Document Database Service

Developer Guide

Issue 01

Date 2022-08-30

HUAWEI

HUAWEI TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Contents

1 Database Usage Suggestions

1.1 Basic Commands

1.1.1 Query

To avoid slow queries, analyze and optimize the execution process (query plan).

db.collection.find().explain()

For details, see **Performance** and **official documents**.

Precaution

- The returned query result is a cursor. After using the cursor, close it in time. Otherwise, memory leaks accumulate.
- Create necessary indexes based on search criteria. For details about index design, see **Index**.
  - Do not use COLLSCAN to scan the entire table.
  - Query conditions and index fields are matched in sequence.
- For cluster instances, shard tables properly based on service requirements. For details about sharding, see **Sharding**.
- For sharded tables, query conditions must be based on shard keys to avoid unnecessary queries such as scatter-gather. For details, see **Distributed Queries**.
- You can specify a readConcern level for queries. For details, see **Read Concern**.
- You can specify the **readPerference** parameter for queries. For details, see **Read Preference**.

1.1.2 Write/Update

- Operations such as write, update, delete,and index insert and delete are actually converted to write operations in the background. The underlying storage engines (WiredTiger and RocksDB) use the appendOnly mechanism. Only when the internal data status of the storage engine meets certain

conditions, the internal compaction operation is triggered to compress data and release disk space. That is why sometimes the disk usage seems greater than the actual data volume, but your services are not affected.

- The write/update operation involves synchronization to the standby node. You can specify the writeConcern level for the operation. For details, see **Write Concern**.

Precaution

- update and upsert: If you use upsert, the data you want to insert will be queried first. If the data does not exist, it will be inserted. Otherwise, an update operation is performed. In the preceding scenario, the update or insert command is recommended.
- The update operation also needs to match the index.
  - Do not use COLLSCAN to scan the entire table.
  - Query conditions and index fields are matched in sequence.
- The size of the document involved in the insert or update command cannot exceed 16 MB.

1.1.3 Delete

- The delete operation is classified into logical delete (condition-based remove) and quick delete (dropCollection, dropDatabase).
- After a large number of delete operations are performed, the actual data volume may not change, but the storage space usage increases. This is because operations such as write, update, delete, and index insert and delete are actually converted to write in the background. The underlying storage engines (WiredTiger and RocksDB) use appendOnly. Only when the internal data status of the storage engine meets certain conditions, the compaction operation is triggered to compress data and release storage space.
- If the entire database is not required, you can run **dropDatabase** to delete it instead of logically deleting it to quickly release disk space.
- The delete operation involves synchronization to the standby node. You can specify the writeConcern level for the delete operation. For details, see **Write Concern**.

Precaution

- Avoid mis-deletion. The delete command cannot be undone. Before the deletion, run **db** to check whether the current database is opened.
- If data is deleted by mistake, restore the data:
  a. **Back up** and **restore** data based on historical backup files.
  b. If a backup instance is available, you can restore the deleted data by referring to **Migrating data using the export and import tools**. If new data is written during the restoration, restoring data may affect data consistency.
  c. Data generated from the last backup time to the time when the instance is deleted by mistake cannot be restored.
- If the delete command is executed successfully, the deletion is successful. If the delete command fails, some data may have been deleted. Do not use the

deleted database table. You are advised to continue to run the delete command until the deletion is successful.

1.2 Development Rules

Database Connections

If the maximum number of mongod or mongos connections is reached, your client cannot connect to the DDS instances. Each connection received by mongod or mongos is processed by a single thread of 1 MB stack space. As the connections increase, too many threads will increase the context switching overhead and memory usage.

- If you connect to databases from clients, calculate the number of clients and the size of the connection pool configured for each client. The total number of connections cannot exceed 80% of the maximum number of connections allowed by the current instance.
- The connection between the client and the database must be stable. It is recommended that the number of new connections per second be less than 10.
- You are advised to set the connection timeout interval of the client to at least three times the maximum service execution duration.
- For a replica set, the IP addresses of both the primary and standby nodes must be configured on the client. For a cluster, at least two mongos IP addresses must be configured.
- DDS uses user **rwuser** by default. When you log in as user **rwuser**, the authentication database must be **admin**.

Reliability

Rules for setting write concern: For mission-critical services, set write concern to {w:n},n>0. A larger value is better consistency but poorer performance.

- **w:1** means that a confirmation message was returned after data was written to the primary node.
- **w:1,journal:true** means that the result was returned after data was written to the primary node and logs.
- **w:majority** means that the result was returned after data was written to more than half of the total standby nodes.

  NOTE

  If data is not written using **w:majority**, the data that is not synchronized to the standby node may be lost when a primary/standby switchover occurs.

If high reliability is required, deploy a cluster in three AZs.

Performance

Specification

- The service program is not allowed to perform full table scanning.

- During the query, select only the fields that need to be returned. In this way, the network and thread processing loads are reduced. If you need to modify data, modify only the fields that need to be modified. Do not directly modify the entire object.
- Do not use $not. DDS does not index missing data. The $not query requires that all records be scanned in a single result collection. If $not is the only query condition, a full table scan will be performed on the collection.
- If you use $and, put the conditions with the fewest matches before other conditions. If you use $or, put the conditions with the more matches first.
- In a single instance, the total number of databases cannot exceed 200, and the total number of collections cannot exceed 500.
- Before bringing a service online, perform a load test to measure the performance of the database in peak hours.
- Do not execute a large number of concurrent transactions at the same time or leave a transaction uncommitted for a long time.
- Before rolling out services, check the performance of all query types through the execution of query plans.

Suggestion

- Each connection is processed by an independent thread in the background. Each thread is allocated with 1 MB stack memory. The number of connections should not be too large. Otherwise, too much memory is occupied.
- Use the connection pool to avoid frequent connection and disconnection. Otherwise, the CPU usage is too high.
- Reduce disk read and write operations: Reduce unnecessary upsert operations.
- Optimize data distribution: Data is sharded and hot data is distributed evenly between shards.
- Reduce lock conflicts: Do not perform operations on the same key too frequently.
- Reduce lock wait time: Do not create indexes on the frontend.

Notice

During the development process, each execution on a collection must be checked using explain() to view its execution plan. Example:

db.T_DeviceData.find({"deviceId":"ae4b5769-896f"}).explain();

db.T_DeviceData.find({"deviceId":"77557c2-31b4"}).explain("executionStats");

A covered query does not have to read a document and returns a result from an index, so using a covered query can greatly improve query efficiency. If the output of explain() shows that indexOnly is true, the query is covered by an index.

Execution plan parsing:

1. Check the execution time. The smaller the values of the following parameters, the better the performance:
   executionStats.executionStages.executionTimeMillisEstimate and **executionStats.executionStages.inputStage. executionTimeMillisEstimate**
   - **executionStats.executionTimeMillis** specifies how much time the database took to both select and execute the winning plan.

  - **executionStats.executionStages.executionTimeMillisEstimate** is the execution completion time of the winning plan.
  - **executionStats.executionStages.inputStage.executionTimeMillisEstimate** is the execution completion time of the child stage of the winning plan.

2. Check the number of scanned records. If the three items are the same, the index is best used.
  - **executionStats. nReturned** is the number of documents that match the query condition.
  - **executionStats .totalKeysExamined** indicates the number of scanned index entries.
  - **executionStats .totalDocsExamined** indicates the number of scanned document entries.
3. Check the stage status. The following combinations of stages can provide good performance.
  - Fetch+IDHACK
  - Fetch+ixscan
  - Limit+ (Fetch+ixscan)
  - PROJECTION+ixscan

Table 1-1 Status description

Status Name	Description
COLLSCAN	Full table scan
SORT	In-memory sorting
IDHACK	_id-based query
TEXT	Full-text index
COUNTSCAN	Number of unused indexes
FETCH	Index scanning
LIMIT	Using Limit to limit the number of returned records
SUBPLA	$or query stage without using an index
PROJECTION	Restricting the return of stage when a field is returned.
COUNT_SCAN	Number of used indexes

Cursor Usage Rules

If a cursor is inactive for 10 minutes, it will be automatically closed. You can also manually close it to save resources.

Rules for Using Distributed Transactions in Version 4.2

- Spring Data MongoDB does not support the retry mechanism after a transaction error is reported. If the client uses Spring Data MongoDB as the client to connect to MongoDB, you need to use Spring Retry to retry the transaction based on the reference file of Spring Data MongoDB. For details, see the **official document**.
- The size of the distributed transaction operation data cannot exceed 16 MB.

1.3 Design Rules

Naming

- The name of a database object (database name, table name, field name, or index name) has to start with a lowercase letter and must be followed by a letter or digit. The length of the name cannot exceed 32 bytes.
- The database name cannot contain special characters (""./\$/*?~#:?|") or null character (\0). The database name cannot be the system database name, such as admin, local, and config.
- The database collection name can only contain letters and underscores (_). The name cannot be prefixed with "system". The total length of <*Database name*>.<*Collection name*> cannot exceed 120 characters.

Index

You can use indexes to avoid full table scans and improve query performance.

- A column index can have up to 512 bytes, an index name can have up to 64 characters, and a composite index can have up to16 columns.
- The total length of <*Database name*>.<*Collection name*>.$<*Index name*> cannot exceed 128 characters.
- Create indexes for fields with high selectivity. If you create indexes for low selective fields, large result sets may be returned. This should be avoided.
- Write operations on a collection will trigger more I/O operations on indexes in the collection. Ensure that the number of indexes in a collection does not exceed 32.
- Do not create indexes that will not be used. Unused indexes loaded to the memory will cause a waste of memory. In addition, useless indexes generated due to changes in service logic must be deleted in a timely manner.
- Indexes must be created in the background instead of foreground.
- An index must be created for the sort key. If a composite index is created, the column sequence of the index must be the same as that of the sort key. Otherwise, the index will not be used.
- Do not create an index based on the leading-edge column of a composite index. If the leading-edge column of a composite index is the column used in another index, the smaller index can be removed. For example, a composite index based on "firstname" and "lastname" can be used for queries on "firstname". In this case, creating another firstname-based index is unnecessary.

Sharding

You can shard collections to maximize the cluster performance.

Suggestions for sharding collections:

- In scenarios where the data volume is large (more than one million rows) and the write/read ratio is high, sharding is recommended if the data volume increases with the service volume.
- If you shard a collection using a hashed shard key, pre-splitting the chunks of the sharded collection can help reduce the impact of automatic balancing and splitting on service running.
- If sharding is enabled for a non-empty collections, the time window for enabling the balancer must be set during off-peak hours. Otherwise conflicts may occur during data balancing between shards and service performance will be affected.
- If you want to perform a sort query based on the shard key and new data is evenly distributed based on the shard key, you can use ranged sharding. In other scenarios, you can use hashed sharding.
- Properly design shard keys to prevent a large amount of data from using the same shard key, which may lead to jumbo chunks.
- If a sharded cluster is used, you must run **flushRouterConfig** after running **dropDatabase**. For details, see **How Do I Prevent Mongos Cache Problem?**
- The update request of the service must match the shard key. When a sharded table is used, an error will be reported for the update request and "An upsert on a sharded collection must contain the shard key and have the simple collation" will be returned in the following scenarios:
  - The filter field of the update request does not contain the shard key field and the value of **multi** is **false**.
  - The set field does not contain the shard key and the value of **upsert** is **true**.

2 Database Usage

2.1 Connecting a Database

The following table lists the common methods for connecting to a DDS instance.

Table 2-1 Connection methods

Method	IP Address	Scenario	Description
Private network connection (recommended)	Private IP address	DDS provides a private IP address by default. • If your applications are deployed on ECS and located in the same region, AZ, and VPC subnet as the DB instance, you are advised to use a private IP address to connect to the DB instance. • By default, a DDS instance is not accessible from an ECS in a different security group. Add an inbound rule to the DDS security group to allow access of the ECS. • The default DDS port is 8635. You can change it if you want to access DDS through another port.	Secure and excellent performance

Method	IP Address	Scenario	Description
Public network connection	EIP	• If your applications are running on an ECS that is in a different region from the one where the DDS instance is located, use an EIP to connect the ECS to your DDS instances. • If your applications are deployed on other cloud platforms, EIP is recommended.	Low security
Application connection	Private IP address	Connect to the database using different applications.	• **Connecting to the Database Using Java** • **Connecting to the Database Using Python**

2.2 Creating and Managing Databases

For details about the rules of the write/update and delete commands, see **Write/Update** and **Delete**.

Procedure

Step 1 Create database **info**.

use info

Enter **db**. If the following information is displayed, the database is opened.

```
info
```

Step 2 Insert a data record into the database.

db.user.insert({"name": "joe"})

> ⚠ CAUTION
>
> Implicit collection is a collection that is created after data is inserted. To create a collection (table), you must insert a document (record) to make the collection creation take effect.

Step 3 View the database

To view all databases, run the following command:

show dbs

The command output is as follows:


```
admin   0.000GB
config  0.000GB
info    0.000GB
local   0.083GB
```

Step 4 Delete the database. Run the following command to delete the **info** database:

db.dropDatabase()

The database is deleted if the following information is displayed:

```
{"dropped" : "info", "ok" : 1}
```

----End

2.3 Creating and Managing Collections

For details about the rules of the write/update and delete commands, see **Write/Update** and **Delete**.

Creating a Collection

Step 1 Run **db.createCollection(name, options)** to create a collection.

```
db.createCollection(<name>, { capped: <boolean>,
                              autoIndexId: <boolean>,
                              size: <number>,
                              max: <number>,
                              storageEngine: <document>,
                              validator: <document>,
                              validationLevel: <string>,
                              validationAction: <string>,
                              indexOptionDefaults: <document>,
                              viewOn: <string>,
                              pipeline: <pipeline>,
                              collation: <document>,
                              writeConcern: <document>}
```

Table 2-2 Parameter description

Field	Type	Description
apped	boolean	Optional. If a capped collection needs to be created, the value is **true**. If the value is **true**, the **size** field needs to be specified at the same time.
autoIndexId	boolean	If this parameter is set to **false**, an index cannot be automatically created in the **_id** field.

Field	Type	Description
size	number	Optional. For a capped collection, this parameter specifies the maximum size of the collection.
max	number	Optional. For a capped collection, this parameter specifies the maximum number of documents that can be stored in the collection. For details, see the **official document**.

If the following information is displayed, the creation is successful:

```
{ "ok" : 1 }
```

Step 2 Inserts a record into the collection.

db.coll.insert({"name": "sample"})

Step 3 View existing collections.

show collections

Step 4 Delete the collection.

db.coll.drop()

----End

Creating a Capped Collection

Capped collections are fixed-size collections. Once a collection is full, it makes room for new documents by overwriting the oldest documents in the collection.

Run the following command to create a collection. The maximum size of the collection is 5 MB, and the maximum number of documents is 5,000.

db.createCollection("log", { capped : true, size : 5242880, max : 5000 })

Creating a Sharded Collection

In the DDS cluster architecture, you can create shards to fully utilize database performance. For details about the rules and suggestions for creating shards, see **Sharding**.

Step 1 Enable sharding on the database.

sh.enableSharding("info")

Step 2 Create a sharded table and specify the shard key. The following command means that the fruit collection in database info is sharded using shard key _id.

sh.shardCollection("info.fruit", {"_id": "hashed"})

NOTE

DDS sharded clusters support two types of sharding policy:

- Range-based sharding allows querying a range of rows by the shard key values.
- Hashed sharding evenly distributes writes to each shard.

----End

Deleting a Collection

Run **db.collection_name.drop()** to delete the collection.

2.4 Creating and Managing Indexes

DDS uses indexes to improve query efficiency. If there is no index, DDS must scan each document in a collection to select the documents that match the query statement. If a query has an appropriate index, DDS can use the index to limit the number of documents it must examine.

- For details about the rules for creating indexes, see **Index**.
- For details about the rules of the write/update and delete commands, see **Write/Update** and **Delete**.

Indexes

Index	Description
Default index	DDS creates a unique index on the **_id** field during the creation of a collection. A unique index ensures that the indexed fields do not store duplicate values. Do not delete the index from the **_id** field. In a sharded cluster, if you do not use the **_id** field as the shard key, your application needs to ensure that the value in the **_id** field is unique to prevent errors. This is usually done by using the standard automatically generated ObjectId.
Single field index	In addition to the **_id** index defined by DDS, DDS also supports the creation of user-defined ascending/descending indexes on a single field of a document. For single-field indexing and sort operations, the sort order (ascending or descending) of index keys is not important because DDS can traverse the index from any direction.

Index	Description
Compound indexes	DDS also supports compound indexes where a single index field contains references to multiple fields. The order of the fields listed in a compound index is important. For example, if there is a compound index {userid: 1, score: -1}, the index is first sorted by userid and then sorted by score within each userid value. The sort order (ascending or descending) of the index keys determines whether the index supports sort operations.
Multikey index	DDS uses a multikey index to index the content stored in arrays. If the index contains fields with array values, DDS creates a separate index entry for each element of the array. These multikey indexes allow queries to select documents that contain an array by matching one or more elements of the array. DDS automatically determines whether to create a multi-key index. If the index field contains an array value, you do not need to explicitly specify the multikey type.

Index Name

The default name for an index is the concatenation of the indexed keys and each key's direction in the index (i.e. 1 or -1) using underscores as a separator. For example, an index created on **{ item : 1, quantity: -1 }** has the name **item_1_quantity_-1**.

You can create indexes with a custom name, such as one that is more human-readable than the default. For example, consider an application that frequently queries the products collection to populate data on existing inventory. The following createIndex() method creates an index on item and quantity named query for inventory:

db.products.createIndex({ item: 1, quantity: -1 } , { name: "query for inventory" })

You can use the **db.collection.getIndexes()** method to view the index name. Once an index is created, you cannot rename it. Instead, you must drop and recreate the index with the new name.

DDS provides many different index types to support specific types of data and queries.

Creating an Index

Step 1 Run the following command to create an index:

db.collection.createIndex(keys, options)

- **key** is the index field to be created. The value **1** indicates that the index is created in ascending order, and the value **-1** indicates that the index is created in descending order.

- **options** receives optional parameters. The following table lists common optional parameters.

Parameter	Type	Description
background	Boolean	The default value is **false**. The index creation process blocks other database operations. You can specify the background mode to create indexes.
unique	Boolean	The default value is **false**. Whether the created index is unique. If this parameter is set to **true**, a unique index is created.
name	string	Index name. If this parameter is not specified, MongoDB generates an index name by joining the index field name and sorting order.
expireAfterSeconds	integer	TTL value in seconds.

Step 2 Create an index.

- Single field index

  db.user.createIndex({"name": 1})

  The preceding statement creates a single-field index for the **name** field, which can accelerate various query requests on the **name** field. This is the most common index type. The ID index created by default is also of this type. {"name": 1} means that indexed items are sorted in ascending order. You can also use {"name": -1} to sort index items in descending order. For a single-field index, the effect of ascending order is the same as that of descending order.

- Composite index

  A composite index is an upgraded version of a single sub-index. It creates an index for multiple fields. Documents are sorted by the first field, documents with the same first field are sorted by the second field, and so on.

  db.user.createIndex({"name": 1, "age": 1})

- Multikey index
  - If an index field is an array, the created index is called a multikey index.
  - A multikey index creates an index for each element of an array.

For example, if a **habit** field (array) is added to the user collection to describe interests and hobbies, the multikey index of the **habit** field can be used to query people with the same interests and hobbies.

{"name" : "jack", "age" : 19, habit: ["football, runnning"]} //This is a piece of user information in the **person** table.

db.user.createIndex({"habit": 1}) //Multi-key indexes are automatically created.

db.user.find({"habit": "football"}) //Query people with the same interests and hobbies.

Step 3 View the collection index.

db.user.getIndexes()

Step 4 Deletes all indexes from a collection.

db.user.dropIndexes()

Step 5 Deletes a specified index from a collection. Run the following command to delete the **name** index from the user collection:

db,user.dropIndex({"name": 1})

----End

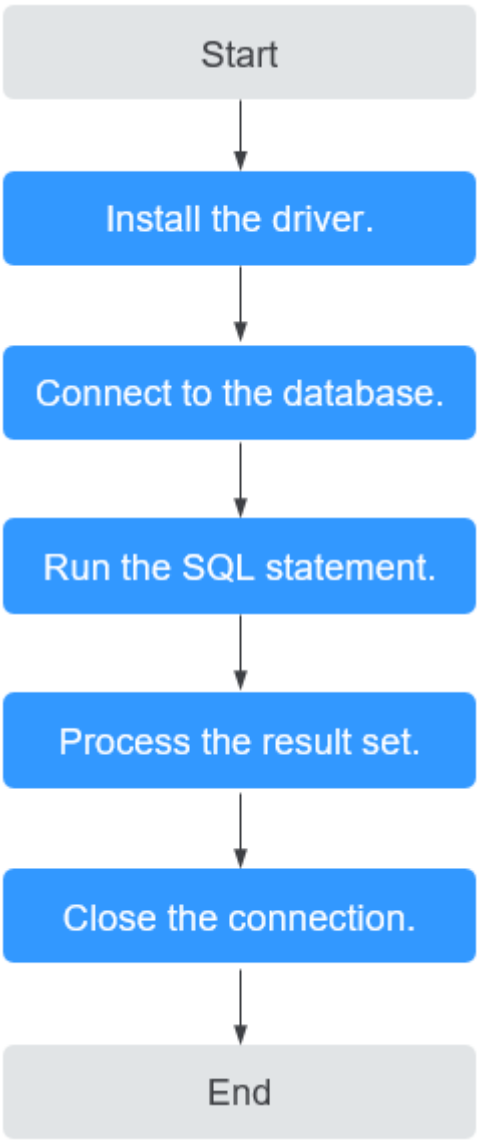
Precaution

In addition to various types of indexes, DDS allows you to customize some special attributes for indexes.

- Unique index: Ensure that the values of the fields corresponding to an index are different. For example, the **_id** index is a unique index.
- TTL index: You can specify the expiration time of a document based on a time field. The document expires after the specified time or at a specified time point.
- Partial index: An index is created only for documents that meet a specific condition.
- Sparse index: Indexes are created only for documents that have index fields, which can be considered as a special case of partial indexes.

3 Application Development

3.1 Development Process



3.2 Common Parameter Configuration on the Driver Side

Common Configuration Items and Recommended Values for Connecting to a DDS DB Instance

1. **connectTimeoutMS**: This connection timeout parameter prevents infinite waiting of the driver during the connection. Recommended configuration:

   connectTimeoutMS = 10000ms

2. **socketTimeoutMS**: This parameter prevents infinite waiting of TCP communication. Recommended configuration:

The duration is two to three times the maximum service duration. The minimum duration is 10s.

socketTimeoutMS = max(10000ms, 3 times the maximum service time)

3. **minPoolSize**: minimum number of connections in the connection pool. Recommended configuration:

   minPoolSize = 10

4. **maxPoolSize**: maximum number of connections in the connection pool. Recommended configuration:

   maxPoolSize = 50 - 100

5. **maxIdleTimeMS**: maximum idle duration that a connection can remain in the pool before it is deleted and closed. Recommended configuration:

   maxIdleTimeMS = 10000ms

CAUTION

Do not use **socketTimeoutMS** to prevent an operation from running for a long time on the database side. **maxTimeMS** is required so that the server can cancel operations that have been abandoned by the client.

3.3 Java-based Development

3.3.1 Driver Package and Environment Dependency

DDS allows you to perform operations using Java. In this way, you can connect to an instance through an SSL connection or an unencrypted connection. The SSL connection encrypts data and is more secure.

SSL is disabled by default for a new DDS instance. To enable SSL, see **Enabling SSL**.

Installing the Driver

1. Click the **JAR driver download address** to download **mongo-java-driver-3.12.9.jar**, which provides APIs for accessing DDS DB instances.
2. To install the driver, see the **official guide**.

Environment

JDK1.8 must be configured for the client. JDK is cross-platform and supports Windows and Linux.

The following uses Windows as an example to describe how to configure JDK:

Step 1 In the DOS window, run **java -version** to check the JDK version. Ensure that the JDK version is 1.8. If JDK is not installed, download the installation package and install it.

Step 2 Right-click the **This PC** icon on the desktop of the Windows OS and choose **Properties** from the shortcut menu.

Step 3 In the **System** window, click **Advanced system settings** in the navigation tree on the left.

Step 4 In the dialog box that is displayed, click **Environment Variables**.

Step 5 In the displayed window, set the variables listed in the following table.

Variable	Operation	Value
JAVA_HOME	• If this parameter exists, click **Edit**. • If this parameter does not exist, click **New**.	Java installation directory. For example, C:\Program Files\Java\jdk1.8.0_131.
Path	Click **Edit**.	• If JAVA_HOME is configured, add **%JAVA_HOME %\bin** to the beginning of the variable value. • If JAVA_HOME is not configured, add the following Java installation path before the variable value. C:\Program Files\Java\jdk1.8.0_131\bin;
CLASSPATH	Click **New**.	.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar;

Step 6 Click **OK** and close the windows in sequence.

----End

3.3.2 Connecting to a Database

Using an SSL Certificate

NOTE

Download the SSL certificate and verify the certificate before connecting to databases.

In the **DB Instance Information** area on the **Basic Information** page, click in the **SSL** field to download the root certificate or certificate bundle.

Step 1 Use Java to connect to the MongoDB database.

- Connect to a single node:
  mongodb://<username>:<password>**@**<instance_ip>:<instance_port>**/**<database_name>**?authSource=admin&ssl=true**
- Connect to a replica set.
  mongodb://<username>:<password>**@**<instance_ip>:<instance_port>**/**<database_name>**?authSource=admin&replicaSet=replica&ssl=true**
- Connect to a cluster:
  mongodb://<username>:<password>**@**<instance_ip>:<instance_port>**/**<database_name>**?authSource=admin&ssl=true**

Table 3-1 Parameter description

Parameter	Description
<username>	Current username
<password>	Password for the current username
<instance_ip>	If you access an instance from an ECS, *instance_ip* is the private IP address of the instance.
	If you access an instance through an EIP, *instance_ip* is the EIP that has been bound to the instance.
<instance_port>	Database port displayed on the **Basic Information** page. Default value: **8635**
<database_name>	Name of the database to be connected.
authSource	Authentication database. The value is **admin**.
ssl	Connection mode. **true** indicates that SSL will be used.

Example script in Java:

```
import java.util.ArrayList;
import java.util.List;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
import com.mongodb.ServerAddress;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.MongoCollection;
import com.mongodb.MongoClientURI;
import com.mongodb.MongoClientOptions;
public class MongoDBJDBC {
public static void main(String[] args){
    try {
        System.setProperty("javax.net.ssl.trustStore", "/home/Mike/jdk1.8.0_112/jre/lib/
security/mongostore");
        System.setProperty("javax.net.ssl.trustStorePassword", "****");
        ServerAddress serverAddress = new ServerAddress("ip", port);
        List addrs = new ArrayList();
        addrs.add(serverAddress);
        MongoCredential credential = MongoCredential.createScramSha1Credential("rwuser",
"admin", "!rwuserPassword".toCharArray());
        List credentials = new ArrayList();
        credentials.add(credential);
        MongoClientOptions opts= MongoClientOptions.builder()
        .sslEnabled(true)
        .sslInvalidHostNameAllowed(true)
        .build();
        MongoClient mongoClient = new MongoClient(addrs,credentials,opts);
        MongoDatabase mongoDatabase = mongoClient.getDatabase("testdb");
        MongoCollection collection = mongoDatabase.getCollection("testCollection");
        Document document = new Document("title", "MongoDB").
        append("description", "database").
        append("likes", 100).
        append("by", "Fly");
```



```
            List documents = new ArrayList();
            documents.add(document);
            collection.insertMany(documents);
            System.out.println("Connect to database successfully");
            } catch (Exception e) {
            System.err.println( e.getClass().getName() + ": " + e.getMessage() );
        }
    }
}
```

Sample code:

```
javac -cp .:mongo-java-driver-3.2.0.jar MongoDBJDBC.java
java -cp .:mongo-java-driver-3.2.0.jar MongoDBJDBC
```

----End

Connection Without the SSL Certificate

NOTE

You do not need to download the SSL certificate because certificate verification on the server is not required.

Step 1 Use Java to connect to the MongoDB database. The Java code format is as follows:

- Connect to a single node:
  mongodb://<username>**:**<password>**@**<instance_ip>**:**<instance_port>**/**<database_name>**?authSource=admin**
- Connect to a replica set.
  mongodb://<username>**:**<password>**@**<instance_ip>**:**<instance_port>**/**<database_name>**?authSource=admin&replicaSet=replica**
- Connect to a cluster:
  mongodb://<username>**:**<password>**@**<instance_ip>**:**<instance_port>**/**<database_name>**?authSource=admin**

Table 3-2 Parameter description

Parameter	Description
<username>	Current username
<password>	Password for the current username
<instance_ip>	If you access an instance from an ECS, *instance_ip* is the private IP address of the instance.
	If you access an instance through an EIP, *instance_ip* is the EIP that has been bound to the instance.
<instance_port>	Database port displayed on the **Basic Information** page. Default value: **8635**
<database_name>	Name of the database to be connected.
authSource	Authentication database. The value is **admin**.

Example script in Java:


```
import com.mongodb.ConnectionString;
import com.mongodb.reactivestreams.client.MongoClients;
import com.mongodb.reactivestreams.client.MongoClient;
import com.mongodb.reactivestreams.client.MongoDatabase;
import com.mongodb.MongoClientSettings;
public class MyConnTest {
    final public static void main(String[] args) {
    try {
        // no ssl
    ConnectionString connString = new ConnectionString("mongodb://
rwuser:****@192.*.*.*:8635,192.*.*.*:8635/test? authSource=admin");
    MongoClientSettings settings = MongoClientSettings.builder()
        .applyConnectionString(connString)
        .retryWrites(true)
        .build();
    MongoClient mongoClient = MongoClients.create(settings);
    MongoDatabase database = mongoClient.getDatabase("test");
     System.out.println("Connect to database successfully");
    } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Test failed");
        }
}
}
```

----End

3.3.3 Accessing a Database

Before accessing a database, import the following classes:

```
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import static com.mongodb.client.model.Filters.*;
import com.mongodb.client.model.CreateCollectionOptions;
import com.mongodb.client.model.ValidationOptions;
```

Accessing DataBase

If an initialized MongoClient instance exists, run the following command to access a database:

```
MongoDatabase database = mongoClient.getDatabase("test");
```

Assessing a Collection

After obtaining a MongoDatabase instance, run the following command to obtain a collection:

```
MongoCollection<Document> coll = database.getCollection("testCollection");
```

Creating a Collection

You can use the createCollection() method to create a collection and specify the attributes of the collection.

```
database.createCollection("testCollection", new CreateCollectionOptions()..sizeInBytes(200000))
```


Inserting Data

```
Document doc0 = new Document("name", "zhangsan")
                .append("age", 3)
                .append("sex", "male");

Document doc1 = new Document("name", "LiSi")
                .append("age", 2)
                .append("sex", "female");

Document doc2 = new Document("name", "wangmazi")
                .append("age", 5)
                .append("sex", "male");

List<Document> documents = new ArrayList<Document>();
documents.add(doc1);
documents.add(doc2);

collection.insertMany(documents);
```

Deleting Data

```
collection.deleteOne(eq("_id", new ObjectId("00000001")));
```

Deleting a Table

```
MongoCollection<Document> collection = database.getCollection("test");
collection.drop()
```

Reading Data

```
MongoCollection<Document> collection = database.getCollection("contacts");
MongoCursor<String> cursor = collection.find();
while (cursor.hasNext())  {
   Object result = cursor.next();
}
```

Query with Filter Criteria

```
MongoCollection<Document> collection = database.getCollection("test");
MongoCursor<String> cursor = collection.find(
    new Document("name","zhangsan")
          .append("age: 5));
while (cursor.hasNext())  {
   Object result = cursor.next();
}
```

Running Commands

Run **buildInfo** and **collStats**.

```
MongoClient mongoClient = MongoClients.create();
MongoDatabase database = mongoClient.getDatabase("test");

Document buildInfoResults = database.runCommand(new Document("buildInfo", 1));
System.out.println(buildInfoResults.toJson());

Document collStatsResults = database.runCommand(new Document("collStats", "restaurants"));
System.out.println(collStatsResults.toJson());
```


Creating an Index

```
MongoClient mongoClient = MongoClients.create();
MongoDatabase database = mongoClient.getDatabase("test");
MongoCollection<Document> collection = database.getCollection("test");
collection.createIndex(Indexes.ascending("age"));
```

3.3.4 Complete Example

```
package mongodbdemo;
import org.bson.*;
import com.mongodb.*;
import com.mongodb.client.*;
public class MongodbDemo {
    public static void main(String[] args) {
        String mongoUri = "mongodb://mongouser:thepasswordA1@10.66.187.127:27017/admin";
        MongoClientURI connStr = new MongoClientURI(mongoUri);
        MongoClient mongoClient = new MongoClient(connStr);
        try {
            //Use the database named someonedb.
            MongoDatabase database = mongoClient.getDatabase("someonedb");
            //Obtain the someonetable handle of the collection/table.
            MongoCollection<Document> collection = database.getCollection("someonetable");
             //Prepare data to be written.
            Document doc = new Document();
            doc.append("key", "value");
            doc.append("username", "jack");
            doc.append("age", 31);
             //Write data.
            collection.insertOne(doc);
            System.out.println("insert document: " + doc);
             //Read data.
            BsonDocument filter = new BsonDocument();
            filter.append("username", new BsonString("jack"));
            MongoCursor<Document> cursor = collection.find(filter).iterator();
            while (cursor.hasNext()) {
                System.out.println("find document: " + cursor.next());
            }
        } finally {
            //Close the connection.
            mongoClient.close();
        }
    }
}
```

For more information about Java APIs, see the **official documents**.

3.4 Python-based Development

3.4.1 PyMongo Package

Python uses PyMongo to provide a unified API for accessing DDS databases. Applications can perform operations based on PyMongo. PyMongo supports SSL connections and uses a connection pool to support multithreaded applications.

To install PyMongo, see the **official guide**.

3.4.2 Connecting to a Database

If you are connecting to an instance using Python, an SSL certificate is optional, but downloading an SSL certificate and encrypting the connection will improve the security of your instance.

SSL is disabled by default for a new DDS instance. To enable SSL, see **Enabling SSL**.

Prerequisites

1. To connect an ECS to an instance, the ECS must be able to communicate with the DDS instance. You can run the following command to connect to the IP address and port of the instance server to test the network connectivity.

   curl *ip:port*

   If the message **It looks like you are trying to access MongoDB over HTTP on the native driver port** is displayed, the ECS and DDS instance can communicate with each other.
2. Install Python and third-party installation package **pymongo** on the ECS. Pymongo 2.8 is recommended.
3. If SSL is enabled, download the root certificate and upload it to the ECS.

Connection Code

- Enabling SSL

```
dbs = connection.database_names()
connection = MongoClient(conn_urls,connectTimeoutMS=5000,ssl=True,
ssl_cert_reqs=ssl.CERT_REQUIRED,ssl_match_hostname=False,ssl_ca_certs=${path to certificate authority file})
conn_urls="mongodb://rwuser:rwuserpassword@ip:port/{mydb}?authSource=admin"
from pymongo import MongoClient
import ssl
est_database
print "connect database success! database names is %s" % dbs
```

- Disabling SSL

```
import ssl
from pymongo import MongoClient
conn_urls="mongodb://rwuser:rwuserpassword@ip:port/{mydb}?authSource=admin"
connection = MongoClient(conn_urls,connectTimeoutMS=5000)
dbs = connection.database_names()
print "connect database success! database names is %s" % dbs
```

NOTE

The authentication database in the URL must be **admin**. Set **authSource** to **admin**.

3.4.3 Accessing a Database

Assume that the client application has connected to the database and a MongoClient client is initialized.

Accessing DataBase

If an initialized MongoClient instance exists, run the following command to access a database:


```
db=client.test_database
```

Alternatively, use the following method:

```
db=client["test_database"]
```

Assessing a Collection

```
collection=db.test_collection
```

Alternatively, use the following method:

```
collection=db["test_collection"]
```

Creating a Collection

You can use the createCollection() method to create a collection and specify the attributes of the collection.

```
collection = db.create_collection("test")
```

Inserting Data

```
student = {
    'id': '20170101',
    'name': 'Jordan',
    'age': 20,
    'gender': 'male'
}
result = collection.insert(student);
```

Deleting Data

```
result = collection.delete_one({'name': 'Kevin'})
```

Deleting a Table

```
db.drop_collection("test")
```

Reading Data

```
result = collection.find_one({'name': 'Mike'})
```

Query with Filter Criteria

```
result = collection.find_one({"author":"Mike"}
```

Running Commands

Run **buildInfo** and **collStats**.

```
db.command("collstats","test")
db.command("buildinfo")
```

Counting

```
count = collection.find().count()db.command("buildinfo")
```


Sorting

```
results = collection.find().sort('name', pymongo.ASCENDING)
print([result['name'] for result in results])
```

Creating an Index

```
result=db.profiles.create_index([('user_id',pymongo.ASCENDING)],... unique=True)
```

3.4.4 Complete Example

```
#!/usr/bin/python
import pymongo
import random
mongodbUri = 'mongodb://mongouser:thepasswordA1@10.66.187.127:27017/admin'
client = pymongo.MongoClient(mongodbUri)
db = client.somedb
db.user.drop()
element_num=10
for id in range(element_num):
   name = random.choice(['R9','cat','owen','lee','J'])
   sex = random.choice(['male','female'])
   db.user.insert_one({'id':id, 'name':name, 'sex':sex})
content = db.user.find()
for i in content:
   print i
```

For more information about PyMongo APIs, see the **official document**.

3.5 Golang-based Development

3.5.1 Driver Package

DDS allows you to operate data using Go. You can connect to a DB instance through an SSL connection or an unencrypted connection. SSL connection is more secure.

SSL is disabled by default for a new DDS instance. To enable SSL, see **Enabling SSL**.

Downloading the Driver

To download the driver, go mod is recommended.

```
require go.mongodb.org/mongo-driver v1.12.1
```

Import the go files.

```
import (
"go.mongodb.org/mongo-driver/bson"
"go.mongodb.org/mongo-driver/mongo"
"go.mongodb.org/mongo-driver/mongo/options"
"go.mongodb.org/mongo-driver/mongo/readpref"
)
```


3.5.2 Connecting to a Database

Prerequisites

1. To connect an ECS to an instance, the ECS must be able to communicate with the DDS instance. You can run the following command to connect to the IP address and port of the instance server to test the network connectivity.

   curl *ip:port*

   If the message **It looks like you are trying to access MongoDB over HTTP on the native driver port** is displayed, the ECS and DDS instance can communicate with each other.

2. If SSL is enabled, download the root certificate and upload it to the ECS.

Connection Code

- Enabling SSL

```
//Construct an authentication credential.
credential := options.Credential{
  AuthMechanism: "SCRAM-SHA-1",
  AuthSource:    "admin",
  Username:      "rwuser",
  Password:      "*******",//Password for connecting to the DDS.
}
//HA URI. Note that SetDirect is set to false.
highProxyUri := "mongodb://host1:8635,host2:8635/?ssl=true"
clientOpts := options.Client().ApplyURI(highProxyUri)
clientOpts = clientOpts.SetTLSConfig(&tls.Config {
InsecureSkipVerify: true,
}).SetDirect(false).SetAuth(credential)
//URI of the direct connection. Note that SetDirect is set to true.
//directUri := "mongodb://host:8635/?ssl=true"
//clientOpts := options.Client().ApplyURI(highProxyUri)
//clientOpts = clientOpts.SetTLSConfig(&tls.Config {
// InsecureSkipVerify: true,
//}).SetDirect(true).SetAuth(credential)
// Connecting to an instance
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()
client, err := mongo.Connect(ctx, clientOpts)
if err != nil {
   fmt.Println("Failed to connect to the mongo instance:", err)
   return
}
//Ping the primary node.
ctx, cancel = context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()err = client.Ping(ctx, readpref.Primary())
if err != nil {fmt.Println ("Failed to ping the primary node: ",err)
   return
}
//Select the database and collection.
collection := client.Database("test").Collection("numbers")
//Insert a record.
ctx, cancel = context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()
oneRes, err := collection.InsertOne(ctx, bson.D{{"name", "e"}, {"value", 2.718}})
if err != nil{fmt.Println("Failed to insert a record: ",err)
   return
}else {
```



```
    fmt.Println(oneRes)
}
// Batch insert
ctx, cancel = context.WithTimeout(context.Background(), 100*time.Second)
defer cancel()
docs := make([]interface{}, 100)
for i := 0; i < 100; i++{
    docs[i] = bson.D{{"name", "name"+strconv.Itoa(i)}, {"value", i}}
}
manyRes, err := collection.InsertMany(ctx, docs)
if err != nil {
    fmt.Println("Batch insertion failed: ",err)
    return
}else {
    fmt.Println(manyRes)
}
```

- Disabling SSL

```
//HA connection. The value of readPreference is primary (by default), indicating that the primary node is read-only. The value primaryPreferred indicates that the primary node is preferred. If the primary node is unavailable, the secondary node is read.
// Secondary: read-only node. If the secondary node is unavailable, an error is reported.
secondaryPreferred: The secondary node is preferred. If the secondary node is unavailable, the primary node is read.
highProxyUri := "mongodb://rwuser:your_password@host1:8635,host2:8635/?authSource=admin&replicaSet=replica&readPreference=secondaryPreferred"
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()
clientOpts := options.Client().ApplyURI(highProxyUri)
client, err := mongo.Connect(ctx, clientOpts)
//Ping the primary node.
ctx, cancel = context.WithTimeout(context.Background(), 2*time.Second)
defer cancel()err = client.Ping(ctx, readpref.Primary())
if err != nil {
    fmt.Println("Failed to ping the primary node: ",err)
    return
}
//Select the database and collection.
collection := client.Database("test").Collection("numbers")
//Insert a record.
ctx, cancel = context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()
res, err := collection.InsertOne(ctx, bson.D{{"name", "e"}, {"value", 2.718}})
if err != nil{
    fmt.Println("Failed to insert a record: ",err)
    return
}else {
    fmt.Println(res)
}
```

3.5.3 Accessing a Database

Accessing DataBase

If an initialized MongoClient instance exists, run the following command to access a database:

```
db:= client.Database("test")
```


Assessing a Collection

After obtaining a MongoDatabase instance, run the following command to obtain a collection:

```
coll := db.Collection("testCollection")
```

Creating a Collection

You can use the CreateCollection() method to create a collection and specify the attributes of the collection.

```
db:= client.Database("test")
ctx, cancel = context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()
sizeInBytes := int64(200000)
testCollection :=
db.CreateCollection(ctx,"testCollection",&options.CreateCollectionOptions{SizeInBytes:
&sizeInBytes})
```

3.5.4 Complete Example

Precautions

1. It is recommended that the context timeout interval be set to a value no less than 10 seconds.
2. **MaxTimeMS** must be set in the following business scenarios:
   - Find
   - FindAndModify
   - DropIndexes
   - Distinct
   - Aggregate
   - CreateIndexes
   - Count

Sample Code

```
import (
    "context"
    "fmt"
    "strconv"
    "time"

    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
    "go.mongodb.org/mongo-driver/mongo/readpref"
)

const (
    ConnectTimeout        = 10 * time.Second
    SocketTimeout         = 60 * time.Second
    MaxIdleTime           = 10 * time.Second
    MaxPoolSize           = 100
    MinPoolSize           = 10
```



```
    DefaultContextTimeOut = 10 * time.Second
    MaxTimeMS             = 10 * time.Second
)

func main() {

    // HA connection string
    highProxyUri := "mongodb://rwuser:your_password@host1:8635,host2:8635/?
authSource=admin&replicaSet=replica&readPreference=secondaryPreferred"
    clientOpts := options.Client().ApplyURI(highProxyUri)
    clientOpts.SetConnectTimeout(ConnectTimeout)
    clientOpts.SetSocketTimeout(SocketTimeout)
    clientOpts.SetMaxConnIdleTime(MaxIdleTime)
    clientOpts.SetMaxPoolSize(MaxPoolSize)
    clientOpts.SetMinPoolSize(MinPoolSize)

    // Connect to a database.
    ConnectCtx, cancel := context.WithTimeout(context.Background(), ConnectTimeout)
    defer cancel()
    client, err := mongo.Connect(ConnectCtx, clientOpts)
    if err != nil {
       fmt.Println("Failed to connect to the mongo instance:", err)
        return
    }
    //Ping the primary node.
    ctx, cancel := context.WithTimeout(context.Background(), DefaultContextTimeOut)
    defer cancel()
    err = client.Ping(ctx, readpref.Primary())
    if err != nil {
        fmt.Println("Failed to ping the primary node:", err)
        return
    }
    //Select the database and collection.
    collection := client.Database("test").Collection("numbers")
    //Insert a data record.
    ctx, cancel = context.WithTimeout(context.Background(), DefaultContextTimeOut)
    defer cancel()
    oneRes, err := collection.InsertOne(ctx, bson.D{{"name", "e"}, {"value", 2.718}})
    if err != nil {
       fmt.Println("Failed to insert a data record:", err)
        return
    } else {
        fmt.Println(oneRes)
    }
    // Batch insert.
    ctx, cancel = context.WithTimeout(context.Background(), DefaultContextTimeOut)
    defer cancel()
    docs := make([]interface{}, 100)
    for i := 0; i < 100; i++ {
        docs[i] = bson.D{{"name", "name" + strconv.Itoa(i)}, {"value", i}}
    }
    manyRes, err := collection.InsertMany(ctx, docs)
    if err != nil {
       fmt.Println("Batch insertion failed:", err)
        return
    } else {
        fmt.Println(manyRes)
    }
    db := client.Database("test")
    // Query data by page.
    ctx, cancel = context.WithTimeout(context.Background(), DefaultContextTimeOut)
    defer cancel()
```



```
    cursor, err := db.Collection("numbers").Find(ctx, struct{}{},
options.Find().SetBatchSize(100).SetMaxTime(MaxTimeMS).SetSkip(int64(1000)).SetLimit(100))
    if err != nil {
        fmt.Println("Pagination query failed:", err)
        return
    }
    for cursor.Next(ctx) {
        fmt.Println(cursor.Current)
    }
}
```

3.6 More Tutorials

For more development tutorials, see **official documents**.

4 Managing Database Permissions

4.1 Default Permission Mechanism

DDS has a series of security enhancements to deal with increasingly severe security challenges. The Community Edition allows you to connect to a database without authentication. In DDS, you must pass the authentication before connecting to the database. Otherwise, the database cannot be used.

- After a DB instance is created, the system creates the default administrator **rwuser**. The administrator must be specified by the customer and meet the password complexity requirements.
- The administrator is used to create and manage user-defined roles and users. The administrator does not have a default password. The password must be specified by the customer and meet the password complexity requirements.

Scenario

During the execution of mongodump and mongorestore, if you back up and restore the entire DB instance, the permission verification fails. This is because user **rwuser** has limited permissions on the **admin** and **config** databases of the DB instance. You need to grant permissions on certain databases and tables to the user.

4.2 Role Management

DDS uses role-based management to control users' data access permissions. Roles are classified into built-in roles and user-defined roles.

4.2.1 Built-In Roles

Built-in roles are automatically generated by the system. The built-in roles read and readWrite can be used by clients.

MongoDB uses roles to manage databases, so you need to assign a role to a user when creating the user. In addition to built-in roles, you can also create user-defined roles (**User-Defined Roles**).

Table 4-1 Common built-in roles

Role	Permission	Actions
read	The read role provides permissions to read data on all non-system collections and some system collections (system.indexes, system.js, and system.namespaces).	changeStream, collStats, dbHash, dbStats, find, killCursors, listIndexes, listCollections
readWrite	The readWrite role provides all the permissions of the read role plus ability to modify data on all non-system collections and the system.js collection.	collStats, convertToCapped, createCollection, dbHash, dbStats, dropCollection, createIndex, dropIndex, find, insert, killCursors, listIndexes, listCollections, remove, renameCollectionSameDB, update
readAnyDatabase	The readAnyDatabase role provides the read-only permissions on all databases except local and config. The role also provides the listDatabases action on the cluster as a whole.	In MongoDB 3.4 and earlier, this role provides the read permission for the config and local databases. In the current version, to provide read permissions on the config and local databases, create a user in the admin database with read role in the config and local databases.
readWriteAnyDatabase	The readWriteAnyDatabase role has the read and write permissions for all databases except config and local. The role also provides the listDatabases action on the cluster as a whole.	In MongoDB 3.4 and earlier, this role has the read and write permissions for the config and local databases. In the current version, if you want to read or write data from or to the config and local databases, create a user in the admin database with the readWrite role in the config and local databases.

Role	Permission	Actions
dbAdmin	The dbAdmin role provides the ability to perform administrative tasks such as schema-related tasks, indexing, and gathering statistics. This role does not grant permissions for user and role management.	• For system collections (system.indexes, system.namespaces, and system.profile), the actions include collStats, dbHash, dbStats, find, killCursors, listIndexes, listCollections, dropCollection, and createCollection (applicable only to system.profile). • For non-system collections, the actions include bypassDocumentValidation, collMod, collStats, compact, convertToCapped, createCollection, createIndex, dbStats, dropCollection, dropDatabase, dropIndex, enableProfiler, reIndex, renameCollectionSameDB, repairDatabase, storageDetails, and validate.
dbAdminAnyDatabase	The dbAdminAnyDatabase role provides the same database management permissions as dbAdmin on all databases except local and config. The role also provides the listDatabases action on the cluster as a whole.	In MongoDB 3.4 and earlier, this role has the management permissions for the config and local databases. In the current version, if you want to manage the two databases, create a user in the admin database with the dbAdmin role in the config and local databases.
clusterAdmin	The clusterAdmin role has the greatest cluster-management access.	This role combines the permissions granted by the clusterManager, clusterMonitor, and hostManager roles, and provides the dropDatabase action.

Role	Permission	Actions
userAdmin	The userAdmin role contains the permissions to create and modify roles and users in the current database. This role allows users to grant any permission to any other user (including themselves). This role also indirectly provides the superuser (root) access to either the database or, if scoped to the admin database, the cluster.	changeCustomData, changePassword, createRole, createUser, dropRole, dropUser, grantRole, revokeRole, setAuthenticationRestriction, viewRole, viewUser
userAdminAnyDatabase	The userAdminAnyDatabase role has the permissions similar to the userAdmin role. It manages all databases except the config and local databases.	• This role contains the following cluster commands: the authSchemaUpgrade, invalidateUserCache, and listDatabases • For system collections (admin.system.users and admin.system.roles), the actions include collStats, dbHash, dbStats, find, killCursors, planCacheRead, createIndex, and dropIndex.

4.2.2 User-Defined Roles

A user-defined role is a customized role created by a user by running a command. It contains only one or more CRUD operations or one or more built-in roles. You can customize roles based on different resources and actions. User-defined roles are applied in the same way as built-in roles.

Creating, Modifying, and Deleting Roles

- Before creating a role, connect to the DB instance as a user with the required permission (for example, rwuser).
- You can use createRole to create a user-defined role to control permissions for different databases and collections or inherit permissions from other roles.
- After a role is created, you can run grantPrivilegesToRole, grantRolesToRole, revokeRolesFromRole, or revokePrivilegesFromRole to obtain or revoke permissions of the role. For details, see **Creating and Managing Roles**.

4.2.3 Creating and Managing Roles

Creating a Role

db.createRole(role, writeConcern)

- **role** is mandatory and its type is document. The details are as follows:

```
{
  role: "<name>",
  privileges: [
     { resource: { <resource> }, actions: [ "<action>", ... ] },
     ...
  ],
  roles: [
     { role: "<role>", db: "<database>" } | "<role>",
      ...
  ],
  authenticationRestrictions: [
    {
      clientSource: ["<IP>" | "<CIDR range>", ...],
      serverAddress: ["<IP>" | "<CIDR range>", ...]
    },
    ...
  ]
}
```

Parameter description

Field	Type	Description
role	string	Role name
privileges	Array	This parameter is mandatory. The array elements indicate the permissions of a role. If this parameter is set to an empty collection, the role does not have any permission.
resource	Documents	The database name or collection name.

Field	Type	Description
actions	Array	List of available operations. Common actions are as follows: • find • count • getMore • listDatabases • listCollections • listIndexes • insert • update • remove For more actions, see the **official document**.
roles	Array	Array element. This parameter is mandatory. The array element is the name of a role inherited by the role. The role can be a preset role read or readWrite or a user-defined role.
authenticationRestrictions	Array	Optional. This parameter specifies the IP address or IP address segment that can be accessed by the role.

- writeConcern specifies the write concern level of a command.

Updating a Role

db.grantPrivilegesToRole(rolename,privileges,writeConcern)

db.revokePrivilegesFromRole(rolename,privileges,writeConcern)

The preceding commands are used to obtain or revoke specified permissions for a role.

- **rolename** specifies the name of the role to be updated. This parameter is mandatory.
- **privileges** indicates the permissions to be adjusted for the role.

```
db.grantPrivilegesToRole(
   "< rolename >",
```



```
  [
     { resource: { <resource> }, actions: [ "<action>", ... ] },
     ...
  ],
  { < writeConcern > }
)
```

Table 4-2 privileges description

Field	Type	Description
resource	Document	The database name or collection name.
actions	Array	For details, see description about createRole.

In addition to the preceding commands, updateRole can also be used to update role information.

db.updateRole(role, update, writeConcern)

Table 4-3 Parameter description

Field	Type	Description
role	string	Role name
update	Array	Mandatory. Its meaning is the same as that of privileges in the command for creating a role. It is used to replace all permission information of a role.
writeConcern	Document	writeConcern specifies the write concern level of a command.

Deleting a Role

db.dropRole(rolename, writeConcern)

- **rolename** specifies the name of the role to be deleted. This parameter is mandatory.
- **writeConcern** specifies the write concern level of a command.

4.3 User Management

In DDS, user permissions are managed based on roles. Differentiated permission control is implemented by assigning different roles to users.

To provide management services for DDS DB instances, admin, monitor, and backup accounts are automatically created when you create a DDS DB instance. Attempting to delete, rename, change the passwords, or change privileges for these accounts will result in errors.

You can change the password of the database administrator **rwuser** and any accounts you create.

4.3.1 Creating a User

Precautions

- All the following operations require permissions. By default, user **rwuser** has the required permissions. If a user-defined user is used for management, check whether the user has the required permissions.
- Connect to a DB instance as a user who has the required permission (for example, **rwuser**).
- You can use createUser to create required users and configure roles to control user rights. Note that the **passwordDigestor** parameter must be set to **server**. Otherwise, the command fails to be executed. This restriction is added to prevent security risks.

Creating a User

db.createUser(user, writeConcern)

- In the command, **user** is mandatory and the type is document. It contains the identity authentication and access information of the user to be created.
- **writeConcern** is an optional parameter of the document type. It contains the write concern level of the creation operation.

The **user** document defines users. The format is as follows:

```
{
  user: "<name>",
  pwd: "<cleartext password>",
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  authenticationRestrictions: [
     {
       clientSource: ["<IP>" | "<CIDR range>", ...],
       serverAddress: ["<IP>" | "<CIDR range>", ...]
     },
     ...
  ]
  mechanisms: [ "<SCRAM-SHA-1|SCRAM-SHA-256>", ... ],
  passwordDigestor: "<server|client>"
}
```


Table 4-4 Description of parameter user

Field	Type	Description
user	string	The new username.
pwd	string	User password. If you run **db.createUser()** on the **$external** database to create a user who stores credentials outside of MongoDB, the **pwd** field is not required.
customData	Document	Optional. Any information, which can be used to store any data that the administrator wants to associate with this particular user. For example, this could be the user's full name or employee ID.
roles	Array	The role assigned to the user. You can specify an empty array [] to create a user without a role.
authenticationRestrictions	Array	Optional. The authentication restrictions forcibly imposed by the server on the created user. It is used to specify the IP address or IP address segment that can be accessed by the role.
mechanisms	Array	Optional. The specific SCRAM mechanism or mechanisms for the user credentials. Valid values are SCRAM-SHA-1 and SCRAM-SHA-256.
passwordDigestor	string	Optional. Whether to verify the password on the server or client. The default value is server.

4.3.2 Updating a User

db.updateUser(username, update, writeConcern)

- **username** indicates the username to be updated.
- **update** is a document containing the replacement data for the user.
- **writeConcern**: The write concern level of the update operation. This parameter is optional.

```
db.updateUser(
  "<username>",
  {
    customData : { <any information> },
    roles : [
      { role: "<role>", db: "<database>" } | "<role>",
      ...
    ],
    pwd: passwordPrompt(),      // Or  "<cleartext password>"
    authenticationRestrictions: [
      {
        clientSource: ["<IP>" | "<CIDR range>", ...],
        serverAddress: ["<IP>", | "<CIDR range>", ...]
      },
      ...
    ],
    mechanisms: [ "<SCRAM-SHA-1|SCRAM-SHA-256>", ... ],
    passwordDigestor: "<server|client>"
  },
  writeConcern: { <write concern> }
)
```

Table 4-5 update description

Field	Type	Description
customData	Documents	Optional. Any information.
roles	Array	Optional. The role assigned to the user. An update to the roles array overrides the previous array's values.
pwd	string	Optional. The user's password.
authenticationRestrictions	Array	Optional. The IP address or CIDR blocks that can be accessed by a role.
mechanisms	Array	Optional. The specific SCRAM mechanism or mechanisms for the user credentials. Valid values are SCRAM-SHA-1 and SCRAM-SHA-256.

Field	Type	Description
passwordDigestor	string	Optional. Whether to verify the password on the server or client. The default value is server.

4.3.3 Deleting a User

db.dropUser(username, writeConcern)

- **username** is the name of the user to be deleted from the database.
- **writeConcern**: The level of write concern for the removal operation. This parameter is optional.

5 System Collections

Table 5-1 Collections of version 4.0

System Collection	Description
admin.system.roles	Stores user-defined roles created and assigned to users to provide access to specific resources.
admin.system.users	Stores a user's authentication credentials and all roles assigned to the user.
admin.system.version	Stores the architecture version of the user credential document.
<database>.system.namespaces	Contains all collection information in the database.
<database>.system.indexes	lists all indexes in a database.
<database>.system.profile	Contains slow query logs of a database.
<database>.system.js	Contains special JavaScript code for server-side JavaScript.
<database>.system.views	Contains information about each view of a database.

6 Common Operations

6.1 Common CRUD Operations

After selecting a database version, you can learn common CRUD operations of MongoDB. For details, see the **official documentation**.