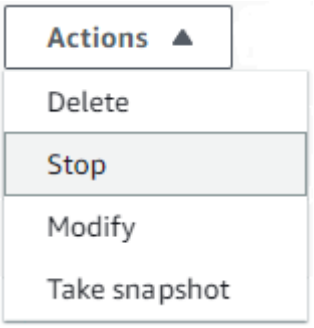
3. In the list of clusters, choose the button to the left of the name of the cluster that you want to stop or start.



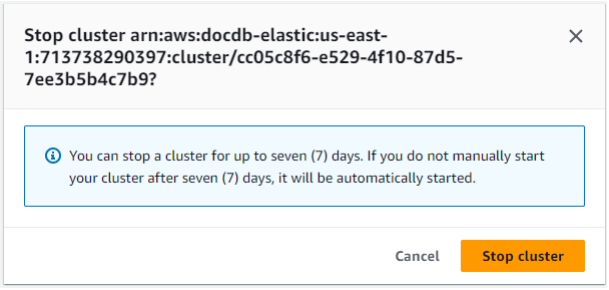
4. Choose **Actions**, and then choose the action that you want to perform on the cluster.

   - If you want to stop the cluster and the cluster is available:

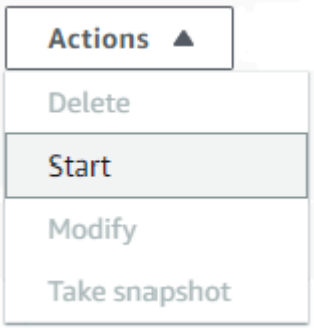
     a. Choose **Stop**.



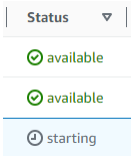
     b. On the confirmation dialog, confirm that you want to stop the elastic cluster by choosing **Stop cluster**, or to keep the cluster running, choose **Cancel**.



   - If you want to start the cluster, and the cluster is stopped, choose **Start**.



5. Monitor the status of the elastic cluster. If you started the cluster, you can resume using the cluster when the cluster is *available*. For more information, see Determining a cluster's status.



Using the AWS CLI

The following code examples show you how to stop an elastic cluster in the active or available state, or start a stopped elastic cluster.

To stop an elastic cluster using the AWS CLI, use the `stop-cluster` operation. To start a stopped cluster, use the `start-cluster` operation. Both operations use the `--cluster-arn` parameter.

Parameter:

- **`--cluster-arn`**—Required. The ARN identifier of the elastic cluster that you want to stop or start.

Example— To stop an elastic cluster using the AWS CLI

In the following example, replace each *`user input placeholder`* with your own information.

The following code stops the elastic cluster with an ARN of `arn:aws:docdb-elastic:us-east-1:477568257630:cluster/b9f1d489-6c3e-4764-bb42-da62ceb7bda2`.

> **Note**
>
> The elastic cluster must be in the active or available state.

For Linux, macOS, or Unix:

```
aws docdb-elastic stop-cluster \
   --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2
```

For Windows:

```
aws docdb-elastic stop-cluster ^
   --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2
```

Example— To start an elastic cluster using the AWS CLI

In the following example, replace each *`user input placeholder`* with your own information.

The following code starts the elastic cluster with an ARN of `arn:aws:docdb-elastic:us-east-1:477568257630:cluster/b9f1d489-6c3e-4764-bb42-da62ceb7bda2`.

> **Note**
>
> The elastic cluster must currently be stopped.

For Linux, macOS, or Unix:

```
aws docdb-elastic start-cluster \
   --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2
```

For Windows:

```
aws docdb-elastic start-cluster ^
   --cluster-arn arn:aws:docdb-elastic:us-east-1:477568257630:cluster/
b9f1d489-6c3e-4764-bb42-da62ceb7bda2
```

Operations you can perform on a stopped elastic cluster

You can't modify the configuration of an Amazon DocumentDB elastic cluster while the cluster is stopped. You must start the cluster before performing any such administrative actions.

Amazon DocumentDB applies any scheduled maintenance to your stopped elastic cluster only after it's started again. After seven days, Amazon DocumentDB automatically starts a stopped elastic cluster so that it doesn't fall too far behind in its maintenance status. When the elastic cluster restarts, you will begin to be charged for the shards in the cluster again.

While an elastic cluster is stopped, Amazon DocumentDB does not perform any automated backups nor does it extend the backup retention period.

Maintaining Amazon DocumentDB elastic clusters

Topics

- Viewing pending elastic cluster maintenance actions
- Elastic cluster engine updates
- Elastic cluster operating system updates

Periodically, Amazon DocumentDB performs maintenance on Amazon DocumentDB elastic cluster resources. Maintenance most often involves updates to the database engine (elastic cluster maintenance) or the elastic cluster's underlying operating system (OS updates). Database engine updates are required patches and include security fixes, bug fixes, and enhancements to the database engine. While most operating system patches are optional, if you don't apply them for a while, the patch may be required and auto applied to maintain your security posture. Apply operating system updates to your Amazon DocumentDB elastic clusters as soon as they are available.

Database engine patches require that you take your Amazon DocumentDB elastic clusters offline for a short time. Once available, these patches are automatically scheduled to apply during an upcoming scheduled maintenance window of your Amazon DocumentDB elastic cluster.

Elastic clusters have their own respective maintenance windows. Elastic cluster modifications that you have chosen not to apply immediately, are applied during the maintenance window. By default, when you create an elastic cluster, Amazon DocumentDB assigns a maintenance window for your elastic cluster. You can choose the maintenance window when creating an elastic cluster. You can also modify the maintenance windows at any time to fit your business schedules or practices. It is generally advised to choose maintenance windows that minimize the impact of the maintenance on your application (for example, on evenings or weekends).

Viewing pending elastic cluster maintenance actions

You can view whether a maintenance update is available for your elastic cluster by using the AWS CLI.

If an update is available, you can do one of the following:

- Defer a maintenance action that is currently scheduled for next maintenance window (for OS patches only).
- Apply the maintenance actions immediately.
- Schedule the maintenance actions to start during your next maintenance window.
- Schedule the maintenance actions to start during your selected apply-on window.

The maintenance window determines when pending operations start, but it does not limit the total execution time of these operations.

Use the following AWS CLI operation to determine what maintenance actions are pending. List all pending maintenance actions:

```
aws docdb-elastic list-pending-maintenance-actions
```

Output from this operation looks something like the following (JSON format):

```
{
'ResourcePendingMaintenanceActions': [
    {
        'ResourceArn': 'string-arn',
        'PendingMaintenanceActionDetails': [
            {
                'Action': 'ENGINE_UPDATE',
                'AutoAppliedAfterDate': 'string',
                'ForcedApplyDate': 'string',
                'OptInStatus': 'string',
                'CurrentApplyDate': 'string',
                'Description': 'string'
            },
        ]
    },
],
'NextToken': 'string'
}
```

Get pending maintenance action (if any) on a given `resourceArn`:

```
aws docdb-elastic get-pending-maintenance-action --resource-arn string-arn
```

Output from this operation looks something like the following (JSON format).

```
{
    'ResourcePendingMaintenanceAction': {
        'ResourceArn': 'string-arn',
        'PendingMaintenanceActionDetails': [
            {
                'Action': 'ENGINE_UPDATE',
                'AutoAppliedAfterDate': 'string',
                'ForcedApplyDate': 'string',
                'OptInStatus': 'string',
                'CurrentApplyDate': 'string',
                'Description': 'string'
            }
        ]
    }
}
```

Parameters:

- `ResourceArn`—The Amazon DocumentDB Amazon Resource Name (ARN) of the resource to which the pending maintenance action applies.
- `Action`—The pending maintenance action being applied to the resource.

  Valid values:

  - `ENGINE_UPDATE`
  - `ENGINE_UPGRADE`
  - `SECURITY_UPDATE`
  - `OS_UPDATE`
  - `MASTER_USER_PASSWORD_UPDATE`
- `AutoAppliedAfterDate`—First maintenance window after this date. `NEXT_MAINTENANCE OPT_IN` is ignored in this case.
- `ForcedApplyDate`—Applied regardless of maintenance window. `IMMEDIATE OPT_IN` is ignored in this case.
- `OptInStatus`—A value that specifies the type of opt-in request, or undoes an opt-in request. An opt-in request of type `IMMEDIATE` can't be undone.

  Valid values:

- `IMMEDIATE`—Apply the maintenance action immediately.
- `NEXT_MAINTENANCE`—Apply the maintenance action during the next maintenance window for the resource.
- `APPLY_ON`—Apply the maintenance action on specified apply date regardless of next maintenance window for the resource.
- `UNDO_OPT_IN`—Cancel any existing `NEXT_MAINTENANCE` or `APPLY_ON` opt in requests.

- `CurrentApplyDate`—Displayed if opt-in-type is `APPLY_ON`.
- `Description`—An option description for the maintenance action.

Elastic cluster engine updates

With Amazon DocumentDB, you can choose when to apply maintenance operations. You can decide when Amazon DocumentDB applies updates using the AWS CLI.

Apply pending maintenance actions:

```
aws docdb-elastic apply-pending-maintenance-action
--resource-arn string-arn
--apply-action string-enum
--opt-in-type string-enum
[--apply-on string-date-range]
```

Parameters:

- **`--resource-arn`**—The Amazon DocumentDB Amazon Resource Name (ARN) of the resource to which the pending maintenance action applies.
- **`--apply-action`**—The pending maintenance action to apply to this resource.

  Valid values:

  - `ENGINE_UPDATE`
  - `ENGINE_UPGRADE`
  - `SECURITY_UPDATE`
  - `OS_UPDATE`
  - `MASTER_USER_PASSWORD_UPDATE`
- **`--opt-in-type`**—A value that specifies the type of opt-in request, or undoes an opt-in request. An opt-in request of type `IMMEDIATE` can't be undone.

Valid values:

- `IMMEDIATE`—Apply the maintenance action immediately.
- `NEXT_MAINTENANCE`—Apply the maintenance action during the next maintenance window for the resource.
- `APPLY_ON`—Apply the maintenance action on specified apply date regardless of next maintenance window for the resource.
- `UNDO_OPT_IN`—Cancel any existing `NEXT_MAINTENANCE` or `APPLY_ON` opt in requests.

- **`[--apply-on]`**—Required if opt-in-type is `APPLY_ON`. Format: `yyyy/MM/dd HH:mm-yyyy/MM/dd HH:mm` (This option uses UTC time. The start time can be any time in the future from a minimum of 30 minutes and a maximum of 14 days, or force/apply date on pending action, whichever is earlier. Start to end time window can be a minimum of 30 minutes to a maximum of 8 hours long.)

Output from this operation looks something like the following (JSON format):

```
{
 'ResourcePendingMaintenanceAction': {
        'ResourceArn': 'string-arn',
        'PendingMaintenanceActionDetails': [
            {
                'Action': 'SECURITY_UPDATE',
                'AutoAppliedAfterDate': 'string',
                'ForcedApplyDate': 'string',
                'OptInStatus': 'IMMEDIATE',
                'CurrentApplyDate': 'string',
                'Description': 'string'
            },
        ]
 }
}
```

Parameters:

- `ResourceArn`—The Amazon DocumentDB Amazon Resource Name (ARN) of the resource to which the pending maintenance action applies.
- `Action`—The pending maintenance action being applied to the resource.

Valid values:

- ENGINE_UPDATE
- ENGINE_UPGRADE
- SECURITY_UPDATE
- OS_UPDATE
- MASTER_USER_PASSWORD_UPDATE

- AutoAppliedAfterDate—First maintenance window after this date. NEXT_MAINTENANCE OPT_IN is ignored in this case.
- ForcedApplyDate—Applied regardless of maintenance window. IMMEDIATE OPT_IN is ignored in this case.
- OptInStatus—A value that specifies the type of opt-in request, or undoes an opt-in request. An opt-in request of type IMMEDIATE can't be undone.

  Valid values:

  - IMMEDIATE—Apply the maintenance action immediately.
  - NEXT_MAINTENANCE—Apply the maintenance action during the next maintenance window for the resource.
  - APPLY_ON—Apply the maintenance action on specified apply date regardless of next maintenance window for the resource.
  - UNDO_OPT_IN—Cancel any existing NEXT_MAINTENANCE or APPLY_ON opt in requests.
- CurrentApplyDate—Displayed if opt-in-type is APPLY_ON.
- Description—An option description for the maintenance action.

Apply dates

Each maintenance action has a respective apply date that you can find when describing the pending maintenance actions. When you read the output of pending maintenance actions from the AWS CLI, three dates are listed:

- CurrentApplyDate—The date the maintenance action will get applied either immediately or during the next maintenance window. If the maintenance is optional, this value can be null.
- ForcedApplyDate—The date when the maintenance will be automatically applied, independent of your maintenance window.

- `AutoAppliedAfterDate`—The date after which the maintenance will be applied during the cluster's maintenance window.

User-created maintenance actions

As an Amazon DocumentDBelastic cluster user, you can initiate updates to your clusters configurations.

Updating cluster primary password

```
aws docdb-elastic update-cluster
--cluster-arn string-arn
[--admin-user-password string]
[--auth-type string-enum]
[--apply-method string-enum]
[--apply-on string-date-range]
#... other parameters of the API that follow here are not relevant for this
 configuration
```

Parameters:

- **`--cluster-arn`**—The Amazon DocumentDB Amazon Resource Name (ARN) of the resource to which the maintenance action will be applied.
- **`[--admin-user-password]`**—The password associated with the admin user.
- **`[--auth-type]`**—The authentication type used to determine where to fetch the password used for accessing the elastic cluster. Valid types are `PLAIN_TEXT` or `SECRET_ARN`.
- **`[--apply-method]`**—A value that specifies the type of method being applied. Allowed values are `IMMEDIATE` and `APPLY_ON`. Default is `IMMEDIATE`.
- **`[--apply-on]`**—Required if `apply-method` is `APPLY_ON`. Format: `yyyy/MM/dd HH:mm-yyyy/MM/dd HH:mm` (This option uses UTC time. The start time can be any time in the future from a minimum of 30 minutes and a maximum of 14 days. Start to end time window can be a minimum of 30 minutes to a maximum of 8 hours long.)

Output from this operation looks something like the following (JSON format):

```
{
 'ResourcePendingMaintenanceAction': {
        'ResourceArn': 'string-arn',
```

```
        'PendingMaintenanceActionDetails': [
            {
                'Action': 'MASTER_USER_PASSWORD_UPDATE',
                'OptInStatus': 'APPLY_ON',
                'CurrentApplyDate': 'string',
                'Description': 'string'
            },
        ]
 }
}
```

Changing your Amazon DocumentDB maintenance windows

The maintenance window should fall at the time of lowest usage and thus might need changing from time to time. Your elastic cluster is unavailable during this time only if system changes (such as a scale storage operation change) are being applied and require an outage. It is unavailable only for the minimum amount of time required to make the necessary changes.

The default is a 30-minute window selected at random from an 8-hour block of time for each Amazon Web Services Region, occurring on a random day of the week.

To change the maintenance window, see Modifying elastic cluster configurations.

Elastic cluster operating system updates

Amazon DocumentDB elastic clusters occasionally require operating system updates. Amazon DocumentDB upgrades the operating system to a newer version to improve database performance and customers' overall security posture. Operating system updates don't change the cluster engine version of an Amazon DocumentDB elastic cluster.

Most operating system updates for Amazon DocumentDB elastic clusters are optional and don't have a set date to apply them. However, if you don't apply these updates for a while, they may eventually become required and automatically applied during your clusters maintenance window. This is to help maintain the security posture of your database. To avoid unexpected downtime, apply operating system updates to your Amazon DocumentDB elastic cluster as soon as they become available, and set your cluster maintenance window at a time that suits your business needs.

Data encryption at rest for Amazon DocumentDB elastic clusters

The following topics help you learn about, create, and monitor AWS Key Management Service encryption keys for Amazon DocumentDB elastic clusters:

Topics

- How Amazon DocumentDB elastic clusters use grants in AWS KMS
- Create a customer managed key
- Monitoring your encryption keys for Amazon DocumentDB elastic clusters
- Learn more

Amazon DocumentDB elastic clusters automatically integrate with AWS Key Management Service (AWS KMS) for key management and uses a method known as envelope encryption to protect your data. For more information about envelope encryption, see Envelope encryption in the *AWS Key Management Service Developer Guide*.

An AWS KMS key is a logical representation of a key. The KMS key includes metadata, such as the key ID, creation date, description, and key state. The KMS key also contains the key material used to encrypt and decrypt data. For more information about KMS keys, see AWS KMS keys in the *AWS Key Management Service Developer Guide*.

Amazon DocumentDB elastic clusters support encryption with two types of keys:

- **AWS owned keys —** Amazon DocumentDB elastic clusters use these keys by default to automatically encrypt personally identifiable data. You can't view, manage, or use AWS-owned keys, or audit their use. However, you don't have to take any action or change any programs to protect the keys that encrypt your data. For more information, see AWS owned keys in the *AWS Key Management Service Developer Guide*.
- **Customer-managed keys —** Symmetric AWS KMS keys that you create, own, and manage. Because you have full control of this layer of encryption, you can perform such tasks as:
  - Establishing and maintaining key policies
  - Establishing and maintaining IAM policies and grants
  - Enabling and disabling key policies
  - Rotating key cryptographic material

- Adding tags
- Creating key aliases
- Scheduling keys for deletion

For more information, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

Important

You must use a symmetric encryption KMS key to encrypt your cluster as Amazon DocumentDB supports only symmetric encryption KMS keys. Do not use an asymmetric KMS key to attempt to encrypt the data in your Amazon DocumentDB elastic clusters. For more information, see Asymmetric keys in AWS KMS in the *AWS Key Management Service Developer Guide*.

If Amazon DocumentDB can no longer gain access to the encryption key for a cluster — for example, when access to a key is revoked — the encrypted cluster goes into a terminal state. In this case, you can only restore the cluster from a backup. For Amazon DocumentDB, backups are always enabled for 1 day. In addition, if you disable the key for an encrypted Amazon DocumentDB cluster, you will eventually lose read and write access to that cluster. When Amazon DocumentDB encounters a cluster that is encrypted by a key that it doesn't have access to, it puts the cluster into a terminal state. In this state, the cluster is no longer available, and the current state of the database can't be recovered. To restore the cluster, you must re-enable access to the encryption key for Amazon DocumentDB, and then restore the cluster from a backup.

Important

You cannot change the KMS key for an encrypted cluster after you have already created it. Be sure to determine your encryption key requirements before you create your encrypted elastic cluster.

How Amazon DocumentDB elastic clusters use grants in AWS KMS

Amazon DocumentDB elastic clusters require a grant to use your customer managed key.

When you create a cluster encrypted with a customer managed key, Amazon DocumentDB elastic clusters create a grant on your behalf by sending a `CreateGrant` request to AWS KMS. Grants in AWS KMS are used to give Amazon DocumentDB elastic clusters access to a KMS key in a customer account.

Amazon DocumentDB elastic clusters require the grant to use your customer managed key for the following internal operations:

- Send `DescribeKey` requests to AWS KMS to verify that the symmetric customer managed key ID, entered when creating an elastic cluster, is valid.
- Send `GenerateDataKey` requests to AWS KMS to generate data keys encrypted by your customer managed key.
- Send `Decrypt` requests to AWS KMS to decrypt the encrypted data keys so that they can be used to encrypt your data.
- You can revoke access to the grant, or remove the service's access to the customer managed key at any time. If you do, Amazon DocumentDB elastic clusters won't be able to access any of the data encrypted by the customer managed key, which affects operations that are dependent on that data.

Create a customer managed key

You can create a symmetric customer managed key by using the AWS Management Console or the AWS KMS API.

Symmetric customer managed key creation

Follow the steps for Creating a symmetric customer managed key in the *AWS Key Management Service Developer Guide*.

Key policy

Key policies control access to your customer managed key. Every customer managed key must have exactly one key policy, which contains statements that determine who can use the key and how they can use it. When you create your customer managed key, you can specify a key policy. For more information, see the KMS key access information located in the AWS Key Management Service overview of the *AWS Key Management Service Developer Guide*.

To use your customer managed key with Amazon DocumentDB elastic cluster resources, the following API operations must be permitted in the key policy:

- `kms:CreateGrant` – Adds a grant to a customer managed key. Grants control access to a specified KMS key, which allows access to the grant operations that Amazon DocumentDB elastic clusters require. For more information about using grants, see Grants in AWS KMS in the *AWS Key Management Service Developer Guide*.
- `kms:DescribeKey` – Provides the customer managed key details to allow Amazon DocumentDB elastic clusters to validate the key.
- `kms:Decrypt` – Allows Amazon DocumentDB elastic clusters to use the stored encrypted data key to access encrypted data.
- `kms:GenerateDataKey` – Allows Amazon DocumentDB elastic clusters to generate an encrypted data key and store it because the data key isn't immediately used to encrypt.

For more information, see Permissions for AWS services in key policies and Troubleshooting key access in the *AWS Key Management Service Developer Guide*.

Restricting customer managed key access via IAM policies

In addition to KMS key policies, you can also restrict KMS key permissions in an IAM policy.

You can make the IAM policy stricter in various ways. For example, to allow the customer managed key to be used only for requests that originate in Amazon DocumentDB elastic clusters, you can use the `kms:ViaService` condition key with the `docdb-elastic.<region-name>.amazonaws.com` value.

For more information, see Allowing users in other accounts to use a KMS key in the *AWS Key Management Service Developer Guide*.

Monitoring your encryption keys for Amazon DocumentDB elastic clusters

When you use a customer managed key with your Amazon DocumentDB elastic cluster resources, you can use AWS CloudTrail or Amazon CloudWatch Logs to track requests that Amazon DocumentDB elastic clusters send to AWS KMS.

The following examples are AWS CloudTrail events for `CreateGrant`, `GenerateDataKeyWithoutPlainText`, `Decrypt`, and `DescribeKey` to monitor AWS KMS key operations called by Amazon DocumentDB elastic clusters to access data encrypted by your customer managed key:

CreateGrant


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:assumed-role/Admin/Sampleuser01",
                "accountId": "111122223333",
                "userName": "Sampleuser01"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2023-05-09T23:04:20Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "docdb-elastic.amazonaws.com"
    },
    "eventTime": "2023-05-09T23:55:48Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "CreateGrant",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "docdb-elastic.amazonaws.com",
    "userAgent": "docdb-elastic.amazonaws.com",
    "requestParameters": {
        "retiringPrincipal": "docdb-elastic.us-east-1.amazonaws.com",
        "granteePrincipal": "docdb-elastic.us-east-1.amazonaws.com",
        "operations": [
            "Decrypt",
            "Encrypt",
            "GenerateDataKey",
            "GenerateDataKeyWithoutPlaintext",
            "ReEncryptFrom",
            "ReEncryptTo",
            "CreateGrant",
            "RetireGrant",
```

```
            "DescribeKey"
        ],
        "keyId": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
    },
    "responseElements": {
        "grantId":
 "0ab0ac0d0b000f00ea00cc0a0e00fc00bce000c000f0000000c0bc0a0000aaafSAMPLE",
        "keyId": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
    },
    "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "readOnly": false,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

GenerateDataKey

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:assumed-role/Admin/Sampleuser01",
```

```
                "accountId": "111122223333",
                "userName": "Sampleuser01"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2023-05-10T18:02:59Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "docdb-elastic.amazonaws.com"
    },
    "eventTime": "2023-05-10T18:03:25Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "docdb-elastic.amazonaws.com",
    "userAgent": "docdb-elastic.amazonaws.com",
    "requestParameters": {
        "keySpec": "AES_256",
        "keyId": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
    },
    "responseElements": null,
    "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```


Decrypt


```
{
```

```
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:assumed-role/Admin/Sampleuser01",
                "accountId": "111122223333",
                "userName": "Sampleuser01"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2023-05-10T18:05:49Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "docdb-elastic.amazonaws.com"
    },
    "eventTime": "2023-05-10T18:06:19Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "docdb-elastic.amazonaws.com",
    "userAgent": "docdb-elastic.amazonaws.com",
    "requestParameters": {
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
        }
    ],
```

```
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```

DescribeKey

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROAIGDTESTANDEXAMPLE:Sampleuser01",
        "arn": "arn:aws:sts::111122223333:assumed-role/Admin/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE3",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROAIGDTESTANDEXAMPLE",
                "arn": "arn:aws:iam::111122223333:assumed-role/Admin/Sampleuser01",
                "accountId": "111122223333",
                "userName": "Sampleuser01"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2023-05-09T23:04:20Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "docdb-elastic.amazonaws.com"
    },
    "eventTime": "2023-05-09T23:55:48Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "docdb-elastic.amazonaws.com",
    "userAgent": "docdb-elastic.amazonaws.com",
    "requestParameters": {
        "keyId": "alias/SampleKmsKey"
    },
    "responseElements": null,
    "requestID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
```

```
    "eventID": "ff000af-00eb-00ce-0e00-ea000fb0fba0SAMPLE",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-
east-1:111122223333:key/1234abcd-12ab-34cd-56ef-123456SAMPLE"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}
```


Learn more

The following resources provide more information about data encryption at rest:

- For more information about AWS KMS concepts, see AWS Key Management Service basic concepts in the *AWS Key Management Service Developer Guide*.
- For more information about AWS KMS security, see Security best practices for AWS Key Management Service in the *AWS Key Management Service Developer Guide*.

Service-linked roles in elastic clusters

Amazon DocumentDB elastic clusters use AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Amazon DocumentDB elastic clusters. Service-linked roles are predefined by Amazon DocumentDB elastic clusters and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes using Amazon DocumentDB elastic clusters easier because you don't have to manually add the necessary permissions. Amazon DocumentDB elastic clusters defines the permissions of its service-linked roles, and unless defined otherwise, only Amazon DocumentDB elastic clusters can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity. You can delete the roles only after first deleting their related resources. This protects your Amazon

DocumentDB elastic clusters resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that are marked with **Yes** in the **Service-Linked Role** column. Choose a Yes with a link to view the service-linked role documentation for that service.

Service-linked role permissions for elastic clusters

Amazon DocumentDB elastic clusters uses the service-linked role named `AWSServiceRoleForDocDB-Elastic` to allow Amazon DocumentDB elastic clusters to call AWS services on behalf of your clusters.

This service-linked role has a permissions policy attached to it called `AmazonDocDB-ElasticServiceRolePolicy` that grants it permissions to operate in your account. The role permissions policy allows Amazon DocumentDB elastic clusters to complete the following actions on the specified resources:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "cloudwatch:namespace": [
                        "AWS/DocDB-Elastic"
                    ]
                }
            }
        }
    ]
}
```

Note

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. If you encounter the following error message: **"Unable to create the resource. Verify that you have permission to create service linked role. Otherwise wait and try again later."**, make sure you have the following permissions enabled:

```
{
"Action": "iam:CreateServiceLinkedRole",
    "Effect": "Allow",
    "Resource": "arn:aws:iam::*:role/aws-service-role/docdb-elastic.amazonaws.com/
AWSServiceRoleForDocDB-Elastic",
    "Condition": {
"StringLike": {
"iam:AWSServiceName":"docdb-elastic.amazonaws.com"
        }
    }
}
```

For more information, see Service-linked role permissions in the *IAM User Guide*.

Creating a service-linked role for Amazon DocumentDB elastic clusters

You don't need to manually create a service-linked role. When you create a DB instance, Amazon DocumentDB elastic clusters creates the service-linked role for you.

Editing a service-linked role for Amazon DocumentDB elastic clusters

Amazon DocumentDB elastic clusters do not allow you to edit the `AWSServiceRoleForDocDB-Elastic` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for Amazon DocumentDB elastic clusters

If you no longer need to use a feature or service that requires a service-linked role, delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must delete all of your clusters before you can delete the service-linked role.

Cleaning up a service-linked role

Before you can use IAM to delete a service-linked role, you must first confirm that the role has no active sessions and remove any resources used by the role.

To check whether the service-linked role has an active session in the IAM console:

1. Sign in to the AWS Management Console and open the IAM console.
2. In the navigation pane of the IAM console, choose **Roles**. Then choose the name (not the check box) of the `AWSServiceRoleForDocDB-Elastic` role.
3. On the **Summary** page for the chosen role, choose the **Access Advisor** tab.

> **Note**
>
> If you are unsure whether Amazon DocumentDB elastic clusters is using the `AWSServiceRoleForDocDB-Elastic` role, you can try to delete the role. If the service is using the role, then the deletion fails and you can view the AWS Regions where the role is being used. If the role is being used, then you must wait for the session to end before you can delete the role. You cannot revoke the session for a service-linked role.
> If you want to remove the `AWSServiceRoleForDocDB-Elastic` role, you must first delete all of your clusters.

Deleting all of your clusters

To delete a cluster in the Amazon DocumentDB console:

1. Sign in to the AWS Management Console and open the Amazon DocumentDB console.
2. In the navigation pane, choose **Clusters**.
3. Choose the cluster that you want to delete.
4. For **Actions**, choose **Delete**.
5. If you are prompted to **Create final Snapshot?**, choose **Yes** or **No**.

6. If you chose **Yes** in the previous step, for **Final snapshot name** enter the name of your final snapshot.
7. Choose **Delete**.

> **Note**
>
> You can use the IAM console, the IAM CLI, or the IAM API to delete the `AWSServiceRoleForDocDB-Elastic` service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Amazon DocumentDB quotas

This topic describes the resource quotas and naming constraints for Amazon DocumentDB (with MongoDB compatibility).

For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS) and Amazon Neptune.

Topics

- Supported instance types
- Supported Regions
- Regional quotas
- Aggregation quotas
- Cluster quotas
- Instance quotas
- Naming constraints
- TTL constraints
- Elastic cluster quotas
- Elastic cluster shard quotas
- Elastic cluster CPU, memory, connection, and cursor quotas per shard

Supported instance types

Amazon DocumentDB supports on-demand instances and the following instance types:

- NVMe-backed:
  - **R6GD instance types:** `db.r6gd.xlarge`, `db.r6gd.2xlarge`, `db.r6gd.4xlarge`, `db.r6gd.8xlarge`, `db.r6gd.12xlarge`, `db.r6gd.16xlarge`.
- Memory optimized:
  - **R8G instance types:** `db.r8g.large`, `db.r8g.xlarge`, `db.r8g.2xlarge`, `db.r8g.4xlarge`, `db.r8g.8xlarge`, `db.r8g.12xlarge`, `db.r8g.16xlarge`, `db.r8g.24xlarge`, `db.r8g.48xlarge`.

  - **R6G instance types:** `db.r6g.large`, `db.r6g.xlarge`, `db.r6g.2xlarge`, `db.r6g.4xlarge`, `db.r6g.8xlarge`, `db.r6g.12xlarge`, `db.r6g.16xlarge`.
  - **R5 instance types:** `db.r5.large`, `db.r5.xlarge`, `db.r5.2xlarge`, `db.r5.4xlarge`, `db.r5.8xlarge`, `db.r5.12xlarge`, `db.r5.16xlarge`.
  - **R4 instance types:** `db.r4.large`, `db.r4.2xlarge`, `db.r4.4xlarge`, `db.r4.8xlarge`, `db.r4.16xlarge`.
- Burstable performance:
  - **T4G instance types:** `db.t4g.medium`.
  - **T3 instance types:** `db.t3.medium`.

For more information on the supported instance types and their specifications, see Instance class specifications.

Supported Regions

Amazon DocumentDB is available in the following AWS Regions:

Region Name	Region	Availability Zones (compute)
US East (Ohio)	us-east-2	3
US East (N. Virginia)	us-east-1	6
US West (Oregon)	us-west-2	4
Africa (Cape Town)	af-south-1	3
South America (São Paulo)	sa-east-1	3
Asia Pacific (Hong Kong)	ap-east-1	3
Asia Pacific (Taipei)	ap-east-2	3
Asia Pacific (Hyderabad)	ap-south-2	3
Asia Pacific (Malaysia)	ap-southeast-5	3
Asia Pacific (Mumbai)	ap-south-1	3
Asia Pacific (Osaka)	ap-northeast-3	3
Asia Pacific (Seoul)	ap-northeast-2	4
Asia Pacific (Singapore)	ap-southeast-1	3
Asia Pacific (Sydney)	ap-southeast-2	3
Asia Pacific (Jakarta)	ap-southeast-3	3

Region Name	Region	Availability Zones (compute)
Asia Pacific (Melbourne)	ap-southeast-4	3
Asia Pacific (New Zealand)	ap-southeast-6	3
Asia Pacific (Thailand)	ap-southeast-7	3
Asia Pacific (Tokyo)	ap-northeast-1	3
Canada (Central)	ca-central-1	3
Canada West (Calgary)	ca-west-1	3
China (Beijing) Region	cn-north-1	3
China (Ningxia)	cn-northwest-1	3
Europe (Frankfurt)	eu-central-1	3
Europe (Zurich)	eu-central-2	3
Europe (Ireland)	eu-west-1	3
Europe (London)	eu-west-2	3
Europe (Milan)	eu-south-1	3
Europe (Paris)	eu-west-3	3
Europe (Spain)	eu-south-2	3
Europe (Stockholm)	eu-north-1	3
Mexico (Central)	mx-central-1	3
Middle East (UAE)	me-central-1	3

Region Name	Region	Availability Zones (compute)
Israel (Tel Aviv)	`il-central-1`	3
AWS GovCloud (US-West)	`us-gov-west-1`	3
AWS GovCloud (US-East)	`us-gov-east-1`	3

Regional quotas

For certain management features, Amazon DocumentDB uses operational technology that is shared with Amazon Relational Database Service (Amazon RDS). The following table contains Regional quotas that are shared among Amazon DocumentDB and Amazon RDS.

> **Note**
>
> The Amazon RDS shared technology described above only applies to Amazon DocumentDB instance-based clusters. Amazon DocumentDB elastic clusters do not share technology with Amazon RDS.

The following quotas apply to Amazon DocumentDB instance-based clusters and are per AWS account per Region.

Resource	AWS default quota
Clusters	40
Cluster parameter groups	50
Clusters per parameter group	100
Instances per parameter group	300
Event subscriptions	20

Resource	AWS default quota
Instances	40
Manual cluster snapshots	100
Read replicas per cluster	15
Subnet groups	50
Subnets per subnet group	20
Tags per resource	50
VPC security groups per instance	5

The following quotas apply to Amazon DocumentDB elastic clusters and are per AWS account per Region.

Resource	AWS default quota
Elastic clusters	20
Elastic clusters vCPU	1024
Manual elastic cluster snapshot	20

You can use Service Quotas to request an increase for a quota, if the quota is adjustable. Some requests are automatically resolved, while others are submitted to Support. You can track the status of a quota increase request that is submitted to Support. Requests to increase service quotas do not receive priority support. If you have an urgent request, contact Support. For more information on Service Quotas, see What Is Service Quotas?

To request a quota increase for Amazon DocumentDB:

1. Open the Service Quotas console at https://console.aws.amazon.com/servicequotas and, if necessary, sign in.
2. In the navigation pane, choose **AWS services**.

3. Select Amazon DocumentDB (with MongoDB compatibility) or Amazon DocumentDB Elastic Cluster from the list, or type either in the search field.
4. If the quota is adjustable, you can select its radio button or its name, and then choose **Request quota increase** from the top right of the page.
5. For **Change quota value**, enter the new value. The new value must be greater than the current value.
6. Choose **Request**. After the request is resolved, the **Applied quota value** for the quota is set to the new value.
7. To view any pending or recently resolved requests, choose **Dashboard** from the navigation pane. For pending requests, choose the status of the request to open the request receipt. The initial status of a request is `Pending`. After the status changes to `Quota requested`, you'll see the case number with Support. Choose the case number to open the ticket for your request.

Aggregation quotas

The following table describes aggregation quotas in Amazon DocumentDB.

Resource	Quota
Maximum number of supported stages	500

Cluster quotas

The following table describes Amazon DocumentDB instance-based cluster quotas.

Resource	Quota
Cluster size (sum of all collections and indexes)	256 TiB for Engine Version 8.0 and beyond (128 TiB for earlier engine versions)
Collection size (sum of all collections cannot exceed cluster quota) – does not include the index size	32 TiB
Collections per cluster	100,000
Databases per cluster	100,000
Database size (sum of all databases cannot exceed cluster quota)	256 TiB for Engine Version 8.0 and beyond (128 TiB for earlier engine versions)
Document nesting depth	200 levels
Document size	16 MiB
Index key size	2,048 bytes
Indexes per collection	64
Keys in a compound index	32

Resource	Quota
Maximum number of writes in a single batch command	100,000
Number of users per cluster	1000

Instance quotas

The following table describes Amazon DocumentDB quotas per instance.

Amazon DocumentDB 8.0

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
T3.medium	4	1000	30	50	102
T4G.medium	4	1000	30	50	102
R5.large	16	3400	450	200	1100
R5.xlarge	32	7000	450	400	2700
R5.2xlarge	64	14200	450	800	4500
R5.4xlarge	128	28400	760	1600	4500
R5.8xlarge	256	60000	1520	3200	4500
R5.12xlarge	384	60000	2280	4800	4500
R5.16xlarge	512	60000	3040	6400	4500
R6G.large*	16	3400	450	200	1100
R6G.xlarge*	32	7000	450	400	2700
R6G.2xlarge*	64	14200	450	800	4500
R6G.4xlarge*	128	28400	760	1600	4500
R6G.8xlarge*	256	60000	1520	3200	4500
R6G.12xlarge*	384	60000	2280	4800	4500

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
R6G.16xlarge*	512	60000	3040	6400	4500
R8G.large	16	3400	450	200	1100
R8G.xlarge	32	7000	450	400	2700
R8G.2xlarge	64	14200	450	800	4500
R8G.4xlarge	128	28400	760	1600	4500
R8G.8xlarge	256	60000	1520	3200	4500
R8G.12xlarge	384	60000	2280	4800	4500
R8G.16xlarge	512	60000	3040	6400	4500
R8G.24xlarge	768	60000	4560	9600	4500
R8G.48xlarge	1536	60000	9120	9600	4500

* including R6GD

Amazon DocumentDB 5.0

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
T3.medium	4	1000	30	50	102

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
T4G.medium	4	1000	30	50	102
R5.large	16	3400	450	200	1100
R5.xlarge	32	7000	450	400	2700
R5.2xlarge	64	14200	450	800	4500
R5.4xlarge	128	28400	760	1600	4500
R5.8xlarge	256	60000	1520	3200	4500
R5.12xlarge	384	60000	2280	4800	4500
R5.16xlarge	512	60000	3040	6400	4500
R6G.large*	16	3400	450	200	1100
R6G.xlarge*	32	7000	450	400	2700
R6G.2xlarge*	64	14200	450	800	4500
R6G.4xlarge*	128	28400	760	1600	4500
R6G.8xlarge*	256	60000	1520	3200	4500
R6G.12xlarge*	384	60000	2280	4800	4500
R6G.16xlarge*	512	60000	3040	6400	4500
R8G.large	16	3400	450	200	1100
R8G.xlarge	32	7000	450	400	2700
R8G.2xlarge	64	14200	450	800	4500

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
R8G.4xlarge	128	28400	760	1600	4500
R8G.8xlarge	256	60000	1520	3200	4500
R8G.12xlarge	384	60000	2280	4800	4500
R8G.16xlarge	512	60000	3040	6400	4500
R8G.24xlarge	768	60000	4560	9600	4500
R8G.48xlarge	1536	60000	9120	9600	4500

* including R6GD

Amazon DocumentDB 4.0

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
T3.medium	4	1000	30	50	102
T4G.medium	4	1000	30	50	102
R5.large	16	3400	450	200	1100
R5.xlarge	32	7000	450	400	2700
R5.2xlarge	64	14200	450	800	4500
R5.4xlarge	128	28400	760	1600	4500

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
R5.8xlarge	256	60000	1520	3200	4500
R5.12xlarge	384	60000	2280	4800	4500
R5.16xlarge	512	60000	3040	6400	4500
R6G.large*	16	3400	450	200	1100
R6G.xlarge*	32	7000	450	400	2700
R6G.2xlarge*	64	14200	450	800	4500
R6G.4xlarge*	128	28400	760	1600	4500
R6G.8xlarge*	256	60000	1520	3200	4500
R6G.12xlarge*	384	60000	2280	4800	4500
R6G.16xlarge*	512	60000	3040	6400	4500

* including R6GD

Amazon DocumentDB 3.6

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
T3.medium	4	500	30	N/A	102
R4.large	15.25	1700	450	N/A	1100
R4.xlarge	30.5	3400	450	N/A	2700

Instance type	Instance memory (GiB)	Connections (all)	Cursor quota	Open transactions	Connections (active)
R4.2xlarge	61	6800	450	N/A	4500
R4.4xlarge	122	13600	725	N/A	4500
R4.8xlarge	288	27200	1450	N/A	4500
R4.16xlarge	488	30000	2900	N/A	4500
R5.large	16	1700	450	N/A	1100
R5.xlarge	32	3500	450	N/A	2700
R5.2xlarge	64	7100	450	N/A	4500
R5.4xlarge	128	14200	760	N/A	4500
R5.8xlarge	256	28400	1520	N/A	4500
R5.12xlarge	384	30000	2280	N/A	4500
R5.16xlarge	512	30000	3040	N/A	4500

You can monitor and alarm on the per instance limits using the following CloudWatch metrics. For more on Amazon DocumentDB CloudWatch metrics, see Monitoring Amazon DocumentDB with CloudWatch.

Resource	CloudWatch quota metric	CloudWatch usage metric (1-min. max)	CloudWatch usage metric
Instance Memory	-	-	FreeableMemory
Connections (all)	DatabaseConnectionsLimit	DatabaseConnectionsMax	DatabaseConnections
Cursors	DatabaseCursorsLimit	DatabaseCursorsMax	DatabaseCursors

Resource	CloudWatch quota metric	CloudWatch usage metric (1-min. max)	CloudWatch usage metric
Transactions	TransactionsOpenLimit	TransactionsOpenMax	TransactionsOpen

Naming constraints

The following table describes naming constraints in Amazon DocumentDB.

Resource	Default quota
Cluster identifier	• Length is [1–63] letters, numbers, or hyphens. • First character must be a letter. • Cannot end with a hyphen or contain two consecutive hyphens. • Must be unique for all clusters (across Amazon RDS, Amazon Neptune, and Amazon DocumentDB) per AWS account, per Region.
Instance identifier	• Length is [1–63] letters, numbers, or hyphens • First character must be a letter • Cannot end with a hyphen or contain two consecutive hyphens • Must be unique for all instances (across Amazon RDS, Amazon Neptune, and Amazon DocumentDB) per AWS account, per Region.
Collection name: `<col>`	Length is [1–255] characters (Amazon DocumentDB 5.0 and later).

Resource	Default quota
	Length is [1–57] characters (Amazon DocumentDB 4.0 and earlier).
Database name: <db>	Length is [1–63] characters.
Fully qualified collection name: <db>.<col>	Length is [3–255] characters (Amazon DocumentDB 5.0 and later). Length is [3–120] characters (Amazon DocumentDB 4.0 and earlier).
Fully qualified index name: <db>.<col>$<index>	Length is [6–511] characters (Amazon DocumentDB 5.0 and later). Length is [6–127] characters (Amazon DocumentDB 4.0 and earlier).
Index name	Length is [1–255] characters (Amazon DocumentDB 5.0 and later). Length is [3–63] characters for the combined <col>$<index> (Amazon DocumentDB 4.0 and earlier).

Resource	Default quota

Primary password	• Length is [8-100] printable ASCII characters.
	• Can use any printable ASCII characters except for the following:
	  • **/** (forward slash)
	  • **"** (double quotation mark)
	  • **@** (at symbol)

Primary user name	• Length is [1-63] alphanumeric characters. • First character must be a letter. • Cannot be a word reserved by the database engine.
Parameter group name	• Length is [1–255] alphanumeric characters. • First character must be a letter. • Cannot end with a hyphen or contain two consecutive hyphens.

TTL constraints

Deletes from a TTL index are not guaranteed within a specific timeframe and are best effort. Factors like instance resource utilization, document size, and overall throughput can affect the timing of a TTL delete.

Elastic cluster quotas

The following table describes maximum quotas for Amazon DocumentDB elastic clusters.

Resource	Quota
Elastic clusters per Region	20
vCPU summed across all elastic clusters per Region	1024
Manual cluster snapshots per Region	20
Shards per cluster	32
Storage per cluster (when data is evenly distributed by shard-key)	4 PiB
Connections to cluster	The lower value of either 300,000 or the number of shards x the connection quota associated with vCPU per shard
UnSharded collection size	32 TiB
Sharded collection size (when data is evenly distributed by shard-key)	1PB
Databases per cluster	10,000
UnSharded collections per cluster	100,000
Sharded collections per cluster	1000
Users per cluster	100
Writes in a single batch command	100,000
Indexes per collection	64
Document nesting depth	100 levels
Document size	16MB
Index key size	2048 bytes

Resource	Quota
Keys in a compound index	32

Elastic cluster shard quotas

The following table describes maximum shard quotas for Amazon DocumentDB elastic clusters.

Resource	Quota
vCPU per shard instance	64
Instances per shard	16
Storage per shard	128 TiB
Storage per collection per shard	32 TiB

Elastic cluster CPU, memory, connection, and cursor quotas per shard

The following table describes maximum CPU, memory, connection, and cursor limits in Amazon DocumentDB elastic cluster shards.

vCPUs per shard	Instance memory (GiB)	Connection quota	Cursor quota
2	16	1700	450
4	32	3500	450
8	64	7100	450
16	128	14200	760
32	256	28400	1520

vCPUs per shard	Instance memory (GiB)	Connection quota	Cursor quota
48	384	30000	2280
64	512	30000	3040

Troubleshooting Amazon DocumentDB

The following sections provide information about how to troubleshoot problems that you might encounter when using Amazon DocumentDB (with MongoDB compatibility).

Topics

- Troubleshooting connectivity issues
- Troubleshooting indexes
- Troubleshooting performance and resource utilization
- Garbage collection in Amazon DocumentDB
- Error codes

Troubleshooting connectivity issues

Having trouble connecting? Here are some common scenarios and how to resolve them.

Topics

- Can't connect to an Amazon DocumentDB endpoint
- Testing a connection to an Amazon DocumentDB instance
- Connecting to an invalid endpoint
- Driver configuration impacting number of connections

Can't connect to an Amazon DocumentDB endpoint

When you try to connect to Amazon DocumentDB, the following is one of the most common error messages that you might receive.

```
connecting to: mongodb://docdb-2018-11-08-21-47-27.cluster-ccuszbx3pn5e.us-east-
1.docdb.amazonaws.com:27017/
2018-11-14T14:33:46.451-0800 W NETWORK [thread1] Failed to connect to
172.31.91.193:27017 after 5000ms milliseconds, giving up.
2018-11-14T14:33:46.452-0800 E QUERY [thread1] Error: couldn't connect to server
docdb-2018-11-08-21-47-27.cluster-ccuszbx3pn5e.us-east-1.docdb.amazonaws.com:27017,
connection attempt failed :
```

```
connect@src/mongo/shell/mongo.js:237:13
@(connect):1:6
exception: connect failed
```

What this error message typically means is that your client (the mongo shell in this example) cannot access the Amazon DocumentDB endpoint. This might be the case for several reasons:

Topics

- Connecting from public endpoints
- Cross-Region connections
- Connecting from different Amazon VPCs
- Security group blocks inbound connections
- Java Mongo driver read preference issue

Connecting from public endpoints

You are trying to connect to an Amazon DocumentDB cluster directly from your laptop or local development machine.

Trying to connect to an Amazon DocumentDB cluster directly from a public endpoint, such as your laptop or local development machine, will fail. Amazon DocumentDB is virtual private cloud (VPC)-only and does not currently support public endpoints. Thus, you can't connect directly to your Amazon DocumentDB cluster from your laptop or local development environment outside of your VPC.

To connect to an Amazon DocumentDB cluster from outside an Amazon VPC, you can use an SSH tunnel. For more information, see Connecting to an Amazon DocumentDB cluster from outside an Amazon VPC. Additionally, if your development environment is in a different Amazon VPC, you can also use VPC Peering and connect to your Amazon DocumentDB cluster from another Amazon VPC in the same Region or a different Region.

Cross-Region connections

You are trying to connect to an Amazon DocumentDB cluster in another Region.

If you try to connect to an Amazon DocumentDB cluster from an Amazon EC2 instance in a Region other than the cluster's Region—for example, trying to connect to a cluster in US East (N. Virginia) Region (us-east-1) from US West (Oregon) Region (us-west-2)—the connection will fail.

To verify the Region of your Amazon DocumentDB cluster, run the following command. The Region is in the endpoint.

```
aws docdb describe-db-clusters \
   --db-cluster-identifier sample-cluster \
   --query 'DBClusters[*].Endpoint'
```

Output from this operation looks something like the following.

```
[
    "sample-cluster.node.us-east-1.docdb.amazonaws.com"
]
```

To verify the Region of your EC2 instance, run the following command.

```
 aws ec2 describe-instances \
     --query 'Reservations[*].Instances[*].Placement.AvailabilityZone'
```

Output from this operation looks something like the following.

```
[
    [
        "us-east-1a"
    ]
]
```

Connecting from different Amazon VPCs

You are trying to connect to an Amazon DocumentDB cluster from a VPC that is different than the Amazon VPC your cluster is deployed to.

If both your Amazon DocumentDB cluster and Amazon EC2 instance are in the same AWS Region, but not in the same Amazon VPC, you cannot connect directly to your Amazon DocumentDB cluster unless VPC Peering is enabled between the two Amazon VPCs.

To verify the Amazon VPC of your Amazon DocumentDB instance, run the following command.

```
aws docdb describe-db-instances \
   --db-instance-identifier sample-instance \
   --query 'DBInstances[*].DBSubnetGroup.VpcId'
```

To verify the Amazon VPC of your Amazon EC2 instance, run the following command.

```
aws ec2 describe-instances \
   --query 'Reservations[*].Instances[*].VpcId'
```

Security group blocks inbound connections

You are trying to connect to an Amazon DocumentDB cluster, and the cluster's security group does not allow inbound connections on the cluster's port (default port: 27017).

Suppose that your Amazon DocumentDB cluster and Amazon EC2 instance are both in the same Region and Amazon VPC and use the same Amazon VPC security group. If you can't connect to your Amazon DocumentDB cluster, the likely cause is that your security group (that is, firewall) for your cluster doesn't allow inbound connections on the port you chose for your Amazon DocumentDB cluster (default port is 27017).

To verify the port for your Amazon DocumentDB cluster, run the following command.

```
aws docdb describe-db-clusters \
   --db-cluster-identifier sample-cluster \
   --query 'DBClusters[*].[DBClusterIdentifier,Port]'
```

To get your Amazon DocumentDB security group for your cluster, run the following command.

```
aws docdb describe-db-clusters \
   --db-cluster-identifier sample-cluster \
   --query 'DBClusters[*].[VpcSecurityGroups[*],VpcSecurityGroupId]'
```

To check the inbound rules for your security group, see the following topics in the Amazon EC2 documentation:

- Authorizing Inbound Traffic for Your Linux Instances
- Authorizing Inbound Traffic for Your Windows Instances

Java Mongo driver read preference issue

Client read preferences are not honored and some clients cannot write to Amazon DocumentDB after failover unless they reboot.

This issue, first discovered in Java Mongo Driver 3.7.x, occurs when a client establishes a connection to Amazon DocumentDB using `MongoClientSettings` and, specifically, when chaining the `applyToClusterSettings` method. The MongoClient Cluster Settings can be defined using a few different methods, such as `hosts()`, `requiredReplicaSetName()`, and `mode()`.

When the client specifies only one host in the `hosts()` method, the mode is set to `ClusterConnectionMode.SINGLE` instead of `ClusterConnectionMode.MULTIPLE` This causes the client to disregard the read preference and only connect to the server configured in `hosts()`. So even if the client settings are initialized like below, all reads would still go to the primary instead of the secondary.

```
final ServerAddress serverAddress0 = new ServerAddress("cluster-endpoint", 27317));
    final MongoCredential credential = MongoCredential.createCredential("xxx",
            "admin", "xxxx".toCharArray());
    final MongoClientSettings settings = MongoClientSettings.builder()
            .credential(credential)
            .readPreference(ReadPreference.secondaryPreferred())
            .retryWrites(false)
            .applyToSslSettings(builder -> builder
                    .enabled(false))
            .applyToClusterSettings(builder -> builder.hosts(
                            Arrays.asList(serverAddress0
                            ))
                    .requiredReplicaSetName("rs0"))
            .build();
    MongoClient mongoClient = MongoClients.create(settings);
```

Failover case

Using the above client connection settings, if there is a failover and a delayed DNS record update for the cluster writer endpoint, the client would still try to issue writes to the old writer (now reader after failover). This results in a server-side error (not primary) which is not handled appropriately by the Java driver (this is still under investigation). Thus, the client can be left in a bad state until the application server is rebooted, for example.

There are two workarounds for this:

- Clients that connect to Amazon DocumentDB via a connection string will not have this issue, since `ClusterConnectionMode` will be set to `MULTIPLE` when setting read preference.

```
MongoClientURI mongoClientURI = new MongoClientURI("mongodb://usr:pass:cluster-
endpoint:27317/test?ssl=false&replicaSet=rs0&readpreference=secondaryPreferred");
MongoClient mongoClient = MongoClients.create(mongoClientURI.getURI());
```

Or using `MongoClientSettings` builder with the `applyConnectionString` method.

```
final MongoClientSettings settings = MongoClientSettings.builder()
        .credential(credential)
        .applyConnectionString(new ConnectionString("usr:pass:cluster-endpoint:27317/
test?ssl=false&replicaSet=rs0&readpreference=secondaryPreferred"))
        .retryWrites(false)
        .applyToSslSettings(builder # builder
                .enabled(false))
        .build();
MongoClient mongoClient = MongoClients.create(settings);
```

- Explicitly set `ClusterConnectionMode` to `MULTIPLE`. This is only needed when using `applyToClusterSettings` and `hosts().size() == 1`.

```
final ServerAddress serverAddress0 = new ServerAddress("cluster-endpoint", 27317));
final MongoCredential credential = MongoCredential.createCredential("xxx","admin",
 "xxxx".toCharArray());
final MongoClientSettings settings = MongoClientSettings.builder()
    .credential(credential)
    .readPreference(ReadPreference.secondaryPreferred())
    .retryWrites(false)
    .applyToSslSettings(builder # builder
    .enabled(false))
    .applyToClusterSettings(builder # builder
                .hosts(Arrays.asList(serverAddress0))
                .requiredReplicaSetName("rs0"))
                .mode(ClusterConnectionMode.MULTIPLE))
    .build();
MongoClient mongoClient = MongoClients.create(settings);
```

Testing a connection to an Amazon DocumentDB instance

You can test your connection to a cluster using common Linux or Windows tools.

From a Linux or Unix terminal, test the connection by entering the following (replace `cluster-endpoint` with the endpoint, and replace `port` with the port of your instance):

```
nc -zv cluster-endpoint port
```

The following is an example of a sample operation and the return value:

```
nc -zv docdbTest.d4c7nm7stsfc0.us-west-2.docdb.amazonaws.com 27017

Connection to docdbTest.d4c7nm7stsfc0.us-west-2.docdb.amazonaws.com 27017 port [tcp/*]
 succeeded!
```

Connecting to an invalid endpoint

When connecting to an Amazon DocumentDB cluster and you use a cluster endpoint that is not valid, an error similar to the following appears.

```
mongo --ssl \
   --host sample-cluster.node.us-east-1.docdb.amazonaws.com:27017 \
   --sslCAFile global-bundle.pem \
   --username <user-name> \
   --password <password>
```

The output looks like this:

```
MongoDB shell version v3.6
connecting to: mongodb://sample-cluster.node.us-east-1.docdb.amazonaws.com:27017/
2018-11-14T17:21:18.516-0800 I NETWORK [thread1] getaddrinfo("sample-cluster.node.us-
east-1.docdb.amazonaws.com") failed:
nodename nor servname provided, or not known 2018-11-14T17:21:18.537-0800 E QUERY
 [thread1] Error: couldn't initialize
connection to host sample-cluster.node.us-east-1.docdb.amazonaws.com, address is
 invalid :
connect@src/mongo/shell/mongo.js:237:13@(connect):1:6
exception: connect failed
```

To get the valid endpoint for a cluster, run the following command:

```
aws docdb describe-db-clusters \
   --db-cluster-identifier sample-cluster \
```

```
    --query 'DBClusters[*].[Endpoint,Port]'
```

To get the valid endpoint for an instance, run the following command:

```
aws docdb describe-db-instances \
    --db-instance-identifier sample-instance \
    --query 'DBInstances[*].[Endpoint.Address,Endpoint.Port]'
```

For more information, see Understanding Amazon DocumentDB endpoints.

Driver configuration impacting number of connections

When using the client driver to connect to an Amazon DocumentDB cluster, it's important to consider the `maxPoolSize` configuration parameter. The `maxPoolSize` setting determines the maximum number of connections that the client driver will maintain in its connection pool.

Troubleshooting indexes

The following topics address what to do if your index or background index build fails.

Topics

- Index build fails
- Background index build latency issues and fails
- Database index bloat

Index build fails

Amazon DocumentDB utilizes local storage on an instance as part of the index creation process. You can monitor this disk usage using the **FreeLocalStorage** CloudWatch metric (`CloudWatch -> Metrics -> DocDB -> Instance Metrics`). When an index build consumes all of the local disk and fails, you will receive an error. When migrating data to Amazon DocumentDB, we encourage you to create indexes first and then insert the data. For more information on migration strategies and creating indexes, see Migrating and upgrading Amazon DocumentDB in the Amazon DocumentDB documentation and the blog: Migrate from MongoDB to Amazon DocumentDB using the offline method.

When creating indexes on an existing cluster, if the index build is taking longer than expected or is failing, scale up the instance to create the index, then scale back down after the index is

created. Amazon DocumentDB enables you to quickly scale instance sizes in minutes using the AWS Management Console or the AWS CLI. For more information, see Managing instance classes. With per-second instance pricing, you only pay for the resource you use up to the second.

Background index build latency issues and fails

Background index builds in Amazon DocumentDB do not start until all queries on the primary instance that started before the index build was initiated complete executing. If there is a long running query, background index builds will block until the query finishes and thus can take longer than expected to complete. This is true even if collections are empty.

Foreground index builds do not exhibit the same blocking behavior. Instead, foreground index builds take an exclusive lock on the collection until the index build is completed. To create indexes on an empty collection and avoid blocking long-running queries, use foreground index builds.

> **Note**
>
> Amazon DocumentDB allows only one background index build to occur on a collection at any given time. If DDL (Data Definition Language) operations such as `createIndex()` or`dropIndex()` occur on the same collection during a background index build, the background index build fails.

Database index bloat

Amazon DocumentDB uses Multi-Version Concurrency Control (MVCC) to manage concurrent transactions. When documents are deleted or updated, their previous versions remain in collections and indexes as "dead" versions. The garbage collection process automatically reclaims space from these dead versions for future operations.

Index bloat occurs when a collection's indexes become larger due to the accumulation of dead or obsolete index entries or fragmentation within the pages. The percentage reported represents the amount of index space that can be used by future index entries. This bloat consumes space in both the buffer cache and storage. If you want to remove the bloat, you will need to rebuild indexes.

Example

Run the following command to determine unused storage for your index:

```
db.coll.aggregate({$indexStats:{}});
```

The result looks similar to this:

```
{
    "name" : "_id_",
    "key" : {
        "_id" : 1
    },
    "host" : "devbox-test.localhost.a2z.com:27317",
    "size" : NumberLong(827392),
    "accesses" : {
        "ops" : NumberLong(40000),
        "docsRead" : NumberLong(46049),
        "since" : ISODate("2025-04-03T21:44:51.251Z")
    },
    "cacheStats" : {
        "blksRead" : NumberLong(264),
        "blksHit" : NumberLong(140190),
        "hitRatio" : 99.8121
    },
    "unusedStorageSize" : {
        "unusedSizeBytes" : 409600,
        "unusedSizePercent" : 49.51
    }
}
```

You can rebuild indexes without downtime using the `reIndex` command, which requires a scan of the entire collection. See Index maintenance using `reIndex`.

Troubleshooting performance and resource utilization

This section provides questions and solutions for common diagnostics issues in Amazon DocumentDB deployments. The examples provided use the *mongo shell* and are scoped to an individual instance. To find an instance endpoint, see Understanding Amazon DocumentDB endpoints.

Topics

- How do I determine the number of insert, update, and delete operations performed on my collection through the Mongo API?

- How do I analyze cache performance?
- How do I find and terminate long running or blocked queries?
- How can I see a query plan and optimize a query?
- How can I see a query plan in elastic clusters?
- How do I list all running operations on an instance?
- How do I know when a query is making progress?
- How do I determine why a system suddenly runs slowly?
- How do I determine the cause of high CPU utilization on one or more cluster instances?
- How do I determine the open cursors on an instance?
- How do I determine the current Amazon DocumentDB engine version?
- How do I analyze index usage and identify unused indexes?
- How do I identify missing indexes?
- How do I determine database collection bloat?
- Summary of useful queries

How do I determine the number of insert, update, and delete operations performed on my collection through the Mongo API?

To view the number of insert, update, and delete operations performed on a certain collection, run the following command on that collection:

```
db.collection.stats()
```

The output from this command describes the following under its `opCounters` field:

- **numDocsIns** - The number of documents inserted into this collection. This includes documents inserted using the `insert` and `insertMany` commands, as well as documents inserted by an upsert.
- **numDocsUpd** - The number of documents updates in this collection. This includes documents updated using the `update` and `findAndModify` commands.
- **numDocsDel** - The number of documents deleted from this collection. This includes documents deleted using the `deleteOne`, `deleteMany`, `remove`, and `findAndModify` commands.

- **lastReset** - The time these counters have been last reset. The statistics provided by this command are reset when starting/stopping the cluster or scaling up/down the instance.

An example output from running `db.collection.stats()` is shown below.

```
{
    "ns" : "db.test",
    "count" : ...,
    "size" : ...,
    "avgObjSize" : ...,
    "storageSize" : ...,
    "capped" : false,
    "nindexes" : ...,
    "totalIndexSize" : ...,
    "indexSizes" : {
        "_id_" : ...,
        "x_1" : ...
    },
    "collScans" : ...,
    "idxScans" : ...,
    "opCounter" : {
        "numDocsIns" : ...,
        "numDocsUpd" : ...,
        "numDocsDel" : ...
    },
    "cacheStats" : {
        "collBlksHit" : ...,
        "collBlksRead" : ..,
        "collHitRatio" : ...,
        "idxBlksHit" : ...,
        "idxBlksRead" : ...,
        "idxHitRatio" : ...
    },
    "lastReset" : "2022-09-02 19:41:40.471473+00",
    "ok" : 1,
    "operationTime" : Timestamp(1662159707, 1)
}
```

This stats command should be used when viewing collection-specific counters for insert, update, and delete operation through the Mongo API. Another way to view collection-specific operation counters is by enabling DML auditing. The number of insert, update, and delete operations over all

collections during one minute time intervals can be viewed in Monitoring Amazon DocumentDB with CloudWatch.

How do I analyze cache performance?

Analyzing cache performance can give insights into the efficiency of data retrieval and system performance, and is based on how much data is read from the disk versus the cache. We provide cache statistics about the number of cache hits (data read from the cache) and cache misses (data that is not found in the cache and read from the disk) in order give insight into the cache performance. The cache statistics for a specific collection can by found by running the following command on that collection:

```
db.collection.stats()
```

The values in the `cacheStats` field in the output of this command provide cache statistics for the collection as well as the total cache statistics for the indexes created on the collection. These statistics are listed below:

- **`collBlksHit`** - The number of blocks read from the cache during operations on this collection.
- **`collBlksRead`** - The number of blocks read from the disk (cache misses) during operations on this collection.
- **`collHitRatio`** - The cache hit ratio for this collection (`100 * [collBlksHit / (collBlksHit + collBlksRead)]`).
- **`idxBlksHit`** - The number of blocks read from the cache for any index created on this collection.
- **`idxBlksRead`** - The number of blocks read from the disk (cache misses) for any index created on this collection.
- **`idxHitRatio`** - The cache hit ratio for the indexes created on this collection (`100 * [idxBlksHit / (idxBlksHit + idxBlksRead)]`).
- **`lastReset`** - The time these statistics have been last reset. The statistics provided by `db.collection.stats()` are reset when starting/stopping the cluster or scaling up/down the instance.

A breakdown of the `idxBlksHit` and `idxBlksRead` fields for each index can also be found using the `indexStats` command. Index specific cache statistics can be found by running the following command:

```
db.collection.aggregate([{$indexStats:{}}]).pretty()
```

For each index, the following cache statistics can be found under the `cacheStats` field:

- **`blksHit`** - The number of blocks read from cache for this index.
- **`blksRead`** - The number of blocks read from the disk for this index.
- **`blksHitRatio`** - The cache hit ratio rounded to four decimal places, calculated by `100 * [blksHit / (blksHit + blksRead)]`.

How do I find and terminate long running or blocked queries?

User queries can run slowly because of a suboptimal query plan or can be blocked due to resource contention.

To find long running queries that slow down due to a suboptimal query plan, or queries that are blocked due to resource contention, use the `currentOp` command. You can filter the command to help narrow down the list of relevant queries to terminate. You must have `opid` associated with the long running query to be able to terminate the query.

The following query uses the `currentOp` command to list all queries that are either blocked or running for more than 10 seconds.

```
db.adminCommand({
    aggregate: 1,
    pipeline: [
        {$currentOp: {}},
        {$match:
            {$or: [
                {secs_running: {$gt: 10}},
                {WaitState: {$exists: true}}]}},
        {$project: {_id:0, opid: 1, secs_running: 1}}],
    cursor: {}
});
```

Next, you can narrow down the query to find the `opid` of a query running for more than 10 seconds and terminate it.

To find and terminate a query running for more than 10 seconds

1. Find the opid of the query.

```
db.adminCommand({
    aggregate: 1,
    pipeline: [
        {$currentOp: {}},
        {$match:
            {$or:
                [{secs_running: {$gt: 10}},
                 {WaitState: {$exists: true}}]}}],
    cursor: {}
});
```

   Output from this operation looks something like the following (JSON format).

```
{
    "waitedMS" : NumberLong(0),
    "cursor" : {
        "firstBatch" : [
            {
                "opid" : 24646,
                "secs_running" : 12
            }
        ],
        "id" : NumberLong(0),
        "ns" : "admin.$cmd"
    },
    "ok" : 1
}
```

2. Terminate the query using the killOp operation.

```
db.adminCommand({killOp: 1, op: 24646});
```

How can I see a query plan and optimize a query?

If a query runs slow, it could be because the query execution requires a full scan of the collection to choose the relevant documents. Sometimes creating appropriate indexes enables the query

to run faster. To detect this scenario and decide the fields on which to create the indexes, use the `explain` command.

> **Note**
>
> Amazon DocumentDB emulates the MongoDB 3.6 API on a purpose-built database engine that utilizes a distributed, fault-tolerant, self-healing storage system. As a result, query plans and the output of `explain()` may differ between Amazon DocumentDB and MongoDB. Customers who want control over their query plan can use the `$hint` operator to enforce selection of a preferred index.

Run the query that you want to improve under the `explain` command as follows.

```
db.runCommand({explain: {<query document>}})
```

The following is an example operation.

```
db.runCommand({explain:{
    aggregate: "sample-document",
    pipeline: [{$match: {x: {$eq: 1}}}],
    cursor: {batchSize: 1}}
});
```

Output from this operation looks something like the following (JSON format).

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "db.test",
        "winningPlan" : {
            "stage" : "COLLSCAN"
        }
    },
    "serverInfo" : {
        "host" : "...",
        "port" : ...,
        "version" : "..."
    },
    "ok" : 1
```

```
}
```

The preceding output indicates that the `$match` stage requires scanning the whole collection and checking if the field `"x"` in each document is equal to 1. If there are many documents in the collection, the collection scan (and therefore the overall query performance) is very slow. Thus the presence of the `"COLLSCAN"` in the output of the `explain` command indicates that the query performance can be improved by creating appropriate indexes.

In this example, the query checks whether the field `"x"` equals 1 in all documents. So creating an index on field `"x"` enables the query to avoid the complete collection scan and use the index to return the relevant documents sooner.

After creating an index on field `"x"`, the `explain` output is as follows.

```
{
    "queryPlanner" : {
         "plannerVersion" : 1,
         "namespace" : "db.test",
         "winningPlan" : {
             "stage" : "IXSCAN",
             "indexName" : "x_1",
             "direction" : "forward"
         }
    },
    "serverInfo" : {
        "host" : "...",
        "port" : ...,
        "version" : "..."
    },
    "ok" : 1
}
```

Creating an index on field `"x"` enables the `$match` stage to use an index scan to reduce the number of documents on which the predicate `"x = 1"` must be evaluated.

For small collections, the Amazon DocumentDB query processor can choose not to use an index if the performance gains are negligible.

How can I see a query plan in elastic clusters?

To examine a query plan in elastic clusters, use the `explain` command. The following is an example `explain` operation on a find query targeting a sharded collection:

```
db.runCommand(
   {
     explain: { find: "cities", filter: {"name": "Seoul"}}
   }
)
```

> **Note**
>
> Amazon DocumentDB emulates MongoDB on a purpose-built database engine. As a result, query plans and the output of `explain()` may differ between Amazon DocumentDB and MongoDB. You can control query plan with the use of the `$hint` operator to enforce selection of a preferred index.

Output from this operation may look something like the following (JSON format):

```
{
  "queryPlanner" : {
    "elasticPlannerVersion" : 1,
    "winningPlan" : {
      "stage" : "SINGLE_SHARD",
      "shards" : [
        {
          "plannerVersion" : 1,
          "namespace" : "population.cities",
          "winningPlan" : {
            "stage" : "SHARD_MERGE",
            "shards" : [
              {
                "shardName" : "f2cf5cfd-fe9c-40ca-b4e5-298ca0d11111",
                "plannerVersion" : 1,
                "namespace" : "population.cities",
                "winningPlan" : {
                  "stage" : "PARTITION_MERGE",
                  "inputStages" : [
                    {
```

```
                    "stage" : "COLLSCAN",
                    "partitionCount" : 21
                  }
                ]
              }
            },
            {
              "shardName" : "8f3f80e2-f96c-446e-8e9d-aab8c7f22222",
              "plannerVersion" : 1,
              "namespace" : "population.cities",
              "winningPlan" : {
                "stage" : "PARTITION_MERGE",
                "inputStages" : [
                  {
                    "stage" : "COLLSCAN",
                    "partitionCount" : 21
                  }
                ]
              }
            },
            {
              "shardName" : "32c5a06f-1b2b-4af1-8849-d7c4a033333",
              "plannerVersion" : 1,
              "namespace" : "population.cities",
              "winningPlan" : {
                "stage" : "PARTITION_MERGE",
                "inputStages" : [
                  {
                    "stage" : "COLLSCAN",
                    "partitionCount" : 22
                  }
                ]
              }
            }
          ]
        },
        "shardName" : "32c5a06f-1b2b-4af1-8849-d7c4a0f3fb58"
      }
    ]
  }
},
"serverInfo" : {
  "host" : "example-4788267630.us-east-1.docdb-elastic.amazonaws.com:27017",
  "version" : "5.0.0"
```

```
  },
  "ok" : 1,
  "operationTime" : Timestamp(1695097923, 1)
}
```

The preceding output shows the query plan for the `find` query on a three-shard cluster. Each shard has multiple data partitions which can have different input stages. In this example, a “COLLSCAN“ (a collection scan) is run on all partitions before the results are merged at the ”PARTITION_MERGE“ stage within each shard. The results across the shards are then merged together at the ”SHARD_MERGE“ stage before being sent back to the client.

How do I list all running operations on an instance?

As a user or primary user, you often want to list all the current operations running on an instance for diagnostics and troubleshooting purposes. (For information about managing users, see Managing Amazon DocumentDB users.)

With the `mongo` shell, you can use the following query to list all the running operations on an Amazon DocumentDB instance.

```
db.adminCommand({currentOp: 1, $all: 1});
```

The query returns the complete list of all user queries and internal system tasks currently operating on the instance.

Output from this operation looks something like the following (JSON format).

```
{
    "inprog" : [
        {
            "desc" : "INTERNAL"
        },
        {
            "desc" : "TTLMonitor",
            "active" : false
        },
        {
            "client" : ...,
            "desc" : "Conn",
            "active" : true,
            "killPending" : false,
            "opid" : 195,
```

```
            "ns" : "admin.$cmd",
            "command" : {
                "currentOp" : 1,
                "$all" : 1
            },
            "op" : "command",
            "$db" : "admin",
            "secs_running" : 0,
            "microsecs_running" : NumberLong(68),
            "clientMetaData" : {
            "application" : {
                "name" : "MongoDB Shell"
            },
            "driver" : {
                ...
            },
            "os" : {
                ...
            }
         }
      },
      {
         "desc": "GARBAGE_COLLECTION",
         "garbageCollection": {
            "databaseName": "testdb",
            "collectionName": "testCollectionA"
         },
         "secs_running": 3,
         "microsecs_running": NumberLong(3123456)
      },
      {
         "desc": "GARBAGE_COLLECTION",
         "garbageCollection": {
            "databaseName": "testdb",
            "collectionName": "testCollectionB"
         },
         "secs_running": 4,
         "microsecs_running": NumberLong(4123456)
      }
   ],
   "ok" : 1
}
```

The following are valid values for the `"desc"` field:

- **`INTERNAL`** — Internal system tasks like the cursor cleanup or stale user cleanup tasks.
- **`TTLMonitor`** — The Time to Live (TTL) monitor thread. Its running status is reflected in the `"active"` field.
- **`GARBAGE_COLLECTION`** — The internal garbage collector thread.
- **`CONN`** — The user query.
- **`CURSOR`** — The operation is an idle cursor waiting on the user to call the "getMore" command to get the next batch of results. In this state, the cursor is consuming memory, but is not consuming any compute.

The preceding output also lists all user queries running in the system. Each user query runs in the context of a database and collection, and the union of these two is called a *namespace*. The namespace of each user query is available in the `"ns"` field.

Sometimes you need to list all user queries that are running in a particular namespace. So the previous output must be filtered on the `"ns"` field. The following is an example query to achieve the output to filter. The query lists all user queries that are currently running in the database `"db"` and collection `"test"` (that is, the `"db.test"` namespace).

```
db.adminCommand({aggregate: 1,
    pipeline: [{$currentOp: {allUsers: true, idleConnections: true}},
               {$match: {ns: {$eq: "db.test"}}}],
    cursor: {}
});
```

As the primary user of the system, you can see queries of all users and also all internal system tasks. All other users can see only their respective queries.

If the total number of queries and internal system tasks exceeds the default batch cursor size, the mongo shell automatically generates an iterator object `'it'` to view the rest of the results. Keep executing the `'it'` command until all results have been exhausted.

How do I know when a query is making progress?

User queries can run slowly due to a suboptimal query plan, or they can be blocked due to resource contention. Debugging such queries is a multi-step process that can require executing the same step multiple times.

The first step of debugging is to list all queries that are long running or blocked. The following query lists all user queries that have been running for more than 10 seconds or that are waiting for resources.

```
db.adminCommand({aggregate: 1,
                 pipeline: [{$currentOp: {}},
                            {$match: {$or: [{secs_running: {$gt: 10}},
                                            {WaitState: {$exists: true}}]}},
                            {$project: {_id:0,
                                        opid: 1,
                                        secs_running: 1,
                                        WaitState: 1,
                                        blockedOn: 1,
                                        command: 1}}],
                 cursor: {}
                });
```

Repeat the preceding query periodically to determine whether the list of queries changes and to identify the long running or blocked queries.

If the output document for the query of interest has a `WaitState` field, it indicates that resource contention is why the query is running slow or is blocked. The resource contention could either be due to I/O, internal system tasks, or other user queries.

Output from this operation looks something like the following (JSON format).

```
{
    "waitedMS" : NumberLong(0),
    "cursor" : {
        "firstBatch" : [
            {
                "opid" : 201,
                "command" : {
                    "aggregate" : ...
                },
                "secs_running" : 208,
                "WaitState" : "IO"
            }
        ],
        "id" : NumberLong(0),
        "ns" : "admin.$cmd"
    },
```

```
    "ok" : 1
}
```

I/O can be a bottleneck if many queries across different collections are running concurrently on the same instance, or if the instance size is too small for the dataset that the query is running on. If the queries are read-only queries, you can mitigate the former situation by separating the queries for each collection across separate replicas. For concurrent updates across different collections, or when the instance size is too small for the dataset, you can mitigate by scaling up the instance.

If the resource contention is due to other user queries, the `"blockedOn"` field in the output document will have the `"opid"` of the query that is affecting this query. Using the `"opid"` follows the chain of `"WaitState"` and `"blockedOn"` fields of all the queries to find the query at the head of the chain.

If the task at the head of the chain is an internal task, the only mitigation in this case would be to terminate the query and rerun it later.

The following is sample output in which the find query is blocked on a collection lock that is owned by another task.

```
{
    "inprog" : [
        {
            "client" : "...",
            "desc" : "Conn",
            "active" : true,
            "killPending" : false,
            "opid" : 75,
            "ns" : "...",
            "command" : {
                "find" : "...",
                "filter" : {

                }
            },
            "op" : "query",
            "$db" : "test",
            "secs_running" : 9,
            "microsecs_running" : NumberLong(9449440),
            "threadId" : 24773,
            "clientMetaData" : {
```

```
                "application" : {
                   "name" : "MongoDB Shell"
                },
                "driver" : {
                    ...
                },
                "os" : {
                    ...
                }
            },
            "WaitState" : "CollectionLock",
            "blockedOn" : "INTERNAL"
        },
        {
            "desc" : "INTERNAL"
        },
        {
            "client" : "...",
            ...
            "command" : {
                "currentOp" : 1
            },
            ...
        }
    ],
    "ok" : 1
}
```

If the "WaitState" has values "Latch", "SystemLock", "BufferLock", "BackgroundActivity", or "Other", the source of resource contention is internal system tasks. If the situation continues for a long time, the only mitigation would be to terminate the query and rerun it later.

How do I determine why a system suddenly runs slowly?

The following are some common reasons for a system slowing down:

- Excessive resource contention between concurrent queries
- The number of active concurrent queries increasing over time
- Internal system tasks such as "GARBAGE_COLLECTION"

To monitor the system usage over time, run the following "currentOp" query periodically and output the results to an external store. The query counts the number of queries and operations in each namespace in the system. You can then analyze the system usage results to understand the load on the system and take appropriate action.

```
db.adminCommand({aggregate: 1,
                 pipeline: [{$currentOp: {allUsers: true, idleConnections: true}},
                            {$group: {_id: {desc: "$desc", ns: "$ns", WaitState:
 "$WaitState"}, count: {$sum: 1}}}],
                 cursor: {}
                });
```

This query returns an aggregate of all queries running in each namespace, all the internal system tasks, and the unique number of wait states (if any) per namespace.

Output from this operation looks something like the following (JSON format).

```
{
    "waitedMS" : NumberLong(0),
    "cursor" : {
        "firstBatch" : [
            {
                "_id" : {
                    "desc" : "Conn",
                    "ns" : "db.test",
                    "WaitState" : "CollectionLock"
                },
               "count" : 2
            },
            {
                "_id" : {
                    "desc" : "Conn",
                    "ns" : "admin.$cmd"
                },
                "count" : 1
            },
            {
                "_id" : {
                    "desc" : "TTLMonitor"
                },
                "count" : 1
            }
```

```
        ],
        "id" : NumberLong(0),
        "ns" : "admin.$cmd"
    },
    "ok" : 1
}
```

In the preceding output, two user queries in namespace `"db.test"` are blocked on collection lock: one query in the namespace `"admin.$cmd"`, and one internal `"TTLMonitor"` task.

If the output indicates many queries with blocking wait states, see How do I find and terminate long running or blocked queries?

How do I determine the cause of high CPU utilization on one or more cluster instances?

The following sections might help you identify the cause of high instance CPU utilization. Your results can vary depending on the workload.

- To determine why an instance is suddenly running slowly, see How do I determine why a system suddenly runs slowly?
- To identify and terminate long running queries on a particular instance, see How do I find and terminate long running or blocked queries?
- To understand whether a query is progressing, see How do I know when a query is making progress?
- To determine why a query takes a long time to run, see How can I see a query plan and optimize a query?
- To track long-running queries over time, see Profiling Amazon DocumentDB operations.

Depending on the reason for your high instance CPU utilization, doing one or more of the following can help.

- If the primary instance exhibits high CPU utilization, but the replica instances don't, consider distributing read traffic across replicas via client read preference settings (for example, `secondaryPreferred`). For more information, see Connecting to Amazon DocumentDB as a replica set.

Using replicas for reads can make better use of the cluster's resources by allowing the primary instance to process more write traffic. Reads from replicas are eventually consistent.

- If the high CPU utilization is a result of your write workload, changing the size of the cluster's instances to a larger instance type increases the number of CPU cores available to service the workload. For more information, see Managing Amazon DocumentDB instances and Instance class specifications.
- If all cluster instances exhibit high CPU utilization, and the workload is using replicas for reads, adding more replicas to the cluster increases the resources available for read traffic. For more information, see Adding an Amazon DocumentDB instance to a cluster.

How do I determine the open cursors on an instance?

When connected to a Amazon DocumentDB instance, you can use the command `db.runCommand("listCursors")` to list the open cursors on that instance. There is a quota of up to 4,560 active cursors open at any given time on a given Amazon DocumentDB instance, depending on the instance type. Close cursors that are no longer in use because cursors utilize resources on an instance. For more information, see Amazon DocumentDB quotas.

```
db.runCommand("listCursors")
```

How do I determine the current Amazon DocumentDB engine version?

To determine your current Amazon DocumentDB engine version, run the following command.

```
db.runCommand({getEngineVersion: 1})
```

Output from this operation looks something like the following (JSON format).

```
{ "engineVersion" : "2.x.x", "ok" : 1 }
```

> **Note**
>
> The engine version for Amazon DocumentDB 3.6 is 1.x.x, the engine version for Amazon DocumentDB 4.0 is 2.x.x, the engine version for Amazon DocumentDB 5.0 is 3.x.x, and the engine version for Amazon DocumentDB 8.0 is 4.x.x.

How do I analyze index usage and identify unused indexes?

To identify the indexes for a given collection, run the following command:

```
db.collection.getIndexes()
```

To analyze how much indexes are being used during operations performed on the collections, the `collStats` and `indexStats` commands can be used. In order to view the total number of scans performed using indexes (index scans) compared to the number of scans performed without an index (collection scans), run the following command:

```
db.collection.stats()
```

The output for this command includes the following values:

- **`idxScans`** - The number of scans performed on this collection using an index.
- **`collScans`** - The number of scans performed on this collection without using an index. These scans would have involved looking over the documents in the collection one at a time.
- **`lastReset`** - The time these counters have been last reset. The statistics provided by this command are reset when starting/stopping the cluster or scaling up/down the instance.

A breakdown of how much each index is used can be found in the output of the following command. It is a best practice to regularly identify and remove unused indexes to improve performance and reduce cost, as it eliminates unnecessary compute, storage, and I/Os used to maintain the indexes.

```
db.collection.aggregate([{$indexStats:{}}]).pretty()
```

The output from this command gives the following values for each index created on the collection:

- **`ops`** - The number of operations that used the index. If your workload has been running for a sufficiently long time and you are confident that your workload is in a steady state, an `ops` value of zero would indicate that the index is not used at all.
- **`numDocsRead`** - The number of documents read during operations using this index.
- **`since`** - The time since Amazon DocumentDB started collecting stats on index usage, which is typically the value since the last database restart or maintenance action.
- **`size`** - The size of this index in bytes.

The following example is a sample output from running the above command:

```
{
    "name" : "_id_",
    "key" : {
        "_id" : 1
    },
    "host" : "example-host.com:12345",
    "size" : NumberLong(...),
    "accesses" : {
        "ops" : NumberLong(...),
        "docsRead" : NumberLong(...),
        "since" : ISODate("...")
    },
    "cacheStats" : {
        "blksRead" : NumberLong(...),
        "blksHit" : NumberLong(...),
        "hitRatio" : ...
    }
}
{
    "name" : "x_1",
    "key" : {
        "x" : 1
    },
    "host" : "example-host.com:12345",
    "size" : NumberLong(...),
    "accesses" : {
        "ops" : NumberLong(...),
        "docsRead" : NumberLong(...),
        "since" : ISODate("...")
    },
    "cacheStats" : {
        "blksRead" : NumberLong(...),
        "blksHit" : NumberLong(...),
        "hitRatio" : ...
    }
}
```

To determine the overall index size for a collection, run the following command:

```
db.collection.stats()
```

To drop an unused index, run the following command:

```
db.collection.dropIndex("indexName")
```

How do I identify missing indexes?

You can use the Amazon DocumentDB profiler to log slow queries. A query that appears repeatedly in the slow query log may indicate that an additional index is required to improve that query's performance.

You can identify opportunities for helpful indexes by looking for long running queries that have one or more stages that perform at least one `COLLSCAN` stage, meaning that the query stage has to read every document in the collection to provide a response to the query.

The following example shows a query on a collection of taxi rides that ran on a large collection.

```
db.rides.count({"fare.totalAmount":{$gt:10.0}}))
```

In order to execute this example, the query had to perform a collection scan (i.e. read every single document in the collection) since there is no index on the `fare.totalAmount` field. Output from the Amazon DocumentDB profiler for this query looks something like the following:

```
{
    ...
    "cursorExhausted": true,
    "nreturned": 0,
    "responseLength": 0,
    "protocol": "op_query",
    "millis": 300679,
    "planSummary": "COLLSCAN",
    "execStats": {
        "stage": "COLLSCAN",
        "nReturned": "0",
        "executionTimeMillisEstimate": "300678.042"
    },
    "client": "172.31.5.63:53878",
    "appName": "MongoDB Shell",
    "user": "example"
}
```

To speed up the query in this example, you want to create an index on `fare.totalAmount`, as shown below.

```
db.rides.createIndex( {"fare.totalAmount": 1}, {background: true} )
```

> **Note**
>
> Indexes created in the foreground (meaning if the `{background:true}` option was not supplied when creating the index) take an exclusive write lock, which prevents applications from writing data to the collection until the index build completes. Be aware of this potential impact when creating indexes on production clusters. When creating indexes, set `{background:true}`.

In general, you want to create indexes on fields that have high cardinality (for example, a large number of unique values). Creating an index on a field with low cardinality can result in a large index that is not used. The Amazon DocumentDB query optimizer considers the overall size of the collection and selectivity of the indexes when creating a query plan. There are times where you will see the query processor select a `COLLSCAN` even when an index is present. This happens when the query processor estimates that utilizing the index will not yield a performance advantage over scanning the entire collection. If you want to force the query processor to utilize a particular index, you can use the `hint()` operator as shown below.

```
db.collection.find().hint("indexName")
```

How do I determine database collection bloat?

Collection bloat occurs when a collection's size becomes larger due to the accumulation of dead or obsolete documents or fragmentation within database pages. The percentage reported represents the amount of document space that can be used by future documents. This bloat consumes space in both the buffer cache and storage. To remove the bloat, collections must be reloaded either through dump/restore or using a migration loop-back and switchover during a maintenance window.

Example

Run the following command to determine unused storage for your collection:

```
db.runCommand({collStats:'coll'})
```

The result looks similar to this:

```
{
        "ns" : "test.coll",
        "count" : 7500,
        "size" : 23250,
        "avgObjSize" : 31,
        "storageSize" : 106496,
        "unusedStorageSize" : {
                "unusedBytes" : 16384,
                "unusedPercent" : 25.12
        },
        "compression" : {
                "enable" : false
        },
        "capped" : false,
        "nindexes" : 1,
        "totalIndexSize" : 57344,
        "indexSizes" : {
                "_id_" : 57344
        },
        "collScans" : 4,
        "idxScans" : 10000,
        "opCounter" : {
                "numDocsIns" : 1000,
                "numDocsUpd" : 0,
                "numDocsDel" : 250
        },
        "cacheStats" : {
                "collBlksHit" : 3570,
                "collBlksRead" : 8,
                "collHitRatio" : 99.7765,
                "idxBlksHit" : 12293,
                "idxBlksRead" : 6,
                "idxHitRatio" : 99.9513
        },
        "lastReset" : "2024-12-18 00:30:21.552019+00",
        "ok" : 1,
        "operationTime" : Timestamp(1734632375, 1)
}
```

Summary of useful queries

The following queries can be useful for monitoring performance and resource utilization in Amazon DocumentDB.

- Use the following command to view statistics about a specific collection, including operation counters, cache statistics, accesses statistics, and size statistics:

```
db.collection.stats()
```

- Use the following command to view statistics about each index created on a collection including the size of the index, index-specific cache statistics, and index usage statistics:

```
db.collection.aggregate([{$indexStats:{}}]).pretty()
```

- Use the following query to list all activity.

```
db.adminCommand({currentOp: 1, $all: 1});
```

- The following code lists all long running or blocked queries.

```
db.adminCommand({aggregate: 1,
                 pipeline: [{$currentOp: {}},
                            {$match: {$or: [{secs_running: {$gt: 10}},
                                            {WaitState: {$exists: true}}]}},
                            {$project: {_id:0,
                                        opid: 1,
                                        secs_running: 1,
                                        WaitState: 1,
                                        blockedOn: 1,
                                        command: 1}}],
                 cursor: {}
                });
```

- The following code terminates a query.

```
db.adminCommand({killOp: 1, op: <opid of running or blocked query>});
```

- Use the following code to get an aggregated view of the system state.

```
db.adminCommand({aggregate: 1,
                 pipeline: [{$currentOp: {allUsers: true, idleConnections: true}},
```

```
                            {$group: {_id: {desc: "$desc", ns: "$ns", WaitState:
"$WaitState"}, count: {$sum: 1}}}],
                 cursor: {}
                });
```

Garbage collection in Amazon DocumentDB

Amazon DocumentDB implements a multi-version concurrency control (MVCC) database architecture that creates new versions of document and index entries for every update operation. This architecture provides enables transaction isolation, preventing one transaction's changes from appearing in another.

Topics

- Understanding Garbage collection in Amazon DocumentDB
- Garbage collection process
- Storage architecture and extended storage
- Monitoring garbage collection
- Example collStats output
- Frequently asked questions

Understanding Garbage collection in Amazon DocumentDB

Garbage collection (GC) is an automated background process that maintains optimal system performance and availability in Amazon DocumentDB. Like many modern databases, Amazon DocumentDB's MVCC architecture creates new document and index versions with each update. Each write operation consumes a unique MVCC ID from a finite counter. These IDs identify which transaction a document version belongs to and whether it has been committed or rolled back. Over time, these old versions and their MVCC IDs accumulate, requiring cleanup to prevent performance degradation.

Functions of garbage collection

Garbage collector serves three essential functions:

- **Reclaims storage space** — It removes obsolete document and index versions that are no longer needed by active queries, freeing space for future write operations.

- **Prevents MVCC ID overflow** — It prevents MVCC ID overflow by managing the finite counter of MVCC IDs. Without this management, the counter would eventually reach its limit, forcing the database into a temporary read-only mode until IDs are recycled.
- **Maintains query performance** — It maintains optimal query performance by eliminating dead document versions that would otherwise accumulate and slow down query processing.

Garbage collection process

The GC process operates per collection and can have multiple processes running concurrently on different collections. The process consists of four sequential phases:

1. **Identification** — The system identifies document and index versions no longer referenced by active transactions or queries.
2. **Memory loading** — Old documents and index entries are loaded into memory if not already present.
3. **Deletion** — Obsolete versions are permanently deleted to reclaim storage space.
4. **MVCC ID recycling** — The system recycles MVCC IDs from deleted versions for new operations.

When garbage collection completes processing old document versions, it removes the oldest MVCC IDs from the system. This cleanup is crucial for preventing MVCC ID overflow by recycling MVCC IDs, making them available for new write operations across the cluster. Without this recycling process, the system would eventually exhaust its finite MVCC ID counter and enter a read-only state.

Garbage collection scheduling

Garbage collection runs automatically in the background at periodic intervals. The timing and frequency adjust dynamically based on system load, available resources, write volume, and MVCC ID consumption levels. During high write activity, the GC process executes more frequently to manage the increased number of document versions.

Storage architecture and extended storage

Amazon DocumentDB uses a sophisticated storage architecture that separates document storage into two distinct segments:

Base storage segment

The base storage segment contains the primary document data and metadata. This segment stores:

- Document content that fits within the standard page size (8 KB).
- Document metadata and structure information.
- Primary indexes and their entries.
- Collection-level statistics and configuration.

Extended storage segment

The extended storage segment utilizes a specialized large document object store designed to handle documents that exceed the standard storage page size. This segment provides:

- **Efficient Large Document Handling** — Documents larger than the base storage threshold are automatically moved to the extended storage segment.
- **Optimized Storage Layout** — The segment uses a different storage format optimized for large objects, reducing fragmentation and improving access patterns.
- **Independent Garbage Collection** — The extended storage segment has its own garbage collection process that can run independently of base storage cleanup.
- **Transparent Access** — Applications access large documents seamlessly without needing to know which storage segment contains the data.

The extended storage segment is particularly beneficial for:

- Collections with documents containing large embedded arrays.
- Documents with extensive nested structures.
- Collections storing binary data or large text fields.
- Applications with mixed document sizes where some documents significantly exceed average size.

Monitoring garbage collection

Cluster level metrics

`AvailableMVCCIds`

- **Location** — Amazon CloudWatch
- **Description** — A counter that shows the number of remaining write operations available from a maximum limit of 1.8 billion. When this counter reaches zero, your cluster enters read-only mode until IDs are reclaimed and recycled. The counter decreases with each write operation and increases as garbage collection recycles old MVCC IDs.
- **Recommendation** — Set an alarm when the value falls below 900 million on Amazon DocumentDB 5.0 and later, or below 1.3 billion on Amazon DocumentDB 3.6 and 4.0. This early warning allows you to take recommended steps discussed later.

`LongestActiveGCRuntime`

- **Location** — Amazon CloudWatch
- **Description** — Duration in seconds of the longest active garbage collection process. Updates every minute and tracks only active operations, excluding processes that complete within the one-minute window.
- **Recommendation** — Compare with `gcRuntimeStats` historical data to identify abnormal garbage collection behavior, such as extended runtimes during bulk deletions.

Collection level metrics

`MVCCIDStats: MVCCIdScale`

- **Location** — Database collStats command
- **Description** — Measures MVCC ID age on a scale of 0 to 1, where 1 indicates the maximum age before a cluster enters a read-only state. Use this metric alongside `AvailableMVCCIds` to identify collections containing the oldest MVCC IDs that are aging the cluster.
- **Recommendation** — Maintain values below 0.3 for each collection.

`gcRuntimeStats`

- **Location** — Database collStats command
- **Description** — Provides a two-month history of garbage collection metrics, including total runs, average duration, and maximum duration. Only includes garbage collection operations lasting more than five minutes to ensure meaningful statistics.

`storageSizeStats`

- **Location** — Database collStats command
- **Description** — Provides detailed breakdown of storage utilization across different storage segments:
  - `storageSegmentBase` — Storage used by the base storage segment for standard documents
  - `storageSegmentExtended` — Storage used by the extended storage segment for large documents
- **Usage** — Helps identify collections with significant large document storage and understand storage distribution patterns.

`unusedStorageSize` (collection level)

- **Location** — Database collStats command
- **Description** — Estimates unused storage space in a collection based on sampled statistics. It includes space from deleted documents and empty segments. The metric provides both combined totals and per-segment breakdowns:
  - Combined `unusedBytes` and `unusedPercent` across all storage segments
  - `storageSegmentBase` — Unused space specifically in the base storage segment
  - `storageSegmentExtended` — Unused space specifically in the extended storage segment

`documentFragmentStats`

- **Location** — Database collStats command
- **Description** — Provides detailed information about document fragments and dead data within collections. Document fragments represent the internal storage units used by the database engine, and dead fragments indicate data that is no longer accessible but hasn't been reclaimed yet. This metric includes:
  - `totalDocFragmentsCount` — Total number of document fragments in the collection

- `deadDocFragmentsCount` — Number of fragments containing dead (inaccessible) data
- `deadDocFragmentsPercent` — Percentage of fragments that contain dead data
- `deadDocFragmentBytes` — Estimated bytes consumed by dead document fragments
- Per-segment breakdown for `storageSegmentBase` and `storageSegmentExtended`

- **Usage** — Monitor this metric to understand the effectiveness of garbage collection and identify collections that may benefit from maintenance operations. High percentages of dead fragments indicate that garbage collection may be falling behind or that the collection would benefit from optimization.

Index level metrics

`unusedStorageSize` (index level)

- **Location** — Database indexStats command
- **Description** — Estimates unused storage space in an index based on sampled statistics. It includes space from obsolete index entries and empty segments.
- **Recommendation** — Use the `reIndex` command to rebuild indexes without downtime and reclaim unused space. Refer to Managing Indexes for more details.

Example collStats output

The following example shows a typical `collStats` output with garbage collection and storage metrics:

```
{
    "ns" : "Mvcc_consumption_test_db.mvcc_test_collection",
    "MVCCIdStats" : {
        "MVCCIdScale" : 0.03
    },
    "gcRuntimeStats" : {
        "numRuns" : 1,
        "historicalAvgRuntime" : 3295,
        "historicalMaxRuntime" : 3295,
        "lastRuntime" : 3295,
        "lastRuntimeStart" : ISODate("2025-06-24T08:47:14Z")
    },
    "documentFragmentStats" : {
        "totalDocFragmentsCount" : 45000000,
```

```
        "deadDocFragmentsCount" : 2250000,
        "deadDocFragmentsPercent" : 5.0,
        "deadDocFragmentBytes" : 98304000,
        "storageSegmentBase" : {
            "totalDocFragmentsCount" : 30000000,
            "deadDocFragmentsCount" : 1500000,
            "deadDocFragmentsPercent" : 5.0,
            "deadDocFragmentBytes" : 65536000
        },
        "storageSegmentExtended" : {
            "totalDocFragmentsCount" : 15000000,
            "deadDocFragmentsCount" : 750000,
            "deadDocFragmentsPercent" : 5.0,
            "deadDocFragmentBytes" : 32768000
        }
    },
    "collScans" : 14,
    "count" : 30000000,
    "size" : 1320000000,
    "avgObjSize" : 44,
    "storageSize" : 6461497344,
    "storageSizeStats" : {
        "storageSegmentBase" : 4307664896,
        "storageSegmentExtended" : 2153832448
    },
    "capped" : false,
    "nindexes" : 2,
    "totalIndexSize" : 9649553408,
    "indexSizes" : {
        "_id_" : 1910661120,
        "c_1" : 7738892288
    },
    "unusedStorageSize" : {
        "unusedBytes" : 4201881600,
        "unusedPercent" : 65.05,
        "storageSegmentBase" : {
            "unusedBytes" : 2801254400,
            "unusedPercent" : 65.05
        },
        "storageSegmentExtended" : {
            "unusedBytes" : 1400627200,
            "unusedPercent" : 65.05
        }
    },
```

```
    "cacheStats" : {
        "collBlksHit" : 171659016,
        "collBlksRead" : 754061,
        "collHitRatio" : 99.5627,
        "idxBlksHit" : 692563636,
        "idxBlksRead" : 1177921,
        "idxHitRatio" : 99.8303
    },
    "idxScans" : 41823984,
    "opCounter" : {
        "numDocsIns" : 0,
        "numDocsUpd" : 20911992,
        "numDocsDel" : 0
    },
    "lastReset" : "2025-06-24 05:57:08.219711+00",
    "ok" : 1,
    "operationTime" : Timestamp(1750968826, 1)
}
```

Frequently asked questions

How do I identify if garbage collection is not working efficiently?

Monitor these warning signs that indicate inefficient garbage collection:

- **Excessive Collection Bloat** — Steadily increasing `unusedStorageSize` metrics during heavy writes or bulk deletions, especially with large indexes.
- **High Dead Fragment Percentage** — `documentFragmentStats` showing consistently high `deadDocFragmentsPercent` values (above 10-15%).
- **Degraded Query Latency** — Increased query latency due to accumulated dead documents.
- **Extended GC Duration** — Garbage collection operations taking longer than historical averages in `gcRuntimeStats`.
- **Elevated GC Processing** — High `LongestActiveGCRuntime` indicating the garbage collector cannot keep up with system demands.

Does garbage collection affect my database performance?

Under normal conditions, garbage collection has minimal performance impact. However, when garbage collection falls behind, you may experience:

- Increased storage costs from accumulated dead documents.
- Slower query performance due to obsolete index entries.
- Temporary read-only mode if MVCC IDs are depleted.
- Higher resource usage during intensive collection runs, especially on smaller instances.
- Reduced efficiency in extended storage segment operations for large documents.

Can I manually trigger garbage collection?

No, garbage collection in Amazon DocumentDB cannot be manually triggered. The system manages garbage collection automatically as part of its internal maintenance operations.

What alarms should I set as an operational best practice?

Set up monitoring at both the cluster and collection levels to ensure optimal performance of your Amazon DocumentDB system.

For cluster-level monitoring, start by creating a Amazon CloudWatch alarm for the `AvailableMVCCIds` metric with a threshold of 900 million on Amazon DocumentDB 5.0 and later, or 1.3 billion on Amazon DocumentDB 3.6 and 4.0. This gives you adequate time to take action before the metric reaches zero, at which point your cluster would enter read-only mode. Keep in mind that this metric may fluctuate based on your specific usage patterns - some customers see it drop below the threshold and then recover above 1.5 billion as garbage collection completes its work.

It's also important to monitor the `LongestActiveGCRuntime` metric through Amazon CloudWatch. This metric, along with `gcRuntimeStats`, helps you understand how efficiently garbage collection is performing across your system.

For collection-level monitoring, focus on these key metrics:

- `MVCCIdScale` — Watch for increasing values that suggest MVCC IDs are aging and may need attention.
- `gcRuntimeStats` — Identify garbage collection processes taking unusually long or extending over multiple days.
- `documentFragmentStats` — Monitor `deadDocFragmentsPercent` values - consistently high percentages (above 10-15%) may indicate garbage collection is falling behind.

- `storageSizeStats` and `unusedStorageSize` — Track storage utilization patterns and identify collections with significant unused space in either storage segment.

Collections with frequent write operations need extra attention, as they generate more work for the garbage collector. Check these metrics more frequently for collections with heavy write activity to ensure garbage collection keeps up with your workload.

Note that these monitoring recommendations serve as a starting point. As you become more familiar with your system's behavior, you may want to adjust these thresholds to better match your specific usage patterns and requirements.

What should I do if my `AvailableMVCCIds` falls below the alarm threshold?

If your `AvailableMVCCIds` metric drops below the alarm threshold, take immediate action to prevent your cluster from entering read-only mode. First, scale up your instance size to provide the garbage collector with more computing resources. This allows your application to continue normal operations while giving the garbage collector the additional power it needs to catch up.

If scaling up alone doesn't improve the situation, consider reducing your write operations. Use the `MVCCIdScale` metric to identify which specific collections contain older MVCC IDs that need attention. Additionally, monitor `documentFragmentStats` to identify collections with high dead fragment percentages that might be contributing to garbage collection inefficiency.

Once you've identified these collections, you might need to temporarily reduce write operations to them to allow garbage collection to catch up. During the recovery period, closely monitor the `AvailableMVCCIds` metric to ensure your actions are having the desired effect. Your cluster is considered healthy once the `AvailableMVCCIds` value returns to 1.5 billion or higher.

Remember that these steps are preventive measures to help your system recover before it reaches a critical state. The sooner you take action after seeing the metric drop below the threshold, the more likely you are to avoid any impact to your write operations.

Do batch operations affect the MVCC ID consumption rate?

Yes. Because a batch executed within a single atomic transaction consumes only one MVCC ID, grouping writes into batches lowers the rate at which MVCC IDs are consumed. As a best practice, use `insertMany` and `updateMany` whenever possible to minimize MVCC ID consumption and reduce the load on garbage collection.

Why doesn't `AvailableMVCCIds` recover until it reaches a certain threshold?

A steady decline in `AvailableMVCCIds` with no recovery is expected temporarily for a write-dominant workload that has not yet consumed enough MVCC IDs to trigger the garbage collection reclamation cycle. Garbage collection triggers when any of the following conditions are met: a certain portion of the documents within a collection become obsolete, a certain portion of new documents are inserted, or `AvailableMVCCIds` drops below its reclamation threshold.

Until `AvailableMVCCIds` falls below its reclamation threshold, forced reclamation does not occur. Once it does, reclamation triggers and `AvailableMVCCIds` will recover. However, long-running transactions and open cursors can pin the reclaiming horizon and prevent reclamation even after the threshold is reached.

Error codes

When a command fails, you receive an error from Amazon DocumentDB (with MongoDB compatibility) that contains a numeric error `code` and a descriptive message (`errmsg`). Use the following table to identify the error codes that Amazon DocumentDB 8.0 might return to your client application. Where an error code corresponds to a single condition, the description matches the message text returned by the server. Codes that cover multiple conditions have a general description.

> **Important**
>
> Don't build application logic that depends on specific numeric error codes because Amazon DocumentDB might change error codes between major versions.

Error code	Description
1	Item not found
2	A command argument, option, or field value is invalid or unsupported.
4	A required field is missing or a configuration value is not correctly formatted.
5	A view definition references itself, creating a cycle, or a filesystem error occurred.

Error code	Description
9	A command, field, hint, or resume token could not be parsed or has an invalid format.
10	The target collection was not found or no collection was specified.
11	A referenced user could not be found, or the target collection already exists.
13	You are not authorized to run this command, or your authentication is invalid.
14	A field or value has the wrong data type.
15	A value overflowed its allowed range during evaluation.
16	The write batch is empty, malformed, or exceeds the allowed operation count.
17	Not all objects have the same type
18	Authentication failed because the credentials are invalid or the user does not exist.
19	Data processing primitive operation not found or primitive type not found
20	The requested operation is not permitted in this context.
26	The target namespace or collection does not exist.
27	The requested index does not exist or no suitable index is available.
37	A pipeline or aggregation stage argument is missing, empty, or invalid.
38	Database name not found
39	The operation was canceled because it exceeded maxTimeMS, ran too long, or exceeded available memory.
40	An update conflicts with another field, or writes are throttled or disabled by garbage collection.
41	JavaScript error

Error code	Description
43	The specified cursor no longer exists or has expired.
44	Array index out of bounds
48	A collection, view, or namespace with that name already exists.
51	Illegal update operator
52	A field name uses a reserved '$' prefix that is not valid for storage.
53	The _id field has an invalid or unsupported type, such as an array or regex.
54	cannot infer query fields to set
55	Transaction already committed
56	A field name in the request is empty, which is not allowed.
57	Transaction invalidated by other transaction
58	Field value too long
59	The requested command is unknown or not supported.
61	Invalid collection
62	Cannot make changes to collection
65	batch op errors occurred
66	An update attempted to modify the immutable _id field.
67	The index could not be created because of an invalid or unsupported specifica tion.
68	An index with conflicting properties already exists.
70	field used with $snippet is not configured as snippet column
71	Index already exists with different options

Error code	Description
72	One or more options specified for the command or index are invalid.
73	The namespace, database, or collection name is invalid or not allowed for this operation.
74	Collection does not have a text index defined.
75	A text index is defined for this collection, but not for this field.
77	Can't use snapshot with sort
78	Document not found
80	The query exceeds the maximum allowed size.
82	Time frame has invalid formatting. Valid format: '1 m' or '3 y' '<number> h/d/w/s/m/y'
85	An index with that name already exists but with different options.
86	The requested index conflicts with an existing index that has the same name.
87	$sort key ordering must be 1 (for ascending) or -1 (for descending)
92	Not enough space for operation
93	User already exists
94	The target collection, cluster, or resource privilege is invalid or unavailable.
95	Database does not exist
96	'writeConcern' parameter is invalid
97	Password cannot be empty
99	Undefined role or privilege
102	Cannot add a role that is already built-in

Error code	Description
103	A valid role could not be extracted from the collection.
105	The user is invalid or the authorization library could not be initialized.
107	Database cannot be granted roles from other database
111	_id must be exactly specified when providing update document
112	Write Conflict
113	The specified role does not exist.
114	Role already exists
115	anyResource should be true when specified
116	resources should have either db and collection set, cluster set or anyResource set
117	The operation conflicts with another in-progress operation, such as an active transaction.
120	'showPrivileges' cannot be specified as we are showing all users
121	Document failed validation
122	Update error
124	Error processing schema id
125	The command failed, often because of an ambiguous reference or missing session.
126	Database already exists
127	Database name already exists in a difference case name
128	Too many fields
129	Waiting for drop database

Error code	Description
130	Bad parameter value
136	The cursor position was lost, or a required change stream feature is not enabled.
139	JS Interpreter failure
145	Alias not defined in query
147	not master
148	Insufficient resources to run query
149	Alias already defined in query
154	An argument has an illegal value or an incorrect (non-numeric) type.
155	NULL value encountered
156	UNDEFINED value encountered
157	NAN value encountered
158	Collection already in use for another purpose
159	A duration value overflowed or a network operation failed.
161	The requested collation version is invalid or unavailable.
163	A field or name is not valid for storage.
164	A query operator was nested in a way that is not allowed.
165	A view exceeds the maximum nesting depth of 20 or references itself in a cycle.
166	The operation is not supported on a view, or too many positional elements were used.
167	The requested operation or stage is not supported on a view.

Error code	Description
168	An expression or pipeline operator is unrecognized or has an invalid argument.
169	BadValue $maxTimeMS has non-integral value
170	BadValue $maxTimeMS is out of range
171	An index cannot be built over multiple array fields.
173	Skip value must be positive
174	Expected log verbosity to be a number
181	The key pattern matches multiple indexes; specify the index by name instead.
197	The index specification contains an invalid or unsupported field.
207	uuid must be a 16-byte binary field with UUID (4) subtype
224	This query feature is not allowed in the current context.
225	A newer transaction has started on the session, so this older one cannot proceed.
241	A value could not be converted to the requested type in $convert.
250	Invalid view
251	The specified transaction number does not match any active transaction.
252	Recursion limit reached
253	Could not insert entry into view
254	Could not delete entry from view
256	Document filtered by $having clause
257	The total size of all operations in the transaction exceeds the allowed limit.
263	This operation cannot be run inside a multi-document transaction.

Error code	Description
291	unable to find index for $geoNear query
300	Authentication failed because of a SASL handshake or session error.
301	An authentication mechanism or requested operation is not supported.
302	Authentication needs additional data to complete the SASL exchange.
303	The requested stage, option, or role configuration is not supported.
304	The requested aggregation stage is not supported.
305	The aggregation projection operator is not supported.
306	The specified aggregation group accumulator operator is not supported.
502	Column already exists
505	Insertion not allowed here
601	Coordinates are out of bounds
602	Index error
700	Changes to this parameter is not allowed in live mode
701	Invalid value for this parameter
703	Illegal state transition for this parameter
999	Bad transaction ID
4570	$lookup parameter expecting string value
4572	$lookup is missing required fields
10065	A parameter that must be an object (such as vars or $rand) was given a non-object value.
10107	not master

Error code	Description
11000	A duplicate key violates a unique index constraint.
11601	The operation was interrupted.
11603	Client request corrupted or illegal syntax
13103	too many compound keys
13297	A database with the same name already exists but differs only in letter case.
13388	Unrecognized pipeline stage name: $collStats
13435	not master and slaveOk=false
14031	Not enough disk space
15888	must pass name of collection to create
15947	a group's fields must be specified in an object
15950	Group aggregate field name cannot be an operator
15952	unknown $group operator
15955	Syntax error in group aggregation operator: the _id field must be specified
15956	The $skip argument is negative; it must be a non-negative number.
15957	the limit must be specified as a number
15958	The $limit value must be a positive number.
15959	the match filter must be an expression in an object
15969	Syntax error: the value of the $project operator must be a document.
15972	Argument to $skip must be a number
15973	the $sort key specification must be an object

Error code	Description
15974	$sort key ordering must be specified using a number
15975	$sort key ordering must be 1 (for ascending) or -1 (for descending)
15976	$sort stage must have at least one sort key
15981	query field is not a string
15983	An object representing an expression must have exactly one field.
15992	disallowed field type NULL in object expression
15998	Field path names may not be empty strings.
16006	A value could not be converted to a Date because it is not a valid date type.
16007	A value cannot be converted to a string because a string was expected.
16020	An expression operator was called with the wrong number of arguments.
16034	$substrBytes starting index must be a numeric type.
16035	$substrBytes length argument must be a numeric type.
16395	$where is not allowed inside of a $match aggregation expression
16400	can't add an expression for a field because there is already an expression for that field or one of its sub-fields.
16401	can't add an expression for a subfield because there is already an expression that applies to the whole field
16404	$ expressions are not allowed in this context
16406	Cannot exclude fields other than _id
16410	A $project field name or path illegally starts with or contains '$'.
16411	Field names cannot contain null character(s)

Error code	Description
16419	field path must not contain embedded null characters
16424	$near is not allowed inside of a $match aggregation expression
16426	$nearSphere/$near is not allowed inside of a $match aggregation expression
16435	Pipeline component must contain one element
16493	Tried to create string longer than 16MB
16554	$add only supports numeric or date types.
16605	$geoNear requires a valid 'near' field and correctly typed geoNear options.
16606	$geoNear requires a 'distanceField' option as a String
16607	$geoNear requires the given option to be specified as a string.
16608	Divide by zero
16610	$mod was given a divisor of zero.
16611	$mod only supports numeric types for both operands.
16612	only one date allowed in an $add expression
16613	The operation attempted to subtract a non-numeric, non-date value from a date.
16702	$concat only supports string arguments.
16730	textIndexVersion is out of range
16755	can't index geometry with strict winding order
16800	can't have 2 geo fields
16801	2d has to be first in index
16804	location object expected, location array not in correct format

Error code	Description
16810	$mod requires an array of exactly two values as its parameter.
16837	An update path cannot modify the same field more than once.
16868	Invalid variable name
16869	Invalid variable name
16870	Invalid variable name
16872	A field path is empty or '$' alone is used as a field path in $project.
16873	Fieldpath does not start with $
16874	$let only supports an object as its argument
16875	$let was given an unrecognized parameter.
16876	Missing 'vars' parameter to $let
16877	Missing 'in' parameter to $let
16878	$map only supports an object as its argument
16879	$map is given an unrecognized parameter.
16880	$map is missing its required 'input' parameter.
16881	Missing required field
16882	Missing 'in' parameter to $map
16883	The input to $map must be an array.
16898	$minDistance must be non-negative
16899	$maxDistance must be a number
16900	$maxDistance must be non-negative

Error code	Description
16945	Exceeded memory limit for $group or $lookup. Pass allowDiskUse:true to opt in.
16949	Invalid aggregation option; allowDiskUse must be a boolean value.
16990	$out requires a string or object argument naming the output target.
16996	insert for $out failed
16997	$out failed because the target collection's indexes changed during processing.
17040	$allElementsTrue's argument must be an array.
17041	$anyElementTrue's argument must be an array.
17042	Both operands of $setIsSubset must be arrays.
17043	All operands of $setUnion must be arrays.
17044	An operand of $setEquals is not an array.
17045	$setEquals requires at least two arguments.
17046	Both operands of $setIsSubset must be arrays.
17047	All operands of $setIntersection must be arrays.
17048	Both operands of $setDifference must be arrays.
17049	Both operands of $setDifference must be arrays.
17053	$redact's expression should not return anything aside from the variables $$KEEP, $$DESCEND, and $$PRUNE
17080	Missing 'if' parameter to cond
17081	Missing 'then' parameter to cond
17082	Missing 'else' parameter to cond

Error code	Description
17083	Unrecognized parameter to $cond
17124	The argument to $size must be an array.
17275	Can't redefine ROOT
17276	Variable not found
17280	The index key value is too large to index.
17287	$merge cannot modify the same target row more than once.
17293	text index option 'textIndexVersion' must be a number
17302	This argument value must be greater than or equal to zero.
17304	$geoNear requires a valid 'near' point and geo query arguments.
17308	$meta is given an unsupported argument.
17310	Both operands of $setIsSubset must be arrays.
17313	$match with $text is only allowed as the first pipeline stage.
17320	createCollection cannot run on a namespace containing a reserved name.
17385	Cannot $out to special collection
17390	$group does not support inclusion-style expressions
17419	The resulting document after update exceeds the maximum allowed size.
17420	Document to upsert is larger than 16777216
18533	Format field must be a string
18534	Unrecognized field to $dateToString
18535	Date format string is invalid because of a trailing percent sign.

Error code	Description
18536	Invalid $dateToString time format
18537	$dateToString cannot format a date component outside the range 0-9999.
18627	Missing 'format' parameter to $dateToString
18628	Missing 'date' parameter to $dateToString
18629	incorrect argument type
25278	$geoNear requires a 'distanceField' argument
26823	An operation ID cannot be represented within the available bits.
28539	The first argument to the distinct command must be a string
28646	$Filter expects an object as a parameter
28647	Unexpected field to $filter
28648	$Filter requires an 'input' field
28649	$Filter requires an 'as' field
28650	$Filter requires a 'cond' field
28651	Array input required
28656	$substrBytes: Invalid range, starting index is a UTF-8 continuation byte.
28657	$substrBytes: Invalid range, ending index is in the middle of a UTF-8 character.
28664	$concatArrays only supports array inputs.
28667	An expression such as $slice, $range, or $indexOfArray got the wrong number of arguments.
28680	$abs is applied to a value, like the minimum long, that cannot be negated.
28689	An operator argument must be an array but a different type was given.

Error code	Description
28690	Second argument to arrayElemAt must be numeric
28691	Second argument to arrayElemAt must be represented by a 32 bit integer
28714	$sqrt's argument must be greater than or equal to 0
28724	The first argument to $slice must be an array.
28725	The second argument to $slice must be a numeric value.
28726	The second argument to $slice must fit in a 32-bit integer.
28727	The third argument to $slice must be a number.
28728	The third argument to $slice must be a 32-bit integer.
28729	$slice's third argument must be a positive 32-bit integer.
28745	the $sample stage specification must be an object
28746	size argument to $sample must be a number
28747	size argument to $sample must not be negative
28748	$sample received an unrecognized option.
28749	$sample stage must specify a size
28756	$log's argument must be numeric.
28757	$log requires its base argument to be a numeric value.
28758	The argument to $log must be a positive number.
28759	$log's base must be a positive number not equal to 1.
28761	$log10's argument must be a positive number.
28762	$pow's base must be a numeric value.

Error code	Description
28763	$pow's exponent must be numeric.
28764	$pow cannot take a base of 0 and a negative exponent
28765	$ceil, $exp, $floor, $ln, or $log10 receives a non-numeric type.
28766	$ln requires a positive number argument.
28812	Fieldpath is empty
28818	Fieldpath does not start with a '$'
31022	This operator requires an 'input' parameter to be specified.
31023	This operator requires a 'regex' parameter.
31024	The operator is given an unknown argument.
31032	A field name cannot contain an embedded null byte.
31034	A value falls outside the allowed range of [-32768, 32767].
31095	'isoWeekYear' must evaluate to an integer between 1 and 9999.
31138	The $meta value used for sorting is not a valid metadata keyword.
31170	An empty object was expected but a non-empty value was provided.
31249	A projection path collides with another field in the projection.
31254	An inclusion cannot be specified within an exclusion projection.
31393	$bsonSize requires a document as input.
31394	A positional projection operator cannot appear in the middle of a field path.
34413	An invalid operator was used in a geo near query.
34435	The argument to $reverseArray must be an array.

Error code	Description
34443	$range requires a numeric starting value.
34444	$range requires a starting value representable as a 32-bit integer.
34445	$range requires its ending value to be numeric.
34446	$range ending value must fit in a 32-bit integer.
34447	$range requires a numeric step value.
34448	$range requires a step value representable as a 32-bit integer.
34449	$range requires a non-zero step value
34450	$substrCP's starting index must be a numeric type.
34451	The $substrCP starting index cannot be represented as a 32-bit integer.
34452	The length argument to $substrCP is not a numeric type.
34453	$substrCP length cannot be represented as a 32-bit integer.
34454	$substrCP: length must be a nonnegative integer.
34455	$substrCP: the starting index must be nonnegative integer.
34460	$zip only supports an object as its argument.
34461	$zip inputs must be an array of expressions.
34462	The 'defaults' option must be an array of expressions.
34463	$zip's useLongestLength option must be a boolean.
34464	$zip was given an unknown argument.
34465	$zip requires at least one input array
34466	cannot specify defaults unless useLongestLength is true

Error code	Description
34467	defaults and inputs must have the same length
34468	$zip receives a non-array expression in its input.
34471	$strLenCP requires a string argument.
34473	$strLenBytes requires a string argument.
40060	$switch requires an object as its argument.
40061	$switch requires 'branches' to be an array.
40062	$switch requires each branch to be an object.
40063	$switch is given an unknown argument within a branch.
40064	$switch requires each branch have a 'case' expression.
40065	$switch requires each branch have a 'then' expression.
40066	$switch could not find a matching branch for an input, and no default was specified.
40067	The $switch expression contains an unknown argument.
40068	$switch requires at least one branch
40075	$reduce requires object
40076	Unknown argument to $reduce
40077	$reduce requires 'input'
40078	$reduce requires 'initialValue'
40079	$reduce requires 'in'
40080	The input to $reduce must be an array.
40081	The operator requires an array as its second argument.

Error code	Description
40085	$split requires its first argument to evaluate to a string.
40086	$split's second argument must evaluate to a string.
40087	$split requires a non-empty separator
40090	$indexOfArray requires an array as a first argument
40091	The first argument to $indexOfBytes is not a string.
40092	$indexOfBytes requires a string as its second argument.
40093	$indexOfCP requires a string as its first argument.
40094	$indexOfCP requires a string as its second argument.
40096	$indexOfArray requires integral starting and ending index values.
40097	$indexOfArray requires nonnegative numeric start and end indexes.
40148	the sortByCount field must be defined as a $-prefixed path or an expression inside an object
40149	the sortByCount field must be specified as a string or as an object
40156	the count field must be a non-empty string
40157	the count field must be a non-empty string
40158	the count field cannot be a $-prefixed path
40159	the count field cannot contain a null byte
40160	the count field cannot contain '.'
40169	the $facet specification must be a non-empty object
40170	arguments to $facet must be arrays
40171	elements of arrays in $facet spec must be non-empty objects

Error code	Description
40172	sub-pipeline in $facet stage cannot be empty
40176	Conflicting fieldnames given
40177	$project and $addFields require at least one output field.
40178	Cannot exclude fields except '_id' in an inclusion projection
40179	Cannot include fields or reset fields or add computed fields in an exclusion projection
40180	$project does not allow an empty object as a field value.
40181	Invalid $project :: caused by :: an expression specification must contain exactly one field.
40183	A dotted field name cannot be used inside a sub-object.
40192	The $bucket 'boundaries' field must have at least 2 values
40193	All values in the the 'boundaries' option to $bucket must have the same type
40194	The 'boundaries' option to $bucket must be sorted, but elements are not in ascending order
40195	The $bucket 'default' field must be a constant expression
40196	The $bucket 'output' field must be an object
40197	Unrecognized option to $bucket
40198	$bucket requires 'groupBy' and 'boundaries' to be specified
40199	The $bucket 'default' field must be less than the lowest boundary or greater than or equal to the highest boundary.
40200	The $bucket 'boundaries' field must be an array
40201	Argument to $bucket stage must be an object

Error code	Description
40202	The $bucket 'groupBy' field must be defined as a $-prefixed path or an expression
40203	The field must be an accumulator object
40204	The field name cannot contain '.'
40205	The field name cannot be an operator name
40206	The accumulator is a unary operator
40218	query requires text score metadata, but it is not available
40228	$replaceRoot 'newRoot' expression must evaluate to an object.
40229	$replaceRoot requires an object as its specification.
40230	$replaceRoot only accepts the 'newRoot' option.
40234	The $group field must be an accumulator object.
40235	A field name cannot contain a '.' character.
40236	Field names may not start with '$'
40237	This accumulator is a unary operator and takes a single argument.
40239	The $bucketAuto 'groupBy' field must be defined as a $-prefixed path or an expression object
40240	The argument to $bucketAuto must be an object
40241	The $bucketAuto 'buckets' field must be a numeric value
40242	The $bucketAuto 'buckets' field must be representable as a 32-bit integer
40243	The $bucketAuto 'buckets' field must be greater than 0
40244	The $bucketAuto 'output' field must be an object

Error code	Description
40245	Unrecognized option to $bucketAuto
40246	$bucketAuto requires 'groupBy' and 'buckets' to be specified
40257	Unknown rounding granularity
40258	$bucketAuto can specify a 'granularity' with numeric boundaries only
40260	$bucketAuto can specify a 'granularity' with non-negative numbers only, but found a negative number
40261	The $bucketAuto 'granularity' field must be a string
40272	The $addFields stage specification must be an object.
40319	the $lookup stage specification must be an object
40320	missing 'from' option to $lookup stage specification
40321	'from' option to $lookup must be a string
40322	invalid $lookup namespace
40323	A pipeline stage specification object must contain exactly one field.
40324	The pipeline contains an unrecognized stage name.
40331	This aggregation stage is not allowed inside a $facet pipeline.
40352	$project field paths cannot be empty strings.
40353	A field path must not end with a '.'.
40386	$arrayToObject requires an array input.
40390	$objectToArray requires a document as input.
40391	$arrayToObject requires a consistent input format. Elements mustall be arrays or all be objects. Object was detected, now found: array

Error code	Description
40392	$arrayToObject requires objects with exactly the keys 'k' and 'v'.
40393	$arrayToObject requires an object with keys 'k' and 'v'. Missing either or both keys
40394	$arrayToObject requires objects with string 'k' keys and 'v' values.
40395	$arrayToObject requires key-value pairs whose keys are strings.
40396	$arrayToObject requires a consistent input format. Elements mustall be arrays or all be objects. Array was detected, now found: object
40397	$arrayToObject requires an array of two-element arrays.
40398	$arrayToObject is given an input of an unrecognized type or format.
40400	$mergeObjects requires all inputs to be objects.
40414	A required command field is missing or failed validation.
40415	The command contains an unrecognized field.
40422	A BSON array field name has an invalid value.
40485	An unrecognized time zone identifier was supplied.
40489	$dateFromParts does not allow mixing regular calendar fields with ISO week date fields
40515	A $dateFromParts argument does not evaluate to an integer number.
40516	$dateFromParts requires either 'year' or 'isoWeekYear' to be present
40517	The 'timezone' option must evaluate to a string for date operators.
40518	$dateFromParts or $dateToParts got an unrecognized or wrongly typed parameter.
40519	$dateFromParts only supports an object as its argument.

Error code	Description
40520	$dateToParts was given an unrecognized parameter.
40522	$dateToParts requires an object argument with a 'date' parameter.
40523	$dateFromParts 'year' must be an integer in the range 1 to 9999.
40525	$dateFromParts ISO week/day values are out of range or missing required fields.
40533	The timezone argument must be a string.
40535	An unrecognized option was supplied to the stage or expression.
40536	The expression accepts exactly one argument when given an array.
40539	The required 'date' argument to a date operator is missing.
40540	$dateFromString only supports an object as an argument
40541	$dateFromString received an unrecognized argument.
40542	Missing 'dateString' parameter to $dateFromString
40543	$dateFromString requires that 'dateString' be a string.
40545	The date/time string is incomplete, with required elements missing.
40551	you cannot pass in a date/time string with time zone information together with a timezone argument
40553	$dateFromString cannot parse the given date string.
40554	you cannot pass in a date/time string with GMT offset together with a timezone argument
40575	$changeStream 'fullDocument' must be "default" or "updateLookup".
40600	$out cannot be used within a $facet stage.
40601	$out and $merge can only be the final stage in the pipeline.

Error code	Description
40602	A stage like $changeStream or $geoNear is not the first in the pipeline.
40603	$geoNear was not the first stage in the pipeline after optimization. Is optimizat ion disabled or inhibited?
40647	The change stream resume token has a missing or wrong-typed _data field.
40674	Only one type of resume option is allowed, but multiple were found.
50001	autoIndexId:false is not allowed for the specified collection.
50694	This operator was given an unknown argument.
50695	This operator requires an 'input' field.
50696	The operator only supports an object as its argument.
50699	This expression operator requires its input to be a string.
50700	The 'chars' argument must be a string.
50736	getMore cannot run in a session on a cursor not created in a session.
50737	Running getMore on a session-created cursor requires an lsid to be supplied.
50752	The $substrBytes starting index is negative.
50808	$changeStream stage expects a document as its argument.
51024	The collation 'strength' value must be 5 or less.
51047	This operator or stage is not allowed within a $lookup sub-pipeline.
51075	Cannot set options in both $regex and $options
51076	Insufficient memory to build the vector index for the specified number of lists.
51078	Could not find index name.
51079	Latency Stats are not supported

Error code	Description
51091	A regular expression is invalid or badly formed.
51103	The operator expects an object of named arguments but received another type.
51104	The operator requires its 'input' argument to be a string.
51105	An operator's 'regex' argument is not a string or regex type.
51106	An operator requires its 'options' parameter to be a string.
51107	$regex options cannot be specified in both the 'regex' and 'options' fields.
51108	Regular expression options contain an invalid flag or embedded null byte.
51109	A regular expression cannot contain an embedded null byte.
51110	Regular expression options cannot contain an embedded null byte.
51111	The regular expression supplied to this operator is invalid.
51151	The size of buffer to store output exceeded the data type limit
51156	A regular expression failed to execute in $regex during evaluation.
51270	An empty sub-projection object is not a valid projection value.
51272	Syntax error: $projection requires at least one output field.
51744	The operator requires 'replacement' to be a string.
51745	The operator requires its 'find' argument to be a string.
51746	An operator's 'input' argument is not a string.
51747	An operator requires a 'replacement' parameter.
51748	This operator requires a 'find' parameter.
51749	This operator requires an 'input' parameter.

Error code	Description
51750	An aggregation operator received an unknown argument.
100003	Internal system error while processing the optimized query path
100005	A configured limit, such as collections, custom roles, or privileges, has been exceeded.
100006	The maximum number of allowed users has been exceeded.
3040501	$rand does not currently accept arguments
3041701	$getField received an unknown argument in $project.
3041702	Invalid $project :: caused by :: $getField requires 'field' to be specified
3041703	Invalid $project :: caused by :: $getField requires 'input' to be specified
3041704	$getField requires the 'field' argument to evaluate to a string.
4161100	$setField and $unsetField only support an object as their argument.
4161101	$setField or $unsetField was given an unknown argument.
4161102	$setField and $unsetField require a 'field' to be specified.
4161103	Invalid $project :: caused by :: $setField requires 'value' to be specified
4161105	$setField or $unsetField requires 'input' to evaluate to an object.
4161106	$setField and $unsetField require 'field' to evaluate to a constant.
4161107	$setField or $unsetField requires 'field' to evaluate to a string.
4161108	$project disallows a field path reference here; use {$literal: ...} instead.
4161109	$setField and $unsetField require an 'input' field to be specified.
4662500	The command contains an unrecognized field.
4862100	The fully qualified namespace exceeds the maximum allowed length.

Error code	Description
5166301	$dateDiff only supports an object as its argument
5166302	$dateDiff was given an unrecognized argument.
5166303	Missing 'startDate' parameter to $dateDiff
5166304	Missing 'endDate' parameter to $dateDiff
5166305	Missing 'unit' parameter to $dateDiff
5166307	A $dateDiff argument is not a date.
5166308	$dateDiff overflowed while computing the date difference.
5166400	$dateAdd and $dateSubtract require their argument to be an object.
5166401	$dateAdd or $dateSubtract got an unrecognized argument.
5166402	$dateAdd or $dateSubtract require startDate, unit, and amount to be present.
5166403	$dateAdd requires startDate to be convertible to a date
5166404	$dateAdd expects string defining the time unit
5166405	$dateAdd expects integer amount of time units
5166406	dateAdd overflowed
5166503	An unknown time unit value was provided.
5439002	$dateTrunc overflowed while computing the truncated date.
5439006	dateTrunc unsupported binSize value
5439007	$dateTrunc only supports an object as its argument
5439008	Unrecognized $dateTrunc argument; expected date, unit, binSize, timezone, startOfWeek.
5439009	Missing 'date' parameter to $dateTrunc

Error code	Description
5439010	Missing 'unit' parameter to $dateTrunc
5439012	$dateTrunc requires 'date' to be a date.
5439013	$dateDiff and $dateTrunc require 'unit' to be a string.
5439014	$dateDiff and $dateTrunc 'unit' value is not a recognized time unit.
5439015	$dateDiff or $dateTrunc require 'startOfWeek' to be a string.
5439016	$dateDiff or $dateTrunc 'startOfWeek' value is not a valid day of the week.
5439017	$dateTrunc requires 'binSize' to be a 64-bit integer.
5439018	$dateTrunc requires 'binSize' to be greater than 0.
5447000	The $collStats stage must be a nested object.
5626500	$geoNear, $near, and $nearSphere are not allowed in this context, as these operators require sorting geospatial data. If you do not need sort, consider using $geoWithin instead.
5654600	$project does not allow a field path reference in this context.
5654601	Invalid $project :: caused by :: $getField requires 'field' to evaluate to a constant, but got a non-constant argument
5654602	$getField requires 'field' to evaluate to a string.
5733415	can't $mod by zero
5860400	$geoNear requires a 'near' argument
5887502	All operands of $setEquals must be arrays.
5911200	PlanExecutor error during aggregation :: caused by :: $mergeObjects only supports objects
5976500	$dateAdd was given an invalid 'amount' parameter value.

Error code	Description
6045000	$dateSubtract was given an invalid 'amount' parameter value.
7158303	$switch found no matching branch for the input and no default was specified.

Release notes

These release notes describe the Amazon DocumentDB features, improvements, and bug fixes organized by major engine version. Within each version section, entries are listed by date in a table format.

For details about how engine versions and patch versions are numbered (including the mapping between engine patch version prefixes and Amazon DocumentDB engine versions), see Engine version numbering.

You can determine the current Amazon DocumentDB engine patch version by running the following command:

```
db.runCommand({getEngineVersion: 1})
```

> **Note**
>
> Starting with Amazon DocumentDB 5.0.2 and 8.0.2, this command returns the engine version and the engine patch version as a single string in the form *engine-version*+*engine-patch-version*. For example, a cluster on Amazon DocumentDB 5.0.2 returns `5.0.2+3.0.12345`, and a cluster on 8.0.2 returns `8.0.2+4.0.12345`. Clusters on earlier versions return the engine patch version on its own, such as `3.0.12345`. For more information, see Engine version numbering.

If your cluster is not on the latest version of the engine, it is likely that you have pending maintenance available that will upgrade your engine. For more information, see Maintaining Amazon DocumentDB in the Developer Guide.

Topics

- Amazon DocumentDB engine version release notes
- General Amazon DocumentDB release notes
- Amazon DocumentDB elastic clusters release notes

Amazon DocumentDB engine version release notes

Release notes for Amazon DocumentDB engine versions organized by major version.

Amazon DocumentDB 8.0

Release notes for Amazon DocumentDB engine version 8.0 (MongoDB 8.0 compatibility).

Amazon DocumentDB 8.0.3

Date	Release notes
November 11, 2026	**Amazon DocumentDB 8.0.3 (Engine Version: 4.0.13572)** **New features and improvements** 1. Added hybrid post-quantum key exchange for TLS 1.3 connectio ns. It uses the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM). Clients that support ML-KEM use post-quantum key exchange automatically. Other clients continue to use classical key exchange. For more information, see Amazon DocumentDB post-quantum key exchange. 2. Added the `TLSHandshakesClassical` and `TLSHandsh akesPostQuantum` Amazon CloudWatch (CloudWatch) metrics. These metrics count the successful TLS handshakes that use classical key exchange and hybrid post-quantum key exchange, respectively, on each instance. For more information, see Amazon DocumentDB metrics. 3. Added support for wildcard indexes. 4. Added support for the `$unionWith` aggregation stage. 5. Added support for `$lookup` and `$graphLookup` in views created with `createView` . 6. Added support for correlated `$lookup` with expanded sub-pipeline stages, adding `$unionWith` and `$facet` to the set of allowed stages. 7. Enhancements to the `$facet` aggregation stage.

Date	Release notes
	8. Change streams now report successful index removals from `dropIndex()` and `dropIndexes()` as `dropIndexes` events. 9. Added per-collection compression metrics (`input_bytes`, `output_bytes`, `compression_operations`, `compression_ratio`) to `collStats` and `stats()` output. For more information, see Compressor statistics. **Bug fixes** 1. Fixed the retryable write error label returned by `findAndModify`. 2. Fixed an issue with sort when the query plan uses disjoint index scans.

Amazon DocumentDB 8.0.2

Date	Release notes
September 28, 2026	**Amazon DocumentDB 8.0.2 (Engine Version: 4.0.12042)** **New features and improvements** 1. Added support for the `$setWindowFields` aggregation stage and its window operators in Planner version 3. 2. Added support for the `$bucketAuto` aggregation stage in Planner version 3. 3. Added support for the `$facet` aggregation stage in Planner version 3. 4. Added support for the `$graphLookup` aggregation stage in Planner version 3. 5. Added support for correlated `$lookup` subqueries, including the concise correlated subquery syntax, with a subset of pipeline stages in Planner version 3.

Date	Release notes
	6. Added support for the `$changeStreamSplitLargeEvent` aggregation stage. 7. Change streams now emit events for `createCollection` operations. 8. Change streams now emit events for `createIndex` operations. 9. Improved change stream write performance. 10 Added support for retryable writes. 11 Added support for the `reIndex` command on partial indexes. 12 Planner version 3 now supports index scans on `$expr` predicates. 13 Planner version 3 introduces multi-field Index Only Scans (IXONLYSCAN) for find queries. For more information, see Index-only scans (covered queries). 14 Planner version 3 introduces Index Only Scans (IXONLYSCAN) for aggregation stages such as `$group`. For more information, see Index-only scans (covered queries). 15 Performance optimization for `$count` and `countDocuments()` in Planner version 3. 16 Added support for incremental sort in Planner version 3. For more information, see Incremental sort. 17 Improved plan selection for text index queries. 18. The explain plan for coalesced `$lookup` and `$unwind` stages now reflects the `includeArrayIndex` parameter. 19 Added index filter support across all commands. 20. The query planner now prioritizes unique indexes for equality (`$eq`) predicates. 21 Increased the truncation limit for aggregate commands in `currentOp` output. **Bug fixes** 1. Fixed a bug in the version 1 (default) query planner.

Date	Release notes
	2. Fixed partial index expression text round-trip handling during Major Version Upgrade (MVU). 3. Fixed a profiler issue for fast count queries. 4. Security fix for the `collStats` command.

Amazon DocumentDB 8.0.1

Date	Release notes
October 19, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.1 (Engine Version: 4.0.12164) [in progress]** This engine patch provides the following improvements: • **Security improvements** Authentication handshake hardening. • **Improved latency stability** Reduced intermittent latency spikes that could occur during routine internal configuration reloads. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
September 15, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.1 (Engine Version: 4.0.11638)** Amazon DocumentDB automatically applies this patch during your next maintenance window.
August 31, 2026	**New feature** Amazon DocumentDB now supports in-place major version upgrades (MVU) directly from engine versions 3.6 and 4.0 to version 8.0. This

Date	Release notes
	removes the need for intermediate version upgrades and lets you upgrade your 3.6 or 4.0 clusters while preserving existing data, configurations, and cluster settings. Major version upgrades from Amazon DocumentDB 3.6 and 4.0 to 8.0 are available in all AWS Regions where these versions are currently supported. For more information, see Amazon DocumentDB in-place major version upgrade.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.1 (Engine Version: 4.0.9938)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns. • **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input. Amazon DocumentDB automatically applies this patch during your next maintenance window.
August 11, 2026	**New features** **Amazon DocumentDB 8.0.1 (Engine Version: 4.0.8864)** • Enabled support for SCRAM-SHA-256 authentication. • Enabled tunable RPO for Amazon DocumentDB global clusters. • Improved storage reclamation during background garbage collection to reduce unused storage space over time.

Date	Release notes
July 22, 2026	**New feature** Amazon DocumentDB clusters running Engine Version 8.0 and later now support a maximum storage limit of 256 TiB, doubling the previous limit of 128 TiB. For more information, see Amazon DocumentDB quotas.

Date	Release notes
July 13, 2026	**Amazon DocumentDB 8.0.1 (Engine Version: 4.0.8864)** **New features and improvements** 1. Support for 13 accumulator operators in default Planner (v3): `$top`, `$bottom`, `$topN`, `$bottomN`, `$count`, `$firstN`, `$lastN`, `$minN`, `$maxN`, `$median`, `$percentile` , `$stdDevPop` , `$stdDevSamp` . 2. Support for 36 expression operators in default Planner (v3): `$median`, `$percentile` , `$stdDevPop` , `$stdDevSamp` , `$round`, `$sigmoid`, `$trunc`, `$firstN`, `$lastN`, `$maxN`, `$minN`, `$bitAnd`, `$bitNot`, `$bitOr`, `$bitXor`, `$binarySize` , `$bsonSize` , `$tsIncrement` , `$tsSecond` , `$cos`, `$acos`, `$cosh`, `$acosh`, `$sin`, `$asin`, `$sinh`, `$asinh`, `$tan`, `$atan`, `$tanh`, `$atan2`, `$atanh`, `$degreesToRadians` , `$radiansToDegrees` , `$isNumber` , `$toUUID`. 3. Support for `$sampleRate` in `$match` in default Planner (v3). 4. Support for `$sortByCount` in default Planner (v3). 5. Added support for `$listSearchIndexes` aggregation stage in default Planner (v3). Returns search index metadata (name, status, queryable, version) for text and vector search indexes on a collection. 6. Support snappy compression using the OP_COMPRESSED message type. 7. Support the `min` and `max` cursor methods on the find command in default Planner (v3). 8. Planner version 3 introduces Index Only Scans (IXONLYSCAN) for `countDocuments()` . 9. Added hidden index support for all index types (planner version 2 and above). **Bug fixes**

Date	Release notes
	1. Fixed an issue where `$text` search queries containing underscore characters could return incorrect results when evaluated as a residual filter.

Amazon DocumentDB 8.0.0

Date	Release notes
October 19, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.0 (Engine Version: 4.0.12167) [in progress]** This engine patch provides the following improvements: • **Security improvements** Authentication handshake hardening. • **Improved latency stability** Reduced intermittent latency spikes that could occur during routine internal configuration reloads. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
September 15, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.0 (Engine Version: 4.0.11440)** **Bug fixes** This release includes the following bug fix: 1. Fixed an issue where `$text` search queries containing underscore characters could return incorrect results when evaluated as a residual filter.

Date	Release notes
	Amazon DocumentDB automatically applies this patch during your next maintenance window.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 8.0.0 (Engine Version: 4.0.9912)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns. • **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
July 22, 2026	**New feature** Amazon DocumentDB clusters running Engine Version 8.0 and later now support a maximum storage limit of 256 TiB, doubling the previous limit of 128 TiB. For more information, see Amazon DocumentDB quotas.
May 13, 2026	**New feature** Amazon DocumentDB serverless is now available for Amazon DocumentDB 8.0. For more information, see Using Amazon DocumentDB serverless.

Date	Release notes
May 6, 2026	**Bug fixes and other changes** **Amazon DocumentDB 8.0 (Engine Version: 4.0.6940)** This update provides an optional engine patch with the following improvements: 1. Fixed a bug in garbage collection that could, in an edge case, cause readers to crash.
April 20, 2026	**New feature** In-place major version upgrade (MVU) from Amazon DocumentDB 5.0 to Amazon DocumentDB 8.0 is now available. For more informati on, see Amazon DocumentDB in-place major version upgrade.

Date	Release notes
April 20, 2026	**New features and improvements** **Amazon DocumentDB 8.0 (Engine Versions: 4.0.5995, 4.0.6473)** This is a required engine patch that includes the following new features, improvements, and fixes: 1. `$sortArray` operator in `$project`: • Support for `$sortArray` operator in `$project` stage in aggregation pipeline for Planner version 3. The `$sortArray` operator sorts an array based on its elements (for scalar values) or on a specified field (for document arrays), returning the sorted array in ascending (1) or descending (-1) order. 2. Support for `$out`, `$collStats`, `$indexStats`, `$geoNear` using Planner v3. With Planner V3, the `$out` stage also supports writing to collections across different databases. 3. The aggregation pipeline now coalesces `$lookup` and `$unwind` stages even when the `$unwind` stage includes the `includeArrayIndex` parameter. Previously, this optimization was skipped when `includeArrayIndex` was present. 4. Performance enhancements to improve write throughput for db.r8g.8xlarge and larger instances running on I/O-Optimized configuration. 5. You can now monitor the `Connections (active)` metric for instances. For more information, see Instance quotas. 6. Support for `$dateToParts` and `$dateFromParts` aggregation operators. 7. Command-level performance metrics are now available in CloudWatch for CRUD operations (`find`, `insert`, `findAndModify`, `update`, `delete`, `aggregate`, `count`, `distinct`, `getMore`, `abortTransaction`, `commitTransaction`), exposing `AvgDuration`, `P100Duration`, and `MaxConcurrent` dimensions for improved operational monitoring.

Date	Release notes
	8. Fixed a bug in `$replaceRoot` behavior where `newRoot` fields containing boolean values of `true/false` or numeric values of `0/1` were missing in the output document. 9. Fixed a bug in `$indexOfArray` where valid queries were returning errors indicating that they were invalid. 10. The explain plan now includes `docsExamined` metrics in `executionStats` output. 11. Improved SASL authentication performance. 12. Optimized document updates to reduce storage overhead when index values remain unchanged and documents fit within existing pages. 13. Support for `bypassDocumentValidation` in `$merge`. All Amazon DocumentDB 8.0 clusters will be automatically upgraded to this patch version during their next maintenance window.
March 26, 2026	**New features** Amazon DocumentDB now supports the latest generation of memory-optimized instances, db.r8g, powered by Arm-based AWS Graviton4 processors, providing up to 30% better performance over R6G instances. For more information, see Managing instance classes in this guide

Date	Release notes
November 11, 2025	**New features** Amazon DocumentDB now offers full wire protocol compatibi lity with MongoDB 8.0. Amazon DocumentDB 8.0 improves query performance enabling you to build high-performance applications at a lower cost. **Amazon DocumentDB 8.0 (Engine Patch Version 4.0.4448)** • Added support for MongoDB 8.0 API drivers. • Added support for Planner Version3 that extends performan ce improvements to aggregation stage operators, along with supporting aggregation pipeline optimizations and distinct commands. • Added Support for dictionary-based compression through the Zstandard compression algorithm, improving compression ratio by 5x. • Added new capabilities: Amazon DocumentDB 8.0 supports collation and views. • Added support for new aggregation stages and operators: $replaceWith, $vectorSearch, $merge, $set, $unset, $bucket, and 3 new aggregation operators $pow, $rand, $dateTrunc. • Added support for a new version of text index: Text index v2 in Amazon DocumentDB 8.0 introduces additional tokens, enhancing text search capabilities. • Includes vector search improvements through parallel vector index build reducing index build time by up to 30x. • Removed support for TLS 1.0 and TLS 1.1. The minimum TLS version is now TLS 1.2.

Amazon DocumentDB 5.0

Release notes for Amazon DocumentDB engine version 5.0 (MongoDB 5.0 compatibility).

Amazon DocumentDB 5.0.2

Date	Release notes
September 28, 2026	**Amazon DocumentDB 5.0.2 (Engine Version: 3.0.23065)** **New features and improvements** 1. Added hidden index support for all index types (planner version 2 and above). 2. Support for the `min()` and `max()` cursor methods on the find command. 3. Added index filter support across all commands. 4. The query planner now prioritizes unique indexes for equality ($eq) predicates. 5. Increased the truncation limit for aggregate commands in `currentOp` output. **Bug fixes** 1. Fixed a bug in the version 1 (default) query planner. 2. Fixed partial index expression text round-trip handling during Major Version Upgrade (MVU). 3. Fixed a profiler issue for fast count queries.

Amazon DocumentDB 5.0.1

Date	Release notes
November 2, 2026	**Security and stability improvements** **Amazon DocumentDB 5.0.1 (Engine Version: 3.0.23310) [in progress]** This engine patch provides the following improvements:

Date	Release notes
	• **Security improvements** Authentication handshake hardening. • **Improved latency stability** Reduced intermittent latency spikes that could occur during routine internal configuration reloads. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 5.0.1 (Engine Version: 3.0.22319)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns. • **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
August 11, 2026	**New feature** **Amazon DocumentDB 5.0.1 (Engine Version: 3.0.21138)** Enabled support for SCRAM-SHA-256 authentication.

Date	Release notes
June 8, 2026	**New minor version** **Amazon DocumentDB 5.0.1 (Engine Version 3.0.21138)** Amazon DocumentDB 5.0.1 is the first minor version release for the Amazon DocumentDB 5.0 major version. Minor versions include improvements such as new features, security fixes, and bug fixes. You can upgrade to 5.0.1 from 5.0.0 using a minor version upgrade. For more information, see Amazon DocumentDB minor version upgrade. This minor version includes the following new features and improvements: 1. Added `gcRuntimeStats` , `documentFragmentStats` , and breakup of collection level metrics into `storageSegmentBase` and `storageSegmentExtended` in `collstats` to surface more information into the overall health of the garbage collection process. See Garbage collection in Amazon DocumentDB for details. 2. Support for `$rand` and `$pow` aggregation operators. 3. You can now monitor the `Connections (active)` metric for instances. For more information, see Instance quotas. 4. Improved Global Cluster cross-Region resilience by eliminating secondary cluster reboot with Global Survivable Read Only. 5. Support for `$dateToParts` and `$dateFromParts` aggregation operators. 6. Command-level performance metrics are now available in CloudWatch for CRUD operations (`find`, `insert`, `findAndModify` , `update`, `delete`, `aggregate` , `count`, `distinct`, `getMore`, `abortTransaction` , `commitTransaction`), exposing `AvgDuration` , `P100Duration` , and `MaxConcurrent` dimensions for improved operational monitoring. 7. `$replaceRoot` outputs `newRoot` fields containing boolean values (`true`/`false`) and numeric values (`0`/`1`).

Date	Release notes
	8. `$indexOfArray` supports a broader range of valid query expressions. 9. The explain plan includes `docsExamined` metrics in `executionStats` output. 10 Improved SASL authentication performance.

Amazon DocumentDB 5.0.0

Date	Release notes
November 2, 2026	**Security and stability improvements** **Amazon DocumentDB 5.0.0 (Engine Version: 3.0.23157) [in progress]** This engine patch provides the following improvements: • **Security improvements** Authentication handshake hardening. • **Improved latency stability** Reduced intermittent latency spikes that could occur during routine internal configuration reloads. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 5.0.0 (Engine Version: 3.0.22303)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns.

Date	Release notes
	• **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input. • **Access-control enforcement** Amazon DocumentDB now verifies required privileges consistently at all nesting levels for nested query operations. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
May 6, 2026	**Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Version: 3.0.20421)** This release is for a required security patch.
April 20, 2026	**New feature** In-place major version upgrade (MVU) from Amazon DocumentDB 5.0 to Amazon DocumentDB 8.0 is now available. For more informati on, see Amazon DocumentDB in-place major version upgrade.

Date	Release notes
April 13, 2026	**Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Versions: 3.0.19917)** This is a required engine patch that includes the following new features, improvements, and fixes: 1. The `DatabaseConnectionsActiveMax` metric now tracks `Connections (active)` for instances. For more information, see Instance quotas. 2. Improved SASL authentication performance. All Amazon DocumentDB 5.0 clusters will be automatically upgraded to this patch version during their next maintenance window.

Date	Release notes
October 22, 2025	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.17983)** Amazon DocumentDB now supports the latest generation of memory-optimized instances, db.r8g, powered by Arm-based AWS Graviton4 processors, providing up to 30% better performance over R6G instances. For more information, see Managing instance classes in this guide and this blog post. **New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.15902)** • Added support for longer collection name (namespace length up to 255 chars). • Introduced a new query planner (planner version 2.0). For more information, see Query planner v2 in this guide. • Added new fields in `collstats` for "MVCCIDStats" and "gcRuntimeStats" that provide information into the overall health of the collection with respect to garbage collection and aging collections.
October 16, 2025	**New feature** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.15902)** Amazon DocumentDB now supports dual-stack mode (IPv4/IPv6) when your cluster is in a virtual private cloud (VPC). For more information, see this blog post and Amazon VPC and Amazon DocumentDB in the *Amazon DocumentDB Developer Guide*.

Date	Release notes
September 15, 2025	**Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.17983)** Fixed engine crash when queries contain subsequent `$replaceRoot` and `$lookup` stage.
July 29, 2025	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.15902)** • Added support of `$regexFindAll` aggregation operator. • Index bloat and collection based bloat metrics launched • Added CloudWatch metrics `AvailableMVCCIds` and `LongestActiveGCRuntime` that provide information into the overall health of the cluster with respect to garbage collection and aging collections. • The `serverStatus` command now includes an "nvme_cache" field that shows the count of pages written and not written to NVMe cache on NVMe-backed instances. • Removed limitation on number of fields for `$group` and `$project` • Support for the following String operators: `$trim`, `$rtrim`, `$ltrim`, `$regexFindAll` , `$replaceOne` , and `$replaceAll` • Support for the following Array operators: `$first` and `$last` • Support for the following Stage operator: `$collStats` **Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.15902)** Fixed an issue during index creation that affected large collections.

Date	Release notes
July 28, 2025	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.14228)** Amazon DocumentDB instance-based clusters now support Amazon DocumentDB Serverless, an on-demand, auto-scaling configuration. For more information, see Using Amazon DocumentDB serverless and this what's new blog post.
March 24, 2025	**New feature** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.11800)** Added support for `postBatchResumeToken` in change streams. For more information, see Resuming a change stream with `postBatchResumeToken`.
January 15, 2025	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.11800)** Amazon DocumentDB now maintains read availability through writer instance restarts. Reader instances will now continue to serve read requests during writer instance restarts. **Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.11800)** • Fixed `killOp` command to handle a special case of bulk inserts. • Improved network I/O-related memory usage on Amazon DocumentDB instances. • Fixed `count` command for `$text` filter queries.

Date	Release notes
December 18, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.5721)** Amazon DocumentDB instance-based clusters now support NVMe-backed R6GD instances. For more information, see NVMe-backed instances and this what's new blog post.
November 6, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.11051)** • Added support for minimum TLS versions `tls1.2+` and `tls1.3+` to `tls` cluster parameter. • Enabled support for characters dollar($) and dot(.) in field names. For functional differences, see Dollar($) and dot(.) in field names.

Date	Release notes
September 17, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.10696)** Amazon DocumentDB now supports rebuilding indexes with `reIndex` in the `runCommand` . For more information, see Index maintenance using `reIndex`. **Note** `reIndex` is only supported on Amazon DocumentDB 5.0 (Engine Patch Version 3.0.10696 and higher). **Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.10696)** • `$setOnInsert` now supports the `_id` field during inserts resulting from update operations. • Fixed issue that prevents reclaiming storage space after a major version upgrade to Amazon DocumentDB 5.0.

Date	Release notes
August 22, 2024	**New feature** **Amazon DocumentDB 5.0 (All engine patch versions)** Amazon DocumentDB global clusters now support cluster switchovers and managed cluster failovers. For more information, see Performing a switchover for an Amazon DocumentDB global cluster and Performing a managed failover for an Amazon DocumentDB global cluster. **Note** Global cluster switchovers and failovers are supported on Amazon DocumentDB 4.0 and 5.0 only.

Date	Release notes
July 23, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.8126)** • Added support for longer index names (up to 255 characters). For more information, see Naming constraints • Increased the maximum connection limit by two-fold:

Instance type	Original limit	New limit
t3.medium	500	1000
t4g.medium	500	1000
r5.large	1700	3400
r5.xlarge	3500	7000
r5.2xlarge	7100	14200
r5.4xlarge	14200	28400
r5.8xlarge	28400	60000
r5.12xlarge	30000	60000
r5.16xlarge	30000	60000
r5.24xlarge	30000	60000
r6g.large	1700	3400
r6g.xlarge	3500	7000
r6g.2xlarge	7100	14200
r6g.4xlarge	14200	28400
r6g.8xlarge	28400	60000

Instance type	Original limit	New limit
r6g.12xlarge	30000	60000
r6g.16xlarge	30000	60000

For more information, see Instance quotas.

Bug fixes and other changes

Amazon DocumentDB 5.0 (Engine Patch Version 3.0.8126)

Enhanced the logic to synchronize `CurrentTime` and `ResumeToken` for the change stream on readers.

Date	Release notes

July 22, 2024	**New features**
	
	Amazon DocumentDB 5.0 (Engine Patch Version 3.0.6742)
	
	• Added support DML Audit filtering. You can now setup filter conditions to filter out DML Audit logs based on their specific requirements instead of logging every DML query. For more information, see Filtering DML audit events.
	• Added support in document compression for the following:
	  • Setting a minimum compression threshold
	  • Enabling compression for existing collections (applicable to new documents)
	  • Allow default compression setting at the cluster level
	
	For more information, see Managing collection-level document compression.
	• Added support for consuming change streams on reader instances . For more information, see Using change streams on secondary instances.

Date	Release notes
July 9, 2024	**New feature** For Amazon DocumentDB 5.0 (minimum engine patch version 3.0.4780), TLS CA certificate updates no longer require a system reboot. For more information, see Updating your Amazon DocumentDB TLS certificates.
May 29, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.6742)** • Added support for `$regexMatch` and `$regexFind` operators. • Added support to ensure full precision in audit logs when addressing large integers. Audit logs now maintain the exact numeric representation for all numbers, preventing any loss of precision.
April 3, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.5721)** • Added support for `bypassDocumentValidation` and granular error message for `$jsonSchema`. For more information about `bypassDocumentValidation`, see bypassDocumentValidation. • Added support of `$expr`. • Added support for Uncorrelated Joins in `$lookup`. • Added support to retain validation rules in `$out` aggregation stage.

Date	Release notes
January 10, 2024	**New features** **Amazon DocumentDB 5.0 (Engine Patch Versions 3.0.4574, 3.0.4780, 3.0.4960)** • Added support for HNSW vector indexes. For more information, see Vector search for Amazon DocumentDB. • Added a support for partial indexes. For more information, see Partial index. • Added a support for GC runtime on a collection within `currentOp` command. • Added text index support for native text search on Amazon DocumentDB. For more information, see Performing text search with Amazon DocumentDB. • Added support for `$jsonSchema` schema keywords `type`, `allOf`, `oneOf`, `anyOf`, `not`, `maxItems`, `minItems`, `maxProperties` , `minProperties` , `pattern`, `patternProperties` , `multipleOf` , `dependencies` , and `uniqueItems` . For more information see Using JSON schema validation. • Added support for arithmetic operators `$ceil`, `$floor`, `$ln`, `$log`, `$log10`, `$sqrt`, and `$exp`. For more information see Arithmetic operators. • Added support for the conditional expression operator `$switch`. • Added support for parallel `IVFFLAT` vector index builds. Documentation was updated by removing the parallel `IVFFLAT` vector index builds limitation from the developer guide. **Bug fixes and other changes** • Added back case-insensitive functionality for invoking `db.runCommand("dbstats")` . Amazon DocumentDB 5.0customers on

Date	Release notes
	engine patch versions prior to 3.0.4960 or 2.0.10221 should apply these latest engine patches. • Fixed an error when invoking `db.coll.stats()` on mongo shell version 1.7 and later. Documentation was updated by removing the mongo shell `db.coll.stats()` troubleshooting tip from the developer guide.
November 29, 2023	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.3727)** Added support for vector search. For more information, see this blog post and visit Vector search for Amazon DocumentDB in the *Amazon DocumentDB Developer Guide*.
November 21, 2023	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.3727)** Added support for I/O-optimized storage. For more information, see Amazon DocumentDB cluster storage configurations in the *Amazon DocumentDB Developer Guide*. Added integration for no-code machine learning with SageMaker Canvas. For more information, see No-code machine learning with Amazon SageMaker AI Canvas in the *Amazon DocumentDB Developer Guide*.

Date	Release notes
November 6, 2023	**Amazon DocumentDB 5.0 (Engine Patch Versions 3.0.3727)** **New features** • Added support for `$jsonSchema` schema keywords `maxLength`, `minLength`, `maximum`, `minimum`, `exclusiveMaximum`, `exclusiveMinimum`, `items`, and `additionalItems`. Note that JSON schema validation is supported in instance-based clusters only. • Added support for `$convert` aggregation pipeline operator and its shorthand derived operators `$toBool`, `$toInt`, `$toLong`, `$toDouble`, `$toString`, `$toDecimal`, `$toObjectId`, and `$toDate`. • Added support for set expression operators `$setDifference`, `$anyElementTrue`, and `$allElementTrue`. **Bug fixes and other changes** Fixed issue where a change stream update from `-NaN` to `NaN` was not being displayed.
September 15, 2023	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.3140)** Added support for $jsonSchema schema validator in instance-based clusters only. For more information see Using JSON schema validation.

Date	Release notes
July 13, 2023	**New features** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.1948)** • Added support for document compression. • Added support for parallel index builds. • Added support for index build status. **Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.1948)** • Fixed authentication issue with `createCollection` for Amazon DocumentDB elastic clusters when users don't have access to system collections. • Fixed issue where secondary Region instances couldn't use the same primary Region instance names.
June 7, 2023	**Bug fixes and other changes** **Amazon DocumentDB 5.0** • r5 and t3.medium instances are now supported in Amazon DocumentDB 5.0. • `engineVersion` option default is `5.0.0` in AWS SDK, AWS CLI, and CloudFormation.
May 10, 2023	**Bug fixes and other changes** **Amazon DocumentDB 5.0 (Engine Patch Version 3.0.1361)** • Added support for `ignoreunknownindexoptions` in the **createIndex** command. • Stopped adding internal monitoring queries to auditing logs. • User defined local variable names now support "_" (underscore) for projection operators such as `$let` and `$filter`.

Date	Release notes

March 1, 2023	**New features**
	
	Amazon DocumentDB 5.0 (Engine Patch Version 3.0.775)
	
	• Introduced Amazon DocumentDB 5.0
	  • MongoDB 5.0 compatibility (support for MongoDB 5.0 API drivers)
	  • Support for Client-side Field Level Encryption (FLE). You can now encrypt fields at the client-side before writing the data to Amazon DocumentDB cluster. For more information, see Client-side field level encryption
	  • New aggregation operators: `$dateAdd`, `$dateSubtract`
	• Increased storage limit to 128 TiB for all instance-based Amazon DocumentDB clusters and shard-based elastic clusters.
	• Amazon DocumentDB 5.0 now supports index scan with the `$elemMatch` operator in the first nesting level. Index scans are supported when query only have one level of the `$elemMatch` filter and the nested `$elemMatch` query does not support index scan.
	
	Query shape that supports index scan:
	
	`db.foo.find( { "a": {$elemMatch: { "b": "xyz", "c": "abc"} } })`
	
	Query shape that does not support index scan:
	
	`db.foo.find( { "a": {$elemMatch: { "b": {$elemMatch: { "d": "xyz", "e": "abc"} }} } })`

Amazon DocumentDB 4.0

Release notes for Amazon DocumentDB engine version 4.0 (MongoDB 4.0 compatibility).

Date	Release notes
August 31, 2026	**New feature** Amazon DocumentDB now supports in-place major version upgrades (MVU) directly from engine version 4.0 to version 8.0. This removes the need for intermediate version upgrades and lets you upgrade your 4.0 clusters while preserving existing data, configurations, and cluster settings. This upgrade is available in all AWS Regions where these versions are currently supported. For more information, see Amazon DocumentDB in-place major version upgrade.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 4.0 (Engine Version: 2.0.14287)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns. • **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input. • **Access-control enforcement** Amazon DocumentDB now verifies required privileges consistently at all nesting levels for nested query operations. If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
April 27, 2026	**Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Version: 2.0.14278)**

Date	Release notes
	This is a required engine patch that addresses the following issues: 1. Fixed an issue that could, in rare cases, cause point-in-time recovery (PITR) and major version upgrades (MVU) to become unresponsive. All Amazon DocumentDB 4.0 clusters will be automatically upgraded to this patch version during their next maintenance window.
March 9, 2026	**Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Version: 2.0.14165)** This update provides an optional engine patch with the following improvements: 1. The `DatabaseConnectionsActiveMax` metric now tracks `Connections (active)` for instances. For more information, see Instance quotas. 2. Improved SASL authentication performance.
October 22, 2025	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.11747)** Added new fields in `collstats` for "MVCCIDStats" and "gcRuntim eStats" that provide information into the overall health of the collection with respect to garbage collection and aging collections.
October 16, 2025	**New feature** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.11747)** Amazon DocumentDB now supports dual-stack mode (IPv4/IPv6) when your cluster is in a virtual private cloud (VPC). For more information, see this blog post and Amazon VPC and Amazon DocumentDB in the *Amazon DocumentDB Developer Guide*.

Date	Release notes
July 29, 2025	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.11747)** • Index bloat and collection based bloat metrics launched • Added CloudWatch metrics `AvailableMVCCIds` and `LongestActiveGCRuntime` that provide information into the overall health of the cluster with respect to garbage collection and aging collections. • Support for the following String operators: `$trim`, `$rtrim`, and `$ltrim` • Support for the following Stage operator: `$collStats` **Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.11747)** Fixed an issue during index creation that affected large collections.
January 15, 2025	**Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.11096)** • Fixed `killOp` command to handle a special case of bulk inserts. • Improved network I/O-related memory usage on Amazon DocumentDB instances.
November 6, 2024	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10980)** • Added support for minimum TLS versions `tls1.2+` and `tls1.3+` to `tls` cluster parameter.

Date	Release notes
September 17, 2024	**Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10898)** • `$setOnInsert` now supports the `_id` field during inserts resulting from update operations. • Fixed issue that prevents reclaiming storage space after a major version upgrade to Amazon DocumentDB 5.0.
August 22, 2024	**New feature** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.5704)** Amazon DocumentDB global clusters now support cluster switchovers and managed cluster failovers. For more information, see Performing a switchover for an Amazon DocumentDB global cluster and Performing a managed failover for an Amazon DocumentDB global cluster. **Note** Global cluster switchovers and failovers are supported on Amazon DocumentDB 4.0 and 5.0 only.

Date	Release notes
July 23, 2024	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10709)** • Added support for longer index names (up to 255 characters). For more information, see Naming constraints • Increased the maximum connection limit by two-fold:

Instance type	Original limit	New limit
t3.medium	500	1000
t4g.medium	500	1000
r5.large	1700	3400
r5.xlarge	3500	7000
r5.2xlarge	7100	14200
r5.4xlarge	14200	28400
r5.8xlarge	28400	60000
r5.12xlarge	30000	60000
r5.16xlarge	30000	60000
r5.24xlarge	30000	60000
r6g.large	1700	3400
r6g.xlarge	3500	7000
r6g.2xlarge	7100	14200
r6g.4xlarge	14200	28400
r6g.8xlarge	28400	60000

<table>
<tr><th>Date</th><th>Release notes</th></tr>
<tr><td></td><td>
<table>
<tr><th>Instance type</th><th>Original limit</th><th>New limit</th></tr>
<tr><td>r6g.12xlarge</td><td>30000</td><td>60000</td></tr>
<tr><td>r6g.16xlarge</td><td>30000</td><td>60000</td></tr>
</table>
For more information, see Instance quotas.
</td></tr>
<tr><td>July 22, 2024</td><td>New features

Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10593)

• Added support DML Audit filtering. You can now setup filter conditions to filter out DML Audit logs based on their specific requirements instead of logging every DML query. For more information, see Filtering DML audit events.</td></tr>
<tr><td>July 9, 2024</td><td>New feature

For Amazon DocumentDB 4.0 (minimum engine patch version 2.0.10179), TLS CA certificate updates no longer require a system reboot. For more information, see Updating your Amazon DocumentDB TLS certificates.</td></tr>
<tr><td>May 29, 2024</td><td>New features

Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10593)

• Added support to ensure full precision in audit logs when addressing large integers. Audit logs now maintain the exact numeric representation for all numbers, preventing any loss of precision.</td></tr>
</table>

Date	Release notes
April 3, 2024	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.10392)** • Added support for `bypassDocumentValidation` for `$jsonSchema`. For more information about `bypassDocumentValidation`, see bypassDocumentValidation. • Added support of `$expr`. • Added support for Uncorrelated Joins in `$lookup`. • Added support to retain validation rules in `$out` aggregation stage.
January 10, 2024	**New features** **Amazon DocumentDB 4.0 (Engine Patch Versions 2.0.10124, 2.0.10179, 2.0.10221)** • Added a support for GC runtime on a collection within `currentOp` command. • Added support for `$jsonSchema` schema keywords `type`, `allOf`, `oneOf`, `anyOf`, `not`, `maxItems`, `minItems`, `maxProperties`, `minProperties`, `pattern`, `patternProperties`, `multipleOf`, `dependencies`, and `uniqueItems`. For more information see Using JSON schema validation. • Added support for arithmetic operators `$ceil`, `$floor`, `$ln`, `$log`, `$log10`, `$sqrt`, and `$exp`. For more information see Arithmetic operators. • Added support for the conditional expression operator `$switch`.
December 20, 2023	**Other changes** Enabled support for in-place major version upgrade in Amazon DocumentDB 4.0. For more information, see Amazon DocumentDB in-place major version upgrade.

Date	Release notes
November 6, 2023	**Amazon DocumentDB 4.0 (Engine Patch Versions 2.0.9876)** **New features** • Added support for `$jsonSchema` schema keywords `maxLength`, `minLength`, `maximum`, `minimum`, `exclusiveMaximum`, `exclusiveMinimum`, `items`, and `additionalItems`. Note that JSON schema validation is supported in instance-based clusters only. • Added support for `$convert` aggregation pipeline operator and its shorthand derived operators `$toBool`, `$toInt`, `$toLong`, `$toDouble`, `$toString`, `$toDecimal`, `$toObjectId`, and `$toDate`. • Added support for set expression operators `$setDifference`, `$anyElementTrue`, and `$allElementTrue`. **Bug fixes and other changes** Fixed issue where a change stream update from `-NaN` to `NaN` was not being displayed.
September 20, 2023	**New features** Added support for in-place major version upgrades in Amazon DocumentDB 4.0. For more information see Amazon DocumentDB in-place major version upgrade.
September 15, 2023	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.9686)** Added support for $jsonSchema schema validator in instance-based clusters only. For more information see Using JSON schema validation.

Date	Release notes
July 13, 2023	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.9259)** • Added support for parallel index builds. **Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.9259)** • Stopped adding internal monitoring queries to auditing logs.
April 4, 2023	**Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.8934)** • Fixed issue with DML auditing when it is enabled during an ongoing workload. • Fixed issue with DML auditing when aggregate commands with hint is passed a string value. • Fixed issue with `listCollections` command not working when users with readwriteanydatabase role having both authorize dCollections and nameOnly options set to true. • Fixed issue to properly parse numeric string in a field name. • Cancel long running cursors when they are impacting garbage collection. • User defined local variable names now support "_" (underscore) for projection operators such as `$let` and `$filter`.
February 27, 2023	**Bug fixes and other changes** **Amazon DocumentDB 4.0** Added support for AWS Lambda. For more information, see Using AWS Lambda with Change Streams.

Date	Release notes
August 9, 2022	**New features** **4.0** • Added support for Decimal128 data type. The Decimal128 is a BSON data type supported in all Regions where DocumentDB is available. For more information, see Data Types. • Added support for DML query auditing with Amazon CloudWatch Logs. Now **Amazon DocumentDB** can record Data Manipulation Language (DML) events and Data Definition Language (DDL) events to Amazon CloudWatch Logs. For more information, see this blog post. **Bug fixes and other changes** **4.0** • You can now change your own passwoprd with own password with `changeOwnPassword` privilege.
July 25, 2022	**New features** **Amazon DocumentDB 4.0** You can now create clusters faster with the ability to create clones that use the same DocumentDB cluster volume and have the same data as the original cluster. For details, see Managing Amazon DocumentDB Clusters.

Date	Release notes
June 27, 2022	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.7509)** Amazon DocumentDB dynamically resizes your database based on usage patterns. Adding more data increases the space up to 64 Tebibytes (TiB) and deleting data decreases allotted space.
April 7, 2022	**New features** **4.0 (Engine Patch Versions 2.0.6142 and 2.0.6948)** Amazon DocumentDB Performance Insights is now in preview. You can now store seven days of performance history in a rolling window at no additional cost. For more information, see Monitoring with Performance Insights.

January 24, 2022	**New features**
	
	4.0 (Engine Patch Version 2.0.5170)
	
	• Amazon DocumentDB now offers a free trial. For details, see Amazon DocumentDB free trial page.
	• You can now use enhanced features with Geospatial query, including the following APIs:
	  • `$geoWithin`
	  • `$geoIntersects`
	• Added support for the following MongoDB operators:
	  • `$mergeObjects`
	  • `$reduce`
	
	For more information, see the Querying Geospatial data with Amazon DocumentDB.

Date	Release notes

January 21, 2022	**New features**
	
	Amazon DocumentDB 4.0 (Engine Patch Version 2.0.5706)
	
	• Amazon DocumentDB Graviton2 (r6g.large, r6g.2xlarge, r6g.4xlarge, r6g.8xlarge, r6g.12xlarge, r6g.16xlarge and t4g.medium) instances are now supported.
	
	New features
	
	4.0 (Engine Patch Version 2.0.5706)
	
	• Added support for the following MongoDB APIs:
	  • `$reduce`
	  • `$mergeObjects`
	  • `$geoWithin`
	  • `$geoIntersects`

Date	Release notes

October 25, 2021	**New features**
	
	4.0 (Engine Patch Version 2.0.5704)
	
	• Added support for the following MongoDB APIs
	  • `$literal`
	  • `$map`
	  • `$$ROOT`
	• Support for GeoSpatial Query capabilities. See this blog post for more details
	• Support for access control with user-defined roles. See this blog post for more details
	• Amazon DocumentDB JDBC Driver to enable connectivity from BI tools such as Tableau and query tools such as SQL Workbench
	
	Bug fixes and other changes
	
	4.0 (Engine Patch Version 2.0.5704)
	
	• Bug fix for `$natural` to sort correctly when an explicit `.sort()` is present along with `$natural`
	• Bug fix for change stream to work with `$redact`
	• Bug fix for `$ifNull` to work with empty array
	• Bug fix for excessive resource consumption/server crash when a currently logged-in user is deleted or that user's privilege for an ongoing activity is revoked
	• Bug fix in `listDatabase` and `listCollection` privilege check
	• Bug Fix dedupe logic for multi-key elements

Date	Release notes
June 24, 2021	**New features** **4.0 (Engine Patch Version 2.0.3371)** • r5.8xlarge and r5.16xlarge instances are now supported. Learn more at the blog post Amazon DocumentDB Now Supports r5.8xlarge and r5.16xlarge Instances. • Global clusters are now supported to provide disaster recovery from Region-wide outages and enable low-latency global reads by allowing reads from the nearest Amazon DocumentDB cluster. Note that global clusters are not currently supported in the South America (São Paulo), Europe (Milan), China (Beijing), and China (Ningxia) Regions.

Date	Release notes
May 4, 2021	**New features** **4.0 (Engine Patch Version 2.0.3371)** • `renameCollection` • `$zip` • `$indexOfArray` • `$reverseArray` • `$natural` • `$hint` support for update • Index scan for `distinct` **Bug fixes and other changes** **4.0 (Engine Patch Version 2.0.3371)** • Reduced memory usage for `$in` queries • Fixed a memory leak in multikey indexes • Fixed the explain plan and profiler output for `$out` • Added a timeout for operations from internal monitoring system to improve reliability • Fixed a defect impacting the query predicates passed to multikey indexes
January 15, 2021	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.722)** • None

Date	Release notes
November 9, 2020	**New features** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.722)** • MongoDB 4.0 compatibility • ACID transactions • Support for `cluster(client.watch()` or `mongo.watch())` and the database level `(db.watch())` change streams • Ability to start or resume a change streams using `startAtOperationTime` • Extend your change stream retention period to 7 days (previously 24 hours) • AWS DMS target for Amazon DocumentDB 4.0 • CloudWatch metrics: `TransactionsOpen` , `TransactionsOpenMax` , `TransactionsAborted` , `TransactionsStarted` , and `TransactionsCommitted` • New fields for transactions in `currentOp` , `ServerStatus` , and `profiler`. • Ability to use an index with the `$lookup` aggregation stage • `find()` queries with projections can be served direction from an index (covered query) • Ability to use `hint()` with the `findAndModify` • Performance optimizations for `$addToSet` operator • Improvements to reduce overall index sizes. • New aggregation operators: `$ifNull`, `$replaceRoot` , `$setIsSubset` , `$setInstersection` , `$setUnion` , and `$setEquals` • With the `ListCollection` and `ListDatabase` commands, you can now optionally use the `authorizedCollections` and `authorizedDatabases` parameters to allow users to list the collections and databases that they have permission to access

Date	Release notes
	without requiring the `listCollections` and `listDatabase` roles, respectively • Users can also end their own cursors without requiring the `KillCursor` role • Comparing numeric types of subdocuments is now consistent with comparing numeric types of first-level documents. The behavior in Amazon DocumentDB 4.0 is now compatible with MongoDB. **Bug fixes and other changes** **Amazon DocumentDB 4.0 (Engine Patch Version 2.0.722)** • `$setOnInsert` no longer allow updates when using the positional operator $. The behavior in Amazon DocumentDB 4.0 is now compatible with MongoDB. • Fixed issue with `$createCollection` and set `autoIndexId` • Projection for nested documents • Changed default setting for working memory to scale with instance memory size • Garbage collection improvements • Lookup with empty key in path, behavior difference with mongo • Fixed `dateToString` bug in timezone behavior • Fixed `$push` (aggregation) to respect sort order • Fixed bug in `$currentOp` with aggregate • Fixed issue with `readPreference` on secondary • Fixed issue with validating `$createIndex` is the same database as the command was issued • Fixed inconsistent behavior for `minKey`, `maxKey` lookup fails • Fixed issue with `$size` operator not working with composite array • Fixed issue with the negation of `$in` with regex • Fixed issue with `$distinct` command run against a view

Date	Release notes
	• Fixed issue with aggregations and find commands sorting missing fields differently • Fixed `$eq` to regular expression not checking type • Fixed `$currentDate` bug in timestamp ordinal position behavior • Fixed millisecond granularity for `$currentDate`

Amazon DocumentDB 3.6

Release notes for Amazon DocumentDB engine version 3.6 (MongoDB 3.6 compatibility).

Date	Release notes
August 31, 2026	**New feature** Amazon DocumentDB now supports in-place major version upgrades (MVU) directly from engine version 3.6 to version 8.0. This removes the need for intermediate version upgrades and lets you upgrade your 3.6 clusters while preserving existing data, configurations, and cluster settings. This upgrade is available in all AWS Regions where these versions are currently supported. For more information, see Amazon DocumentDB in-place major version upgrade.
August 18, 2026	**Security and stability improvements** **Amazon DocumentDB 3.6 (Engine Version: 1.0.208702)** This engine patch provides the following improvements: • **Improved request processing** Amazon DocumentDB now validates incoming client requests more thoroughly along the connection and message-parsing path. This improves resilience and stability across a wider range of input patterns. • **Authentication handshake hardening** Amazon DocumentDB now validates the authentication exchange more thoroughly. This improves robustness when handling unexpected or malformed input.

Date	Release notes
	If you don't apply it sooner, Amazon DocumentDB automatically applies this patch during your next maintenance window 30 days after release.
April 27, 2026	**New features** **Amazon DocumentDB 3.6 (Engine Version: 1.0.208692)** This update provides an optional engine patch with the following improvements: 1. Added support for `startAtOperationTime` in change streams.
February 12, 2026	**Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Version: 1.0.208675)** 1. The `DatabaseConnectionsActiveMax` metric now tracks `Connections (active)` for instances. For more information, see Instance quotas. 2. Improved SASL authentication performance. 3. Bug fix for $indexStats command.
August 20, 2024	**New feature** For Amazon DocumentDB 3.6 (minimum engine patch version 1.0.208662), TLS CA certificate updates no longer require a system reboot. For more information, see Updating your Amazon DocumentDB TLS certificates.

Date	Release notes
July 22, 2024	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208662)** Removed index limitations on db.r5.* and db.r6.* instances in Amazon DocumentDB MVU. For more information, see Prerequisites. **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208662)** Amazon DocumentDB now recognizes -NaN as a valid JSON token.
December 20, 2023	**Other changes** Enabled support for in-place major version upgrade in Amazon DocumentDB 3.6. For more information, see Amazon DocumentDB in-place major version upgrade.
November 17, 2023	**Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208570)** User defined local variable names now support "_" (underscore) for projection operators such as `$let` and `$filter`.
September 20, 2023	**New features** Added support for in-place major version upgrades in Amazon DocumentDB 3.6. For more information see Amazon DocumentDB in-place major version upgrade.

Date	Release notes
February 2, 2023	**Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208432)** • Fixed issue with DML auditing when it is enabled during an ongoing workload. • Fixed issue with DML auditing when aggregate commands with hint is passed a string value. • Fixed issue with `listCollections` command not working when users with readwriteanydatabase role having both authorize dCollections and nameOnly options set to true. • Fixed issue to properly parse numeric string in a field name. • Cancel long running cursors when they are impacting garbage collection.

Date	Release notes
August 9, 2022	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208152)** • Added support for Decimal128 data type. The Decimal128 is a BSON data type supported in all Regions where DocumentDB is available. For more information, see Data Types. • Added support for DML query auditing with Amazon CloudWatch Logs. Now **Amazon DocumentDB** can record Data Manipulation Language (DML) events and Data Definition Language (DDL) events to Amazon CloudWatch Logs. For more information, see this blog post. **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.208152)** • You can now change your own passwoprd with own password with `changeOwnPassword` privilege.
April 7, 2022	**New features** **Amazon DocumentDB 3.6 (Engine Patch Versions 1.0.207836 and 1.0.208015)** Amazon DocumentDB Performance Insights is now in preview. You can now store seven days of performance history in a rolling window at no additional cost. For more information, see Monitoring with Performance Insights.

Date	Release notes

January 24, 2022	**New features**
	
	Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207684)
	
	• Amazon DocumentDB now offers a free trial. For details, see Amazon DocumentDB free trial page.
	• You can now use enhanced features with Geospatial query, including the following APIs:
	  • `$geoWithin`
	  • `$geoIntersects`
	• Added support for the following MongoDB operators:
	  • `$mergeObjects`
	  • `$reduce`
	
	For more information, see the Querying Geospatial data with Amazon DocumentDB.

January 21, 2022	**New features**
	
	Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207781)
	
	• Added support for the following MongoDB APIs:
	  • `$reduce`
	  • `$mergeObjects`
	  • `$geoWithin`
	  • `$geoIntersects`

Date	Release notes

October 25, 2021	**New features**
	
	Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207780)
	
	• Added support for the following MongoDB APIs
	  • `$literal`
	  • `$map`
	  • `$$ROOT`
	• Support for GeoSpatial Query capabilities. See this blog post for more details
	• Support for access control with user-defined roles. See this blog post for more details
	• Amazon DocumentDB JDBC Driver to enable connectivity from BI tools such as Tableau and query tools such as SQL Workbench
	
	Bug fixes and other changes
	
	Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207780)
	
	• Bug fix for `$natural` to sort correctly when an explicit `.sort()` is present along with `$natural`
	• Bug fix for change stream to work with `$redact`
	• Bug fix for `$ifNull` to work with empty array
	• Bug fix for excessive resource consumption/server crash when a currently logged-in user is deleted or that user's privilege for an ongoing activity is revoked
	• Bug fix in `listDatabase` and `listCollection` privilege check
	• Bug Fix dedupe logic for multi-key elements

Date	Release notes
June 24, 2021	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207117)** • r5.8xlarge and r5.16xlarge instances are now supported. Learn more at the blog post Amazon DocumentDB Now Supports r5.8xlarge and r5.16xlarge Instances. • Global clusters are now supported to provide disaster recovery from Region-wide outages and enable low-latency global reads by allowing reads from the nearest Amazon DocumentDB cluster. Note that global clusters are not currently supported in the South America (São Paulo), Europe (Milan), China (Beijing), and China (Ningxia) Regions.

Date	Release notes
May 4, 2021	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207117)** • `renameCollection` • `$zip` • `$indexOfArray` • `$reverseArray` • `$natural` • `$hint` support for update • Index scan for `distinct` **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.207117)** • Reduced memory usage for `$in` queries • Fixed a memory leak in multikey indexes • Fixed the explain plan and profiler output for `$out` • Added a timeout for operations from internal monitoring system to improve reliability • Fixed a defect impacting the query predicates passed to multikey indexes

Date	Release notes
January 15, 2021	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • Ability to use an index with the `$lookup` aggregation stage • `find()` queries with projections can be served direction from an index (covered query) • Ability to use `hint()` with the `findAndModify` • Performance optimizations for `$addToSet` operator • Improvements to reduce overall index sizes • New aggregation operators: `$ifNull`, `$replaceRoot`, `$setIsSubset`, `$setInstersection`, `$setUnion`, and `$setEquals` • Users can also end their own cursors without requiring the `KillCursor` role
November 9, 2020	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • None **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • None

Date	Release notes
October 30, 2020	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • Added the ability to open a change stream cursor at the cluster level (`client.watch()` or `mongo.watch()`) and the database (`db.watch()`) • Ability to increase the change stream retention period to 7 days(previously 24 hours) **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • Various general case performance improvements • A targeted security improvement • Fixed an issue with skip sort on second field of a compound index • Enable regular index for equality on single field of a multi-key index (not compound) • Fixed authentication race condition • Fixed issue that caused an infrequent garbage collection crash • RBAC security improvement • Added `databaseConnectionsMax` metric • Performance improvements for certain workloads on `r5.24xlarge` instances

Date	Release notes
September 22, 2020	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • `$out` aggregation stage • Increased the maximum number of connections and cursor per instance by as much as 10x **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • None
July 10, 2020	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • Cross Region Snapshot Copy **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • None

Date	Release notes
June 30, 2020	**New features** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • T3 medium instances **Bug fixes and other changes** **Amazon DocumentDB 3.6 (Engine Patch Version 1.0.206295)** • Idle memory reclamation for t3 instances • Authentication improvements • Improved SASL authentication performance • Fixed `currentOp` issue when exceeding maximum possible ops • Fixed `killOps` issue for bulk update and delete • Improvements to `$sample` performance with `$match` • Fixed support for $$ in cond case in redact stage • Fixed various recurring crash root causes • Improvements to TTL sweeping to reduce IOs and latency • Optimized memory utilization for `$unwind` • Fixed collection stats race condition with drop index • Fixed race condition during concurrent index build • Fixed infrequent crash in `hash_search` in index

General Amazon DocumentDB release notes

General service updates including Region availability, tooling integrations, and cross-version features.

Date	Release notes
November 6, 2026	**New feature**

Date	Release notes
	Amazon DocumentDB serverless now supports platform version 4, which delivers up to 35% higher throughput compared to platform version 3. All new serverless clusters launch on the latest platform version available in your AWS Region. You can upgrade existing clusters on platform version 3 to platform version 4 by applying the `serverless-platform-version-update` pending maintenance action. You can view the platform version of a cluster in the AWS Management Console or through the `ServerlessV2PlatformVersion` field of the `DescribeDBClusters` API. For more information, see Amazon DocumentDB serverless scaling capacity and Checking and upgrading the platform version of a serverless cluster.
November 4, 2026	**New feature** Amazon DocumentDB is now available in the Asia Pacific (Taipei) Region. For more information, see this blog post. Amazon DocumentDB is now available in the Asia Pacific (New Zealand) Region. For more information, see this blog post.
November 3, 2026	**New feature** Amazon DocumentDB added support for an additional Availability Zone in the Europe (London) Region.

Date	Release notes
September 15, 2026	**New feature** The Amazon DocumentDB serverless instance type is now available on Amazon DocumentDB 5.0 and 8.0 engine versions in the following Regions: • Asia Pacific (Osaka) • Asia Pacific (Jakarta) • Asia Pacific (Malaysia) • Asia Pacific (Melbourne) • Asia Pacific (Thailand) • Canada West (Calgary) • Europe (Zurich) • Europe (Stockholm) • Israel (Tel Aviv) • Mexico (Central) For more information, see Supported instance classes by Region and Using Amazon DocumentDB serverless.
April 29, 2026	**New feature** Amazon DocumentDB is now available in the Canada West (Calgary) Region. For more information, see this blog post.
April 17, 2026	**New feature** Faster scaling on DocumentDB serverless is now available.

Date	Release notes
February 11, 2026	**New feature** Amazon DocumentDB is now available in the Asia Pacific (Melbourne) Region. For more information, see this blog post. Amazon DocumentDB is now available in the Europe (Zurich) Region. For more information, see this blog post.
January 08, 2026	**New feature** Amazon DocumentDB is now available in the Asia Pacific (Jakarta) Region. For more information, see this blog post.
October 13, 2025	**New feature** Amazon DocumentDB now supports R6G and T4G instances in the Asia Pacific (Hyderabad) Region. For more information, see Supported instance classes by Region.
October 7, 2025	**New feature** Amazon DocumentDB is now available in the following Regions: • Asia Pacific (Malaysia) • Asia Pacific (Osaka) • Asia Pacific (Thailand) • Mexico (Central) For more information, see this blog post.

Date	Release notes
September 26, 2025	**New features** Amazon DocumentDB now supports cross-Region and cross-account snapshot copy and snapshot sharing. For more information, see Copying snapshots across AWS Regions and Sharing Amazon DocumentDB cluster snapshots.
June 18, 2025	**New feature** Amazon DocumentDB is now available in the Israel (Tel Aviv) Region.
May 8, 2025	**New feature** Amazon DocumentDB is now available in the Europe (Stockholm) Region. For more information, see this blog post.
April 2, 2025	**Bug fixes and other changes** Fixed a bug in vector index creation with background indexes.
February 6, 2025	**New feature** Amazon DocumentDB is now integrated with AWS Toolkit for Visual Studio Code. For more information, see this What's New blog post, and see Working with Amazon DocumentDB in the Toolkit in the *AWS Toolkit for Visual Studio Code User Guide*.
October 22, 2024	**New feature** Amazon DocumentDB now supports storage network throughput metrics `StorageNetworkReceiveThroughput`, `StorageNetworkTransmitThroughput`, and `StorageNetworkThroughput`. For more information, see Evaluating Amazon DocumentDB instance usage with CloudWatch metrics.

Date	Release notes
September 18, 2024	**New feature** Amazon DocumentDB is now available in the Africa (Cape Town) Region. For more information, see this blog post. Amazon DocumentDB is now available in the Europe (Spain) Region. For more information, see this blog post.
June 25, 2024	**New feature** Authentication with AWS IAM ARNs is available in Amazon DocumentDB instance-based 5.0 clusters across all supported Regions. For more information, see Authentication using IAM identity.
April 3, 2024	**New feature** Amazon DocumentDB is now available in the Middle East (UAE) Region. For more information, see this blog post. **Bug fixes and other changes** • Fixed an error when invoking `db.coll.stats()` on mongo shell version 1.7 and later. • Fixed a memory leak issue for change stream queries that contain `$regex` as a part of the same aggregation pipeline.
January 30, 2024	**New features** **Amazon DocumentDB global clusters** Global clusters are now available in both AWS GovCloud (US) regions: AWS GovCloud (US-East) and AWS GovCloud (US-West).
December 13, 2023	**New features** Added support for 1-click EC2 connectivity. For more information, see Connect using Amazon EC2.

Date	Release notes
November 17, 2023	**New features** Amazon DocumentDB is now available in the AWS GovCloud (US-East) Region. For more information, see this blog post.
September 25, 2023	**New features** Amazon DocumentDB is now available in the Asia Pacific (Hong Kong) Region. For more information, see this blog post.
September 11, 2023	**New features** Amazon DocumentDB is now available in the Asia Pacific (Hyderabad) Region. For more information, see this blog post.
April 29, 2022	**New features** Amazon DocumentDB is now available in the China (Beijing) Region. For more information, see this blog post.
March 16, 2022	**New features** Amazon DocumentDB is now available in the Europe (Milan) Region. For more information, see this blog post.
February 8, 2022	**New features** Amazon DocumentDB R6G and T4G instances are now available in Asia Pacific, South America, and Europe. For more information, see this blog post.
May 4, 2021	**New features** See all the new features in this blog post.
November 9, 2020	**New features** See all the new features in this blog post.

Date	Release notes
October 30, 2020	**New features** See all the new features in this blog post.
September 22, 2020	**New features** See all the new features in this blog post.
July 10, 2020	**New features** See all the new features in this blog post.
June 30, 2020	**New features** See all the new features in this blog post.

Amazon DocumentDB elastic clusters release notes

Release notes for Amazon DocumentDB elastic clusters.

Date	Release notes
January 28, 2025	**New feature** For one-click connectivity, Amazon DocumentDB is now integrated with AWS CloudShell for instance-based and elastic clusters. For more information see any or all of the following: • Get started with Amazon DocumentDB • Get started with Amazon DocumentDB elastic clusters • What's new blog post • Technical how-to blog post
November 12, 2024	**New features** Amazon DocumentDB elastic clusters now support Background Indexes.

Date	Release notes
November 1, 2024	**New feature** Amazon DocumentDB now supports elastic cluster maintenance actions. For more information, see Maintaining Amazon DocumentDB elastic clusters.
August 8, 2024	**New feature** Amazon DocumentDB elastic clusters are now available in the Asia Pacific (Hong Kong), Canada (Central), and Europe (Paris) Regions. For more information, see Region availability for elastic clusters.
July 8, 2024	**New feature** Amazon DocumentDB elastic clusters are now available in the Europe (Milan) Region. For more information, see Region availability for elastic clusters.
February 22, 2024	**New features** **Amazon DocumentDB elastic clusters** Amazon DocumentDB elastic clusters now support the following features: • Readable secondary shard instance replicas - for more information, see step 5b of Step 1: Create an elastic cluster. • Start/stop cluster - for more information, see Stopping and starting an Amazon DocumentDB elastic cluster. • Configurable shard instances - for more information, see step 5b of Step 1: Create an elastic cluster. • Automatic backups for snapshots - for more information, see Managing an elastic cluster snapshot automatic backup. • Copy snapshot - for more information, see Copying an elastic cluster snapshot.

Date	Release notes
January 30, 2024	**New features** **Amazon DocumentDB elastic clusters** Amazon DocumentDB elastic clusters are now available in the following Regions: • Asia Pacific (Mumbai) • Asia Pacific (Seoul) • South America (São Paulo) • Europe (London) For more information, see Elastic cluster Region and version availability.

August 3, 2023	**New features**
	
	Amazon DocumentDB Elastic clusters
	
	• Amazon DocumentDB Elastic clusters now support the following operations:
	  • `top`
	  • `collStats`
	  • `hint`
	  • `dataSize`
	
	  See Supported MongoDB APIs, operations, and data types in Amazon DocumentDB for the complete list of supported commands and operations.
	• Time to Live (TTL) indexes are now supported.
	• Index `hints` are now supported with index expressions.

Date	Release notes
March 22, 2023	**New features** Amazon DocumentDB elastic clusters are now available in the Asia Pacific (Singapore), Asia Pacific (Sydney), and Asia Pacific (Tokyo) Regions. For more information, see Elastic cluster Region and version availability.
November 30, 2022	**New features** **Amazon DocumentDB Elastic clusters** Amazon DocumentDB elastic clusters is a new type of Amazon DocumentDB cluster that enables users to leverage the MongoDB sharding APIs to scale out their cluster. Elastic clusters handle virtually any number of reads and writes with petabytes of storage capacity by distributing the data and compute across multiple underlying compute instances and volumes. To learn more, see Using Amazon DocumentDB elastic clusters.

Document history for the Amazon DocumentDB Developer Guide

- **API version:** 2014-10-31
- **Latest documentation update:** May 27, 2026

The following table describes the documentation for this release of the *Amazon DocumentDB Developer Guide*.

Change	Description	Date
New Amazon DocumentDB Developer Guide	The Amazon DocumentDB Developer Guide is now published at a new URL. The API Reference is available separately at Amazon DocumentDB API Reference.	May 27, 2026
AWS managed policy update - policy change	Amazon DocumentDB updates the full-access policies for elastic clusters.	February 11, 2025
AWS managed policy update - policy change	Amazon DocumentDB updates the full-access policies for elastic clusters.	February 21, 2024
AWS managed policy update - policy change	Amazon DocumentDB updates the read-only and full-access policies for elastic clusters.	June 21, 2023
AWS managed policy update - new policy	Amazon DocumentDB introduces a new read-only policy for elastic clusters.	June 8, 2023

AWS managed policy update - new policy	Amazon DocumentDB introduces a new full-access policy for elastic clusters.	June 5, 2023
MongoDB 5.0 Compatibility	Amazon DocumentDB is now compatible with version 5.0 of MongoDB.	March 1, 2023
Policy update	To support the Amazon DocumentDB elastic cluster feature, AmazonDocDBConsoleFullAccess policy is updated, and the AmazonDocDB-ElasticServiceRolePolicy is introduced.	November 30, 2022
Elastic clusters	Added new Elastic Cluster feature supporting hash-based partitioning (sharding) of data across Amazon DocumentDB's distributed storage system.	November 30, 2022
Global clusters	Added documentation on how to use Global Clusters.	June 2, 2021
Event subscriptions	Added event subscription documentation.	March 26, 2021
Version 3.6 Upgrades	Documented improvements to version 3.6 in role-based access controls, aggregation operators, and performance.	January 15, 2021
MongoDB 4.0 Compatibility	Amazon DocumentDB is now compatible with version 4.0 of MongoDB.	November 9, 2020

Get started guides	New Get Started Guides for getting started with Amazon DocumentDB using AWS Cloud9, Amazon EC2, Robo3T or Studio3T.	August 15, 2020
Additional Availability Zones supported	Amazon DocumentDB added support for an additiona l Availability Zone in Asia Pacific (Seoul) (ap-north east-2).	July 14, 2020
Added support for copying snapshots across Regions.	Amazon DocumentDB added support for copying cluster snapshots across AWS Regions. For more informati on, see Copying Snapshots Across Regions.	July 10, 2020
Added support for T3 instance class.	Added support for T3 instance types in all Regions supporting Amazon DocumentDB. For more information, see Supported Instance Classes by Region and Instance Class Specifica tions.	June 30, 2020
Added support for AWS GovCloud (US).	Amazon DocumentDB is now available in the AWS GovCloud (US) Region (us-gov-west-1).	June 29, 2020

Added 16 new CloudWatch metrics.	Amazon DocumentDB added support for 16 new Amazon CloudWatch metrics. For more information, see Monitoring Amazon DocumentDB with CloudWatch.	June 23, 2020
Added support for null characters and $regex operator.	Amazon DocumentDB added support for null characters in strings and the ability to use an index for $regex. To view the supported MongoDB APIs and aggregation pipeline capabilities for Amazon DocumentDB, see Functional Differences with MongoDB.	June 22, 2020
Added support for improved multi-key indexing capabilities.	Amazon DocumentDB added support for improved multi-key indexing capabilities that include indexing of arrays larger than 2,048 bytes and the ability to create a compound multi-key index with multiple keys in the same array. For more information, see Functional Differences with MongoDB.	April 23, 2020
Added support for deletion protection for an Amazon DocumentDB CloudFormation stack.	Amazon DocumentDB added support for enabling deletion protection when creating an Amazon DocumentDB CloudFormation stack.	April 20, 2020

Added support for role-based access control.	Amazon DocumentDB added support for role-based access control using built-in roles.	March 26, 2020
Added support for an additional Availability Zone in Canada (Central) (ca-central-1).	Amazon DocumentDB is now available in the Canada (Central) Region (ca-central-1) with R5 class instances and 3 Availability Zones.	March 26, 2020
Added support for two additional MongoDB APIs.	Amazon DocumentDB added support for `$dateFromString` and `executionStats` MongoDB APIs.	March 23, 2020
Added support for five additional MongoDB APIs.	Amazon DocumentDB added support for `$objectToArray` , `$arrayToObject` , `$slice`, `$mod`, and `$range` MongoDB APIs.	February 6, 2020
Added support for Canada (Central).	Amazon DocumentDB is now available in the Canada (Central) Region (ca-central-1) with R5 class instances.	December 11, 2019
Added support for ChangeStreamLogSize.	Amazon DocumentDB added support for `ChangeStreamLogSize` for Cloudwatch metrics.	November 22, 2019
Added support for Europe (Paris) Region	Amazon DocumentDB is now available in the Europe (Paris) Region (eu-west-3) with R5 class instances.	October 30, 2019

Added support for Asia Pacific (Mumbai) Region	Amazon DocumentDB is now available in the Asia Pacific (Mumbai) Region (ap-south-1) with R5 class instances.	October 17, 2019
Added support for three additional MongoDB APIs	Amazon DocumentDB added support for the `$addFields`, `$concatArrays`, and `$lookup` MongoDB APIs.	October 16, 2019
Added support for Asia Pacific (Singapore) Region	Amazon DocumentDB is now available in the Asia Pacific (Singapore) Region (ap-southeast-1) with R5 class instances.	October 14, 2019
Added new document for updating TLS certificates	Added instructions for updating CA certificates to use the new CA certificate to create TLS connections.	October 2, 2019
Added API support for certificates	Amazon DocumentDB a new Certificate data type for instances. For more information, see DBInstance.	October 1, 2019
Support for query profiling	Amazon DocumentDB added the ability to profile supported operations on your cluster's instances and databases.	August 19, 2019
Added third AZ in Asia Pacific (Tokyo)	Amazon DocumentDB added a third Availability Zone (AZ) for your compute instances in Asia Pacific (Tokyo).	August 9, 2019

Support for additional Mongo APIs	Added support for additional aggregation pipeline capabilities that include the `$in`, `$isoWeek`, `$isoWeekYear` , `$isoDayOfWeek` , and `$dateToString` aggregation operators and the `$addToSet` aggregation stage. Amazon DocumentDB also added support for the `top()` command for collection level diagnostics and the ability to modify the `expireAfterSeconds` parameter for TTL indexes using the `collMod()` command.	July 31, 2019
Added support for Europe (London)	Amazon DocumentDB is now available in Europe (London) (eu-west-2) with R5 class instances.	July 18, 2019
Added code samples	Added code examples in R and Ruby for programmatically connecting to Amazon DocumentDB.	July 17, 2019
Added best practice	Added a Best Practice to help you manage your Amazon DocumentDB costs.	July 17, 2019
Support for stopping and starting a cluster	Amazon DocumentDB added support for stopping and starting clusters to manage costs for development and test environments.	July 1, 2019

Support for cluster deletion protection	To protect your clusters from accidental deletion, Amazon DocumentDB added deletion protection. For more information, see the following topics: Creating an Amazon DocumentDB Cluster, Modifying an Amazon DocumentDB Cluster, Deleting an Amazon DocumentDB Cluster, and `DeletionProtection` in the API topic DBCluster.	July 1, 2019
Functional differences update	Added Implicit Transactions to Functional Differences.	June 26, 2019
Functional differences addition	Added note regarding storage and index compression in Amazon DocumentDB.	June 13, 2019
Additional Region supported	Amazon DocumentDB is now available in Asia Pacific (Sydney) (ap-southeast-2) with R5 class instances.	June 5, 2019
R5 instance class supported in additional Regions	Added R5 instance class support for 4 additional Regions: US East (Ohio), US East (N. Virginia), US West (Oregon), and EU (Ireland). With this change, R5 instances are supported in all Regions supporting Amazon DocumentDB.	May 17, 2019

Additional Regions supported	Added support for 2 additional Regions, Asia Pacific (Tokyo) (ap-northeast-1) and Asia Pacific (Seoul) (ap-northeast-2) with R5 instance classes. For more information, see Supported Instance Classes by Region and Instance Class Specifications.	May 8, 2019
Added more connection code examples	Added code examples in Java and C# for connecting to Amazon DocumentDB.	April 24, 2019
Additional Mongo API support	Added support for seven aggregation string operators (`$indexOfBytes`, `$indexOfCP`, `$strLenBytes`, `$strLenCP`, `$toLower`, `$toUpper`, and `$split`), nine date-time operators (`$dayOfYear`, `$dayOfMonth`, `$dayOfWeek`, `$year`, `$month`, `$hour`, `$minute`, `$second`, and `$millisecond`), and the `$sample` aggregation pipeline stage.	April 4, 2019
Added connection code examples	Added code examples in Python, Node.js, PHP, and Go for connecting to Amazon DocumentDB.	March 21, 2019

Support for Frankfurt Region and R5 instances	Added support for Europe (Frankfurt) Region (eu-central-1) with R5 instance classes. For more information, see Supported Instance Classes by Region and Instance Class Specifications.	March 13, 2019
Aggregation pipeline operators support	Added support for new aggregation string operators (`$concat`, `$substr`, `$substrBytes` , `$substrCP` , `$strcasecmp`), an array aggregation operator (`$size`), an aggregation group accumulator operator (`$push`), and aggregation stages (`$redact` and `$indexStats`). We also added support for positional array operators (`$[]` and `$[<identifier>]`) and `hint()`.	February 28, 2019
Engine upgrades	Added documentation for determining pending cluster modifications and upgrading your cluster's engine version.	February 15, 2019
Auditing events	Added support for auditing database events with Amazon CloudWatch Logs.	February 12, 2019
Quick Start	Added a Quick Start topic to help you easily start with Amazon DocumentDB using CloudFormation.	January 11, 2019

Public release	This is the initial public release of Amazon DocumentDB (with MongoDB compatibility). This release includes the Developer Guide and the integrated Resource Management API Reference.	January 9, 2019