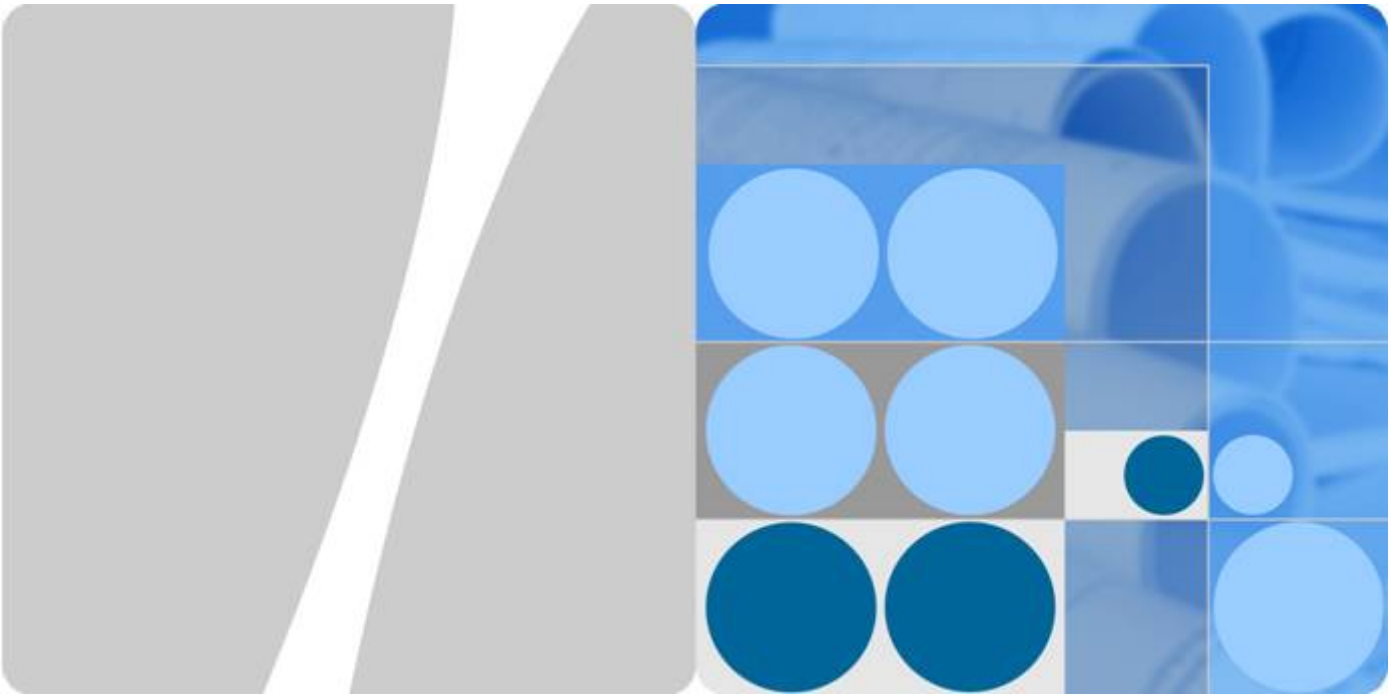


Document Database Service

Best Practices

Issue 01

Date 2020-11-20

HUAWEI TECHNOLOGIES CO., LTD.

HUAWEI

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 Common Methods for Connecting to a DDS DB Instance

This section describes how to connect to a DDS DB instance using the following three measures:

- Mongo shell
- Python Mongo client
- Java Mongo client

Mongo Shell

- Prerequisites
  a. To connect an ECS to a database, the ECS must be able to communicate with the DDS DB instance that runs the database. 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 network connectivity is normal.
  b. On the **MongoDB official website**, you have downloaded the client installation package of version 3.2 or 3.4 that is consistent with the DB instance engine version. Decompress the package, obtain the **mongo** file, and upload it to the ECS.
  c. If SSL is enabled, you need to download the root certificate and upload it to the ECS.
- Connection commands
  – Enabling SSL

    ./mongo *ip:port* **--authenticationDatabase admin -u** *username* **-p** *password* **--ssl --sslCAFile $***path to certificate authority file* **--sslAllowInvalidHostnames**
  – Disabling SSL

    ./mongo *ip:port* **--authenticationDatabase admin -u** *username* **-p** *password*

- Precautions

  a. If SSL is enabled, the connection command must contain **--ssl** and **--sslCAFile**.

  b. The **--authenticationDatabase** value must be set to **admin**. If you log in to the database as user **rwuser**, switch to the admin database for authentication.

For details, see section **Connecting to a DB Instance** in *Document Database Service Getting Started*.

Python Mongo Client

- Prerequisites

  a. To connect an ECS to a database, the ECS must be able to communicate with the DDS DB instance that runs the database. 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 network connectivity is normal.

  b. Install Python and third-party installation package **pymongo** on the ECS. Pymongo 2.8 is recommended.

  c. If SSL is enabled, you need to download the root certificate and upload it to the ECS.

- Input the connection code.

  – Enabling SSL

```
import ssl
from pymongo import MongoClient
conn_urls="mongodb://rwuser:rwuserpassword@ip:port/{mydb}?authSource=admin"
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})
dbs = connection.database_names()
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
```

- Precautions

  The authentication database in the URL must be **admin**. That means setting **authSource** to **admin**.

Java Mongo Client

- Prerequisites

  a. To connect an ECS to a database, the ECS must be able to communicate with the DDS DB instance that runs the database. 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 network connectivity is normal.

b. Download the **MongoDB JAR** package compatible with the DB instance version by referring to the **MongoDB Compatibility** table.

c. The JDK is installed on the ECS.

d. If SSL is enabled, you need to download the root certificate and upload it to the ECS.

- Input the connection code.

  Use keytool to generate a truststore.

  keytool -import -file */var/chroot/mongodb/CA/ca.crt* **-keystore** */home/Mike/jdk1.8.0_112/jre/lib/security/mongostore* **-storetype pkcs12 -storepass** *test123*

  NOTE

  - **/var/chroot/mongodb/CA/ca.crt** indicates the root certificate path.
  - **/home/Mike/jdk1.8.0_112/jre/lib/security/mongostore** indicates the path of the generated truststore.
  - **test123** indicates the password of the truststore.

  – Enabling SSL

```
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", "test123");
            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 codes:

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

– Disabling SSL

```
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 {
            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);
            MongoClient mongoClient = new MongoClient(addrs,credentials);
            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() );
        }
       }
}
```

- Precautions

  a. In SSL mode, you need to manually generate the truststore file.

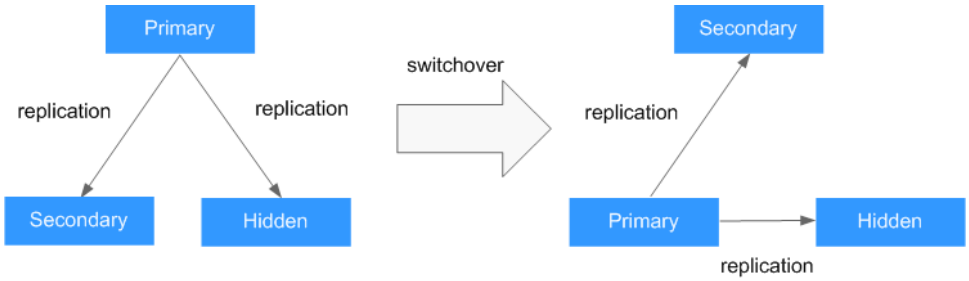
b. The authentication database must be **admin**, and then switch to the service database.

2 Connecting to a Replica Set Instance for Read and Write Separation and High Availability

DDS replica set instances can store multiple duplicates to ensure data high availability and support the automatic change of private IP addresses to ensure service high availability. To enhance the read and write performance, you can use the client to read different data copies. You are advised to use the recommended method to connect the replica set instance. Otherwise, the high availability and high read performance of the replica set instance cannot be guaranteed.

The primary node of a replica set instance is not fixed. If the instance settings are changed, or the primary node fails, or primary and secondary nodes are switched, a new primary node will be elected and the previous one becomes a secondary node. The following figure shows the process of a switchover.

Figure 2-1 Primary/Secondary switchover



(Recommended) Connecting to a Replica Set Instance

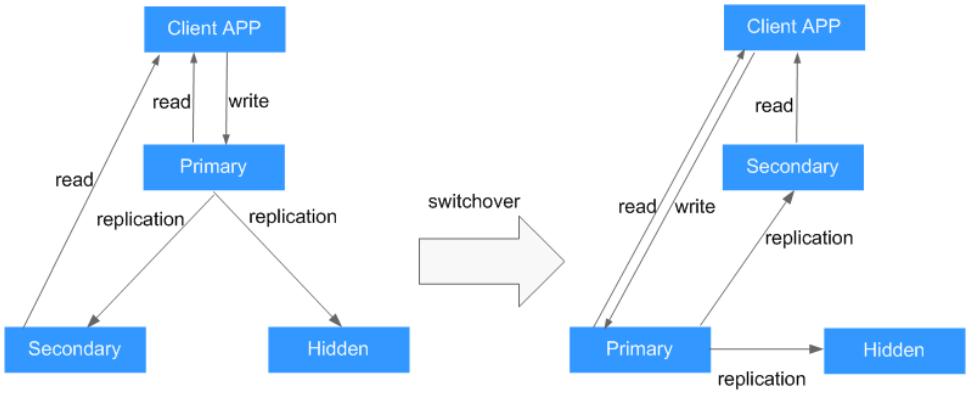
A DDS replica set consists of the primary, secondary, and hidden nodes. Hidden nodes are invisible to users. When you use a URL to connect to a replica set instance, you can connect to the IP addresses and ports of the primary and secondary nodes only.

You are advised to use the following URL to connect a replica set instance:

mongodb://rwuser:**@** *192.168.0.148:8635,192.168.0.96:8635* **/test?authSource=admin&replicaSet=replica**

In the preceding URL, **192.168.0.148:8635** and **192.168.0.96:8635** indicate the IP addresses and ports of the primary and secondary nodes, respectively. If you use this method, the connectivity can be ensured even when a switchover occurs. In addition, using multiple IP addresses and port numbers can enhance the read and write performance of the entire database.

Figure 2-2 Data read and write process



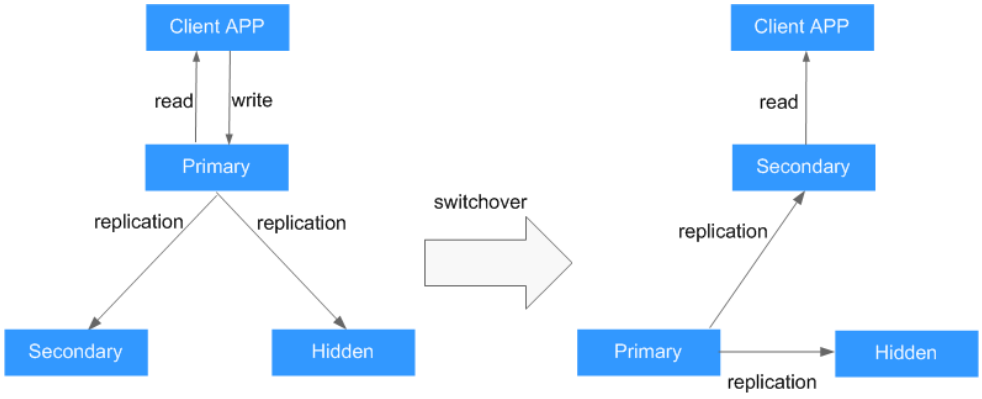
(Not Recommended) Connecting to a Replica Set Instance

Use the following URL to connect a replica set instance:

mongodb://rwuser:**@***192.168.0.148:8635***/test?authSource=admin&replicaSet=replica**

In the preceding URL, **192.168.0.148:8635** indicates the IP address and port number of the current primary node. If a switchover occurs or the primary node is changed, the client fails to connect to the replica set instance because the IP address and port of the newly elected primary node is unknown. As a result, the database service becomes unavailable. In addition, read and write operations can be performed on a fixed primary node only. The read and write performance cannot be expanded to other nodes.

Figure 2-3 Data read and write process



Connecting to a Replica Set Instance Using Java

Sample code:


```
MongoClientURI connectionString = new MongoClientURI("mongodb://
rwuser:****@192.168.0.148:8635,192.168.0.96:8635/test?authSource=admin&replicaSet=replica");
MongoClient client = new MongoClient(connectionString);
MongoDatabase database = client.getDatabase("test");
MongoCollection<Document> collection = database.getCollection("mycoll");
```

NOTE

- **rwuser:******: indicates the username and password for starting authentication.
- **192.168.0.148:8635** and **192.168.0.96:8635**: indicate the IP addresses and ports of the primary and secondary nodes of the replica set instance.
- **test**: indicates the name of the database to be connected to.
- **authSource=admin**: indicates the database to which the username belongs during authentication.
- **replicaSet=replica**: indicates the name of the replica set instance type.

3 Enabling Sharding on a Database

You can shard a large-size collection for a sharded cluster instance. Sharding distributes data across different machines to make full use of the storage space and compute capability of each shard.

NOTE

The operations described in this section apply to cluster instances only.

Procedure

The following is an example using database **mytable**, collection **mycoll**, and the field **name** as the shard key.

1. Log in to a sharded cluster instance using mong shell.
2. Enable sharding for the databases that belong to the cluster instance.
   - Method 1
     ```
     sh.enableSharding("<database>")
     ```
   - Method 2
     ```
     use admin
     db.runCommand({enablesharding:"<database>"}
     ```

   Example:

   ```
   sh.enableSharding("mytable")
   ```
3. Shard a collection.
   - Method 1
     ```
     sh.shardCollection("<database>.<collection>",{"<keyname>":<value> })
     ```
   - Method 2
     ```
     use admin
     db.runCommand({shardcollection:"<database>.<collection>",key:
     {"keyname":<value> }})
     ```


NOTE

- <database>: indicates the database name.
- <collection>: indicates the collection name.
- <keyname>: indicates the shard key, which is used to shard data.
- <value>: indicates the sort order based on the range of the shard key.
  - 1: Ascending indexes
  - -1: Descending indexes
  - hashed: indicates that hash sharding is used. Hashed sharding provides more even data distribution across the sharded cluster.

For details, see **sh.shardCollection()**.

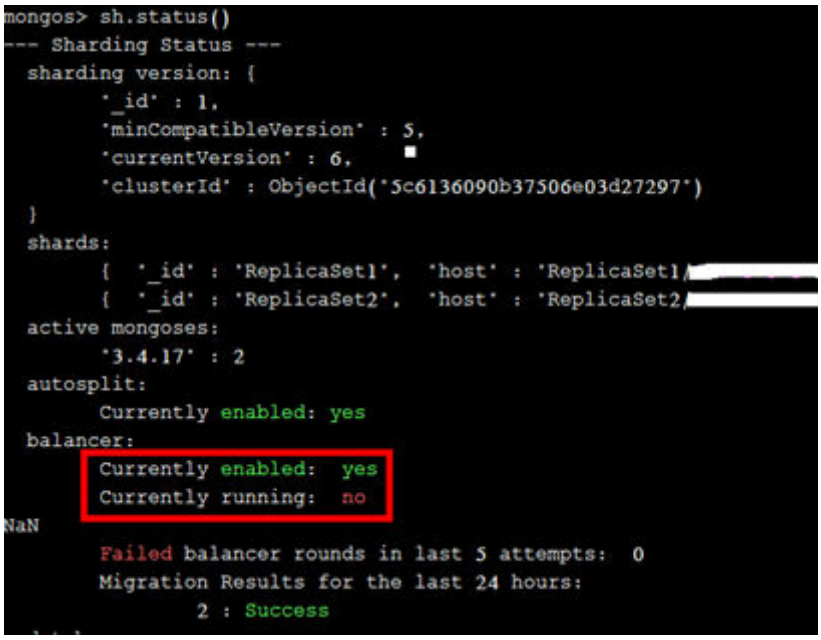
Example:

```
sh.shardCollection("mytable.mycoll",{"name":"hashed"})
```

4. Check the data storage status of the database on each shard.

```
sh.status()
```

Example:



Selecting a Shard Key

Each shard contains collections as its basic unit. Data in the collection is partitioned by the shard key. Shard key is a field in the collection. It distributes data evenly across shards. If you do not select a proper shard key, the cluster performance may deteriorate, and the sharding statement execution process may be blocked.

NOTE

Once the shard key is determined it cannot be changed. If no shard key is suitable for sharding, you need to use a sharding policy and migrate data to a new collection for sharding.

The ideal shard key is as follows:

- All inserts, updates, and deletes are evenly distributed to all shards in the cluster.
- The distribution of keys is sufficient.
- Rare scatter-gather queries.

If the selected shard key does not have all the preceding features, the read and write scalability of the sharded cluster is affected. For example, If the workload of the find() operation is unevenly distributed in the shards, hot shards will be generated. Similarly, if your write load (inserts, updates, and deletes) is not uniformly distributed across your shards, then you could end up with a hot shard. Therefore, you need to adjust the shard keys based on service requirements, such as read/write status, frequently queried data, and written data.

You can use the following dimensions to determine whether the selected shard keys meet your service requirements:

- **Cardinality**

  Cardinality refers to the capability of dividing chunks. For example, if you need to record the student information of a school, you use the age as a shard key for the age can be divided into different ranges. That choice means that data of students of the same age is stored in only one data segment, which may have impact on the performance and manageability of your clusters. A much better shard key would be the student number because it is unique. If the student number is used as a shard key, the relatively large cardinality can ensure the even distribution of data.

- **Write Distribution**

  If a large number of write operations are performed in the same period of time, you want your write load to be evenly distributed over the shards in the cluster. If the data distribution policy is range sharding, a monotonically increasing shard key will guarantee that all inserts go into a single shard.

- **Read Distribution**

  Similarly, if a large number of read operations are performed in the same period, you want your read load to be evenly distributed over the shards in your cluster to fully utilize the computing performance of each shard.

- **Read Targeting**

  The mongos query router can perform either a targeted query (query only one shard) or a scatter/gather query (query all of the shards). The only way for the mongos to be able to target a single shard is to have the shard key present in the query. Therefore, you need to pick a shard key that will be available for use in the common queries while the application is running. If you pick a synthetic shard key, and your application cannot use it during typical queries, all of your queries will become scatter/gather, thus limiting your ability to scale read load.

Choosing a Distribution Policy

A sharded cluster can store a collection's data on multiple shards. You can distribute data based on the shard keys of documents in the collection.

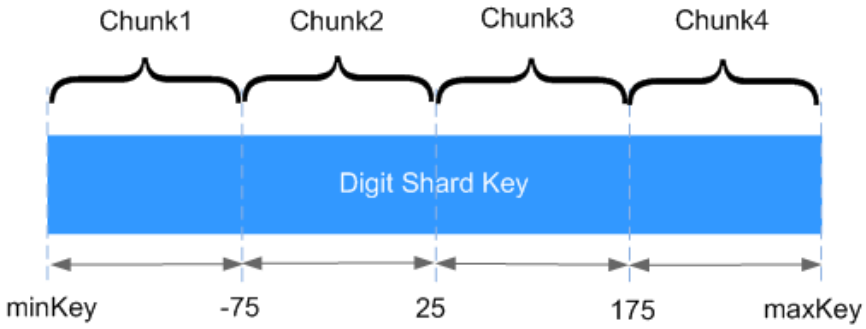
Currently, there are two data distribution policies: range sharding and hash sharding. For details, see **3**.

The following describes the advantages and disadvantages of the two methods.

- **Ranged Sharding**

  Ranged-based sharding involves dividing data into contiguous ranges determined by the shard key values. If you assume that a shard key is a line stretched out from positive infinity and negative infinity, each value of the shard key is the mark on the line. You can also assume small and separate segments of a line and that each chunk contains data of a shard key within a certain range.

  Figure 3-1 Distribution of data

  As shown in the preceding figure, field **x** indicates the shard key of ranged sharding. The value range is [*minKey*,*maxKey*] and the value is an integer. The value range can be divided into multiple chunks, and each chunk (usually 64 MB) contains a small segment of data. For example, chunk 1 contains all documents in range [minKey, -75] and all data of each chunk is stored on the same shard. That means each shard containing multiple chunks. In addition, the data of each shard is stored on the config server and is evenly distributed by mongos based on the workload of each shard.

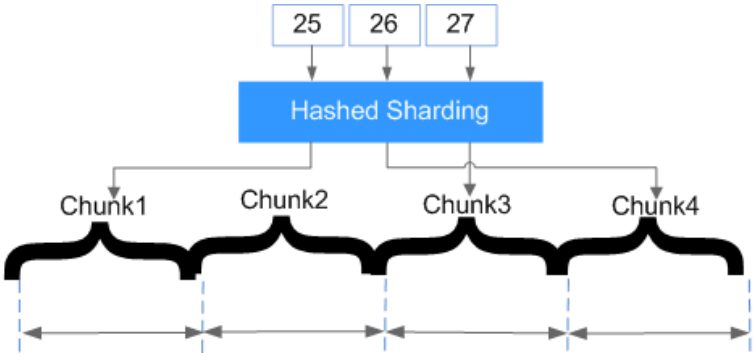
  Ranged sharding can easily meet the requirements of query in a certain range. For example, if you need to query documents whose shard key is in range [-60,20], mongos only needs to forward the request to chunk 2.

  However, if shard keys are in ascending or descending order, newly inserted documents are likely to be distributed to the same chunk, affecting the expansion of write capability. For example, if **_id** is used as a shard key, the high bits of **_id** automatically generated in the cluster are ascending.

- **Hashed Sharding**

  Hashed sharding computes the hash value (64-bit integer) of a single field as the index value; this value is used as your shard key to partition data across your shared cluster. Hashed sharding provides more even data distribution across the sharded cluster because documents with similar shard keys may not be stored in the same chunk.

Figure 3-2 Distribution of data



Hashed sharding randomly distributes documents to each chunk, which fully expands the write capability and makes up for the deficiency of ranged sharding. However, queries in a certain range need to be distributed to all backend shards to obtain documents that meet conditions, resulting in low query efficiency.

4 Improving DDS Performance

DDS is inherently a NoSQL database with high performance and strong extensibility. Similar as RDS, such as RDS for MySQL, Microsoft SQL Server, and Oracle, DDS performance may also be affected by database design, statement optimization, and index creation.

The following provides suggestions for improving DDS performance in different dimensions:

- Creating databases and collections
  a. Use short field names to save storage space. Different from an RDS database, each DDS document has its field names stored in the collection. Therefore, short name has its advantages.
  b. Limit the number of documents in a collection to avoid the impact on the query performance. Archive documents periodically if necessary.
  c. Each document provides a default value of **_id**. Do not set this parameter to a user-defined value.
  d. Compared with other collections, capped collections have a faster insertion speed and can automatically delete old data. You can create a capped collection to improve performance based on service requirements.
- Query Operation
  a. Create proper number of indexes for frequently queried data fields based on service requirements. Indexes occupy some storage space, and inserts and updates consume resources. Therefore, it is recommended that the number of indexes in each collection should not exceed 5.

     Case: Data query is slow. If no index is created, you are advised to create proper indexes for frequently queried data fields to improve the query speed.
  b. For a query that contains multiple shard keys, you are advised to create a compound index that contains these keys. The order of shard keys in a compound index is important. A compound index support queries that use the leftmost prefix of the index, and the query is only relevant to the creation sequence of indexes.
  c. TTL indexes can be used to automatically filter out expired documents and delete them. The index for creating TTL must be of type date. TTL indexes are single-field indexes.

d. You can create field indexes in a collection. However, if a large number of documents in the collection do not contain the key values, you are advised to create sparse indexes.

e. When you create text indexes, the field is specified as **text** instead of **1** or **-1**. Each collection has only one text index, but it can index any fields.

f. The findOne method returns the first document that satisfies the specified query criteria on the collection according to the natural order. To return multiple documents, use this method.

g. If the query does not require the return of the entire document or is only used to determine whether the key value exists, you can use **$project** to limit the returned field, reducing the network traffic and the memory usage of the client.

h. In addition to prefix queries, regular expression queries take longer to execute than using selectors, and indexes are not recommended.

i. Some operators that contain **$** in the query may deteriorate the system performance. The following types of operators are not recommended in services. $or, $nin, $not, $ne, and $exists.

   NOTE

   - $or: The times of query vary depending on the number of conditions. It is used to query all the documents that meet the query conditions in the collection. You are advised to use $in instead.
   - $nin: Matches most of indexes, and the full table scan is performed.
   - $not: The query optimizer may fail to match a specific index, and the full table scan is performed.
   - $ne: Selects the documents where the value of the field is not equal to the specified value. The entire document is scanned.
   - $exists: matches each document that contains the field.

   For more information, see **official MongoDB documents**.

j. If the query results need to be sorted, control the number of result sets.

k. Indexes cannot be used in operators $where and $exists.

l. If multiple field indexes are involved, place the field used for exact match before the index.

m. If the key value sequence in the search criteria is different from that in the compound index, DDS automatically changes the query sequence to the same as index sequence.

- Modification Operation

  Modify a document by using operators can improve performance. This method does not need to obtain and modify document data back and forth on the server, and takes less time to serialize and transfer data.

- Batch Insert

  Batch insert can reduce the number of times data is submitted to the server and improve the performance. The BSON size of the data submitted in batches cannot exceed 48 MB.

- Aggregated Operation

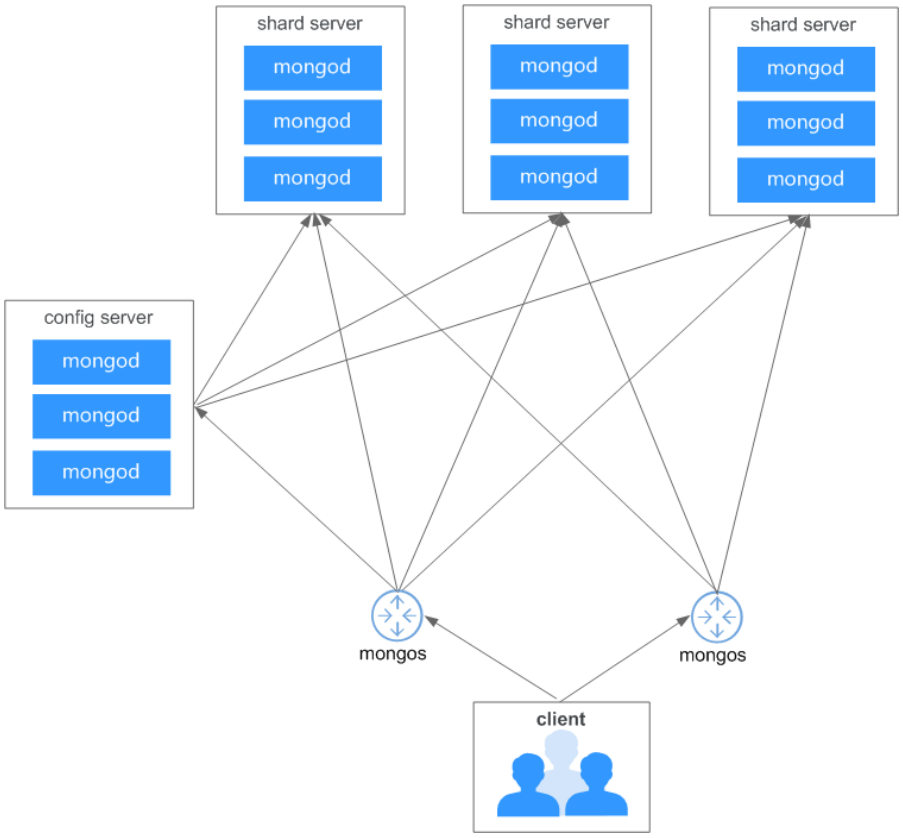
  During aggregation, $match must be placed before $group to reduce the number of documents to be processed by the $group operator.

5 Avoiding mongos Cache Problem

Background Information

DDS is a document-oriented database service based on distributed file storage, famed for its scalability, high performance, open source, and free mode.

Figure 5-1 DDS cluster architecture



A cluster DB instance consists of the following three parts:

- mongos: Is deployed on a single node. It provides APIs to allow access from external users and shields the internal complexity of the distributed database. A DDS cluster can contain 2 to 12 mongos. You can add them as required.

- config server: Is deployed as a replica set. It stores metadata for a sharded cluster. The metadata include information about routes and shards. A cluster contains only one config server.
- shard server: Is deployed as a replica set. It stores user data on shards. You can add shard servers in a cluster as required.

Sharding

Sharding is a method for distributing data evenly across multiple shard servers based on a specified shard key. The collection that has a shard key is called sharded collection. If the collection is not sharded, data is stored on only one shard server. DDS cluster mode allows the coexistence of sharded collection and non-sharded collection.

You can run the **sh.shardCollection** command to convert a non-sharded collection into a sharded collection. Before sharding, ensure that the sharding function is enabled on the database where the collections to be sharded are located. You can run the **sh.enableSharding** command to enable the sharding function.

Caching Metadata with mongos

User data is stored in the shard server and metadata is stored in the config server. The route information belongs to metadata and is also stored in the config server. When a user needs to access data through mongos, mongos sends the user's requests to the corresponding shard server according to the route information stored on the config server.

This means that every time the user accesses the data, mongos needs to connect to the config server for the route information, which may affect the system performance. Therefore, a cache mechanism is developed for the mongos to cache the route information of the config server. In this scenario, not only the config server stores the route information, but also the mongos caches the route information.

If no operation is performed on mongos, mongos does not cache any route information. In addition, the route information cached on mongos may not be the latest because the information is only updated in the following scenarios:

- If the mongos is started, it will obtain the latest route information from the config server and caches them locally.
- If the mongos processes the data request for the first time, it will obtain the route information from the config server. After that, the information is cached and can be used directly at the time when it is required.
- Updating route information by running commands on mongos.

NOTE

Only the metadata related to the requested data is updated.

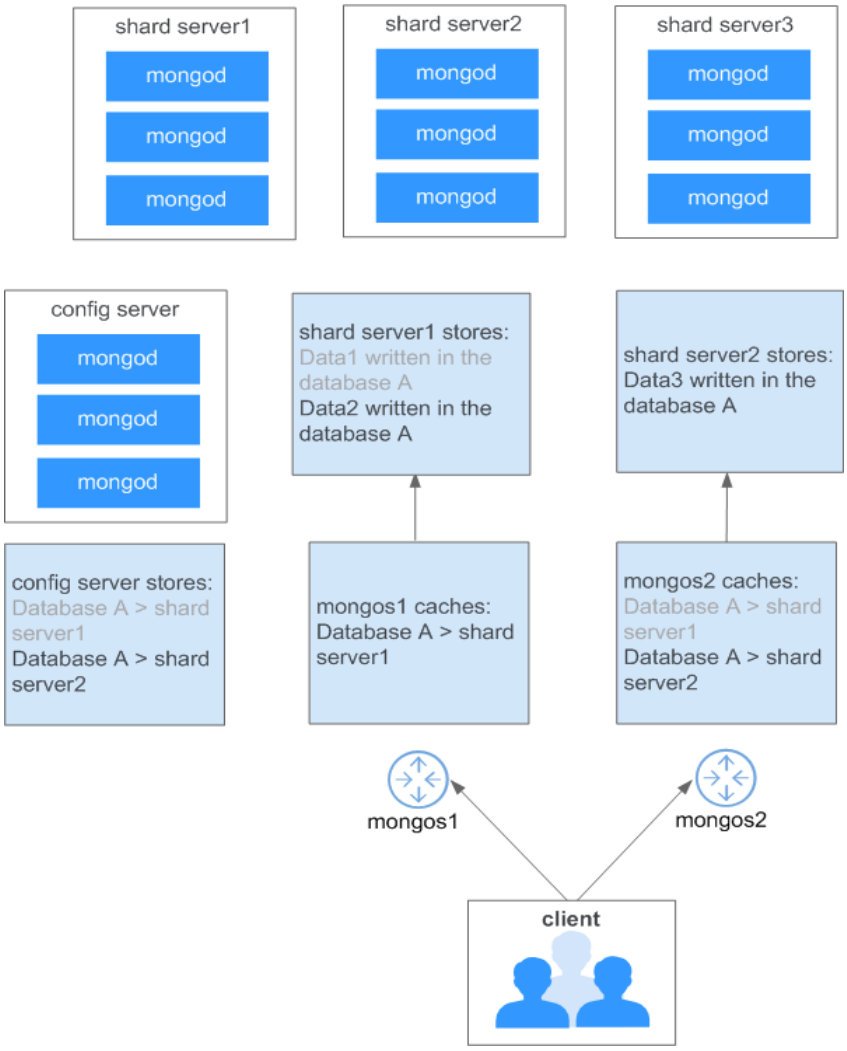
The data to be updated is in the unit of DB.

Scenarios

In the scenario where data is not sharded and multiple mongos exist in a sharded cluster, if data is accessed through different mongos, the cached route information on each mongos may become different. The following shows an example scenario:

1. Create the database A with sharding disabled through mongos1. After data1 is written, data1 is allocated to shard server1 for storage. Then, mongos2 is used to query data. Both mongos1 and mongos2 have cached the route information of the database A.
2. If the deletion operations are performed twice on the database A through mongos2, the information about database A in the config server and shard server1 is deleted. As a result, mongos1 cannot identify the data 1 because it has been deleted.
3. When data2 is written to the database A through the mongos1, data2 will be stored on shard server1 based on the cached route information but actually database A has been deleted. Then, when data3 is written into the database A through the mongos2, new information about the database A will be generated again on the config server and shard server2 because the mongos2 has identified that the database A has been deleted.
4. In this case, the route information cached in the mongos1 and mongos2 is inconsistent. The mongos1 and mongos2 are associated with different shard servers, and only some data can be queried. As a result, the data becomes abnormal.

Figure 5-2 mongos cache defect scenario



The client queries data through different mongos:

- mongos1: Data2 can be queried, but data3 cannot be queried.
- mongos2: Data3 can be queried, but data2 cannot be queried.

Workaround Suggestion

MongoDB official suggestions: After deleting the database, run the **db.adminCommand("flushRouterConfig")** command on all mongos nodes to update the route information.

Reference link: **https://docs.mongodb.com/manual/reference/method/db.dropDatabase/index.html#replica-set-and-sharded-clusters**

https://jira.mongodb.org/browse/SERVER-17397

Workaround Suggestion

- For the cluster mode, you are advised to enable the sharding function and then shard the collections in the cluster.
- If the database with the sharding function disabled is deleted, you are advised not to create a database with the same name.

  If you need to create a database with the same name, log in to all the mongos nodes to update the route information before creating the database.

6 Solving the High CPU Usage Problem

If the CPU usage is high or close to 100% when you use DDS, data read and write will be slow, affecting your services.

This section helps you analyze the requests being processed and the slow queries of the database. After the analysis and optimization, the queries will become reasonable and use indexes efficiently.

NOTE

The CPU resources are often used up in the scenario that the capabilities of the instance may reach the upper limit. You can determine whether or not to change the specifications of the instance or add the number of shards.

Analyzing Requests Being Processed

1. Connect to a DB instance through Mongo Shell.

   For details, see section "Connecting to a DB Instance Through a Public Network" and "Connecting to a DB Instance Through a Private Network" in *Document Database Service Getting Started.*

2. Run the following command to view the current operations on the database:

   db.currentOp()

   Command output:

```
{
	"raw" : {
		"shard0001" : {
			"inprog" : [
				{
					"desc" : "StatisticsCollector",
					"threadId" : "140323686905600",
					"active" : true,
					"opid" : 9037713,
					"op" : "none",
					"ns" : "",
					"query" : {

					},
					"numYields" : 0,
					"locks" : {

					},
					"waitingForLock" : false,
					"lockStats" : {
```



```
                        }
                    },
                    {
                        "desc" : "conn2607",
                        "threadId" : "140323415066368",
                        "connectionId" : 2607,
                        "client" : "172.16.36.87:37804",
                        "appName" : "MongoDB Shell",
                        "active" : true,
                        "opid" : 9039588,
                        "secs_running" : 0,
                        "microsecs_running" : NumberLong(63),
                        "op" : "command",
                        "ns" : "admin.",
                        "query" : {
                            "currentOp" : 1
                      },
                        "numYields" : 0,
                        "locks" : {

                        },
                        "waitingForLock" : false,
                        "lockStats" : {

                        }
                    }
                ],
                "ok" : 1
            },
    ...
}
```

NOTE

- **client**: IP address of the client that sends the request
- **opid**: unique operation ID
- **secs_running**: elapsed time for execution, in seconds If the returned value of this field is too large, check whether the request is reasonable.
- **microsecs_running**: elapsed time for execution, in seconds If the returned value of this field is too large, check whether the request is reasonable.
- **op**: operation type The operations can be query, insert, update, delete, or command.
- **ns**: operation target collection
- For details, see the **db.currentOp()** command in **official document**.

3. Based on the command output, check whether there are requests that take a long time to process.
   - If the CPU usage is low during routine operation and becomes high due to a certain operation, you need to pay attention to the requests that take a long time to execute.
   - If an abnormal request is found, you can find the **opid** corresponding to the request and run the **db.killOp(opid)** command to terminate the request.

Analyzing Slow Queries

On DDS, slow request profiling is enabled by default. The system automatically records the execution of queries that exceed 100 ms to the **system.profile** collection in the corresponding database.

1. Connect to a DB instance through Mongo Shell.

   For details, see section "Connecting to a DB Instance Through a Public Network" and "Connecting to a DB Instance Through a Private Network" in *Document Database Service Getting Started.*

2. Run the following command to enter the specified database (using the **test** database as an example):

   use test

3. Check whether the slow SQL queries are generated in **system.profile**.

   show collections;

   - If the command output contains **system.profile**, slow SQL queries are generated. Go to the next step.

     ```
     mongos> show collections
     system.profile
     test
     ```

   - If the command output does not contain **system.profile**, no slow SQL query is generated. The database does not require slow query analysis.

     ```
     mongos> show collections
     test
     ```

4. Check the slow query logs in the database.

   db.system.profile.find().pretty()

5. Analyze slow query logs to find the cause of the high CPU usage.

   The following is an example of a slow query log. The log shows the request that scans the entire table and 1561632 files without using indexes.

   ```
   {
       "op" : "query",
       "ns" : "taiyiDatabase.taiyiTables$10002e",
       "query" : {
           "find" : "taiyiTables",
           "filter" : {
               "filed19" : NumberLong("852605039766")
           },
           "shardVersion" : [
               Timestamp(1, 1048673),
               ObjectId("5da43185267ad9c374a72fd5")
           ],
           "chunkId" : "10002e"
       },
       "keysExamined" : 0,
       "docsExamined" : 1561632,
       "cursorExhausted" : true,
       "numYield" : 12335,
       "locks" : {
           "Global" : {
               "acquireCount" : {
                   "r" : NumberLong(24672)
               }
           },
           "Database" : {
               "acquireCount" : {
                   "r" : NumberLong(12336)
               }
           },
           "Collection" : {
               "acquireCount" : {
                   "r" : NumberLong(12336)
               }
           }
       },
       "nreturned" : 0,
   ```



```
        "responseLength" : 157,
        "protocol" : "op_command",
        "millis" : 44480,
        "planSummary" : "COLLSCAN",
        "execStats" : {
            "stage" :
"SHARDING_FILTER",
             [3/1955]
             "nReturned" : 0,
             "executionTimeMillisEstimate" : 43701,
             "works" : 1561634,
             "advanced" : 0,
             "needTime" : 1561633,
             "needYield" : 0,
             "saveState" : 12335,
             "restoreState" : 12335,
             "isEOF" : 1,
             "invalidates" : 0,
             "chunkSkips" : 0,
             "inputStage" : {
                    "stage" : "COLLSCAN",
                    "filter" : {
                           "filed19" : {
                                  "$eq" : NumberLong("852605039766")
                           }
                    },
                    "nReturned" : 0,
                    "executionTimeMillisEstimate" : 43590,
                    "works" : 1561634,
                    "advanced" : 0,
                    "needTime" : 1561633,
                    "needYield" : 0,
                    "saveState" : 12335,
                    "restoreState" : 12335,
                    "isEOF" : 1,
                    "invalidates" : 0,
                    "direction" : "forward",
                    "docsExamined" : 1561632
             }
        },
        "ts" : ISODate("2019-10-14T10:49:52.780Z"),
        "client" : "172.16.36.87",
        "appName" : "MongoDB Shell",
        "allUsers" : [
             {
                    "user" : "__system",
                    "db" : "local"
             }
        ],
       "user" : "__system@local"
}
```

Pay attention to the following keywords in slow query logs:

- Full collection (full table) scan: **COLLSCAN**

  When a request (such as query, update, and delete) requires a full table scan, a large amount of CPU resources are occupied. If the slow query log contains the keyword **COLLSCAN**, the queries may occupy CPU resources.

  If such requests are frequent, you are advised to create indexes for the fields to be queried.

- Full collection (full table) scan: **docsExamined**

  You can view the value of **docsExamined** to check the number of scanned files. A larger value indicates a higher CPU usage.

- Inappropriate index: **IXSCAN** and **keysExamined**

NOTE

An excessive number of indexes can affect the write and update performance.

If your application has more write operations, creating indexes may increase write latency.

You can view the value of **keyExamined** to check the number of indexes that are scanned in a query. A larger value indicates a higher CPU usage.

If the index is not proper or there are many matching results, the CPU usage may not stay low and the execution is slow.

Example: For the data of a collection, the number of values of the **a** field is small (only **1** and **2**), but the **b** field has many values.

```
{a: 1, b: 1}
{a: 1, b: 2}
{a: 1, b: 3}
......
{ a: 1, b: 100000}
{a: 2, b: 1}
{a: 2, b: 2}
{a: 2, b: 3}
......
{a: 1, y: 100000}
```

The following shows how to implement the {a: 1, b: 2} query.

```
db.createIndex({a: 1}): The query is not effective because the a field has too many same values.
db.createIndex({a: 1, b: 1}): The query is not effective because the a field has too many same values.
db.createIndex({b: 1}): The query is effective because the b field has a few same values.
db.createIndex({b: 1, a: 1}): The query is not effective because the a field has a few same values.
```

For the differences between {a: 1} and {b: 1, a: 1}, see the **official documents**.

- Sorting a large amount of data: **SORT** and **hasSortStage**

  If the query request involves sorting, the value of the **hasSortStage** parameter in the **system.profile** collection is **true**. If the sorting cannot be implemented through indexes, the query results are sorted. Sorting occupies a large amount of CPU resources. In this scenario, you need to create indexes for fields that are frequently sorted.

  If the **system.profile** collection contains **SORT**, you can use indexes to improve the sorting.

Other operations, such as index creation and aggregation (combination of traversal, query, update, and sorting), also apply to the above mentioned scenarios because they also occupy a large amount of CPU resources. For more information about profiling, see **official documents**.

7 Connecting to a DB Instance Through an EIP

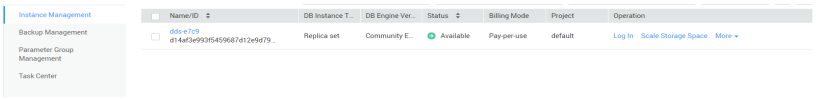
This section uses a DDS replica set instance and Windows operating system as an example to describe how to bind an EIP on the DDS console, set a security group, and connect to the DDS DB instance using the Robo 3T tool in your local environment. The procedures are as follows:

- **Step 1: Bind an EIP**
- **Step 2: Set a Security Group**
- **Step 3: Connect to a DDS DB Instance**

Step 1: Bind an EIP

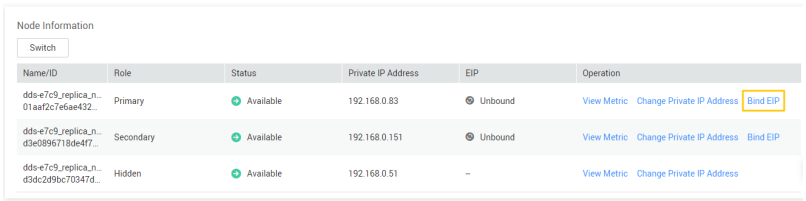
1. Log in to the **management console**.
2. On the **Instance Management** page, click the target DB instance. The **Basic Information** page is displayed.

   Figure 7-1 Instance management

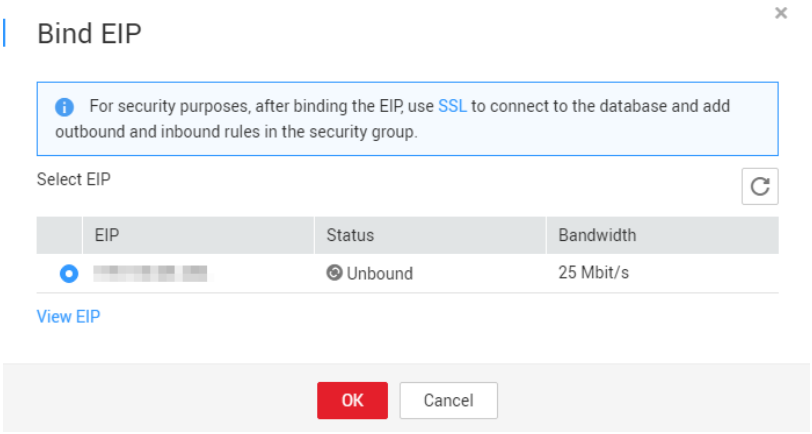
3. In the **Node Information** area, locate the row that contains the primary node and click **Bind EIP**.

   Figure 7-2 Node information

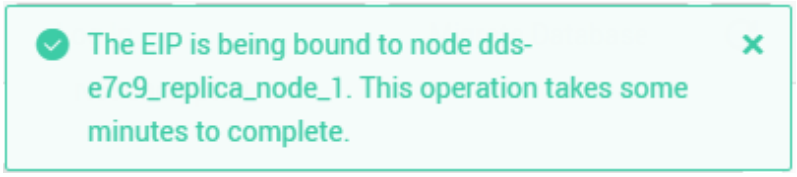
4. In the displayed dialog box, select a purchased EIP and click **OK**.

Figure 7-3 Binding an EIP



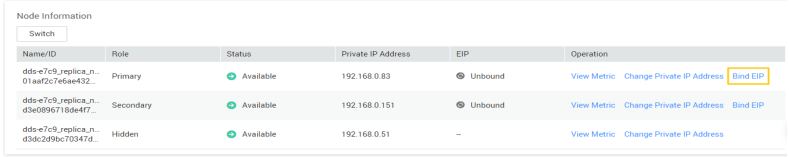
A message in the upper right corner is displayed, indicating the request is successfully issued.

Figure 7-4 EIP being bound



5. After the binding is successful, view the EIP in the **Node Information** area.

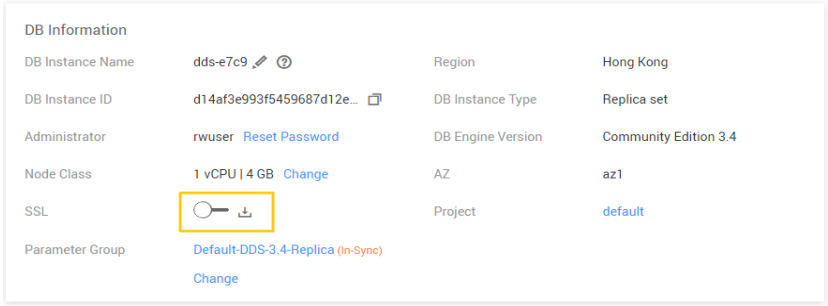
Figure 7-5 EIP bound successfully



Step 2: Set a Security Group

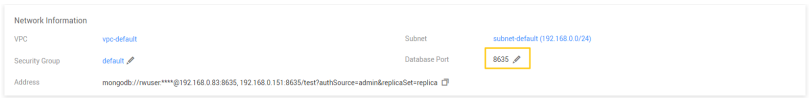
1. In the **DB Information** area on the **Basic Information** page, verify that SSL is disabled.

Figure 7-6 Database information



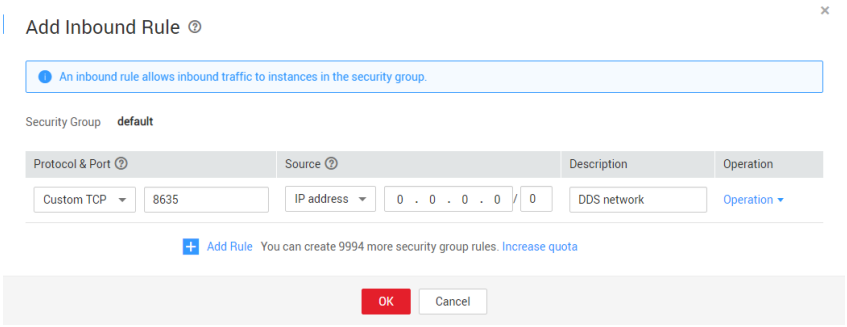
2. In the **Network Information** area, check the database port of the DB instance.

Figure 7-7 Network information



3. In the **Network Information** area, click the security group name.
4. On the **Security Groups** page, click the security group name.
5. Click the **Inbound Rules** tab and click **Add Rule**. In the displayed dialog box, add an inbound rule for the database port.

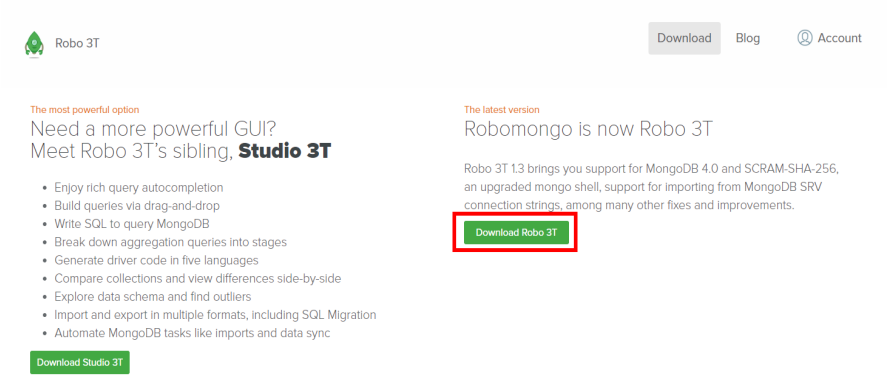
Figure 7-8 Add Inbound Rule



Step 3: Connect to a DDS DB Instance

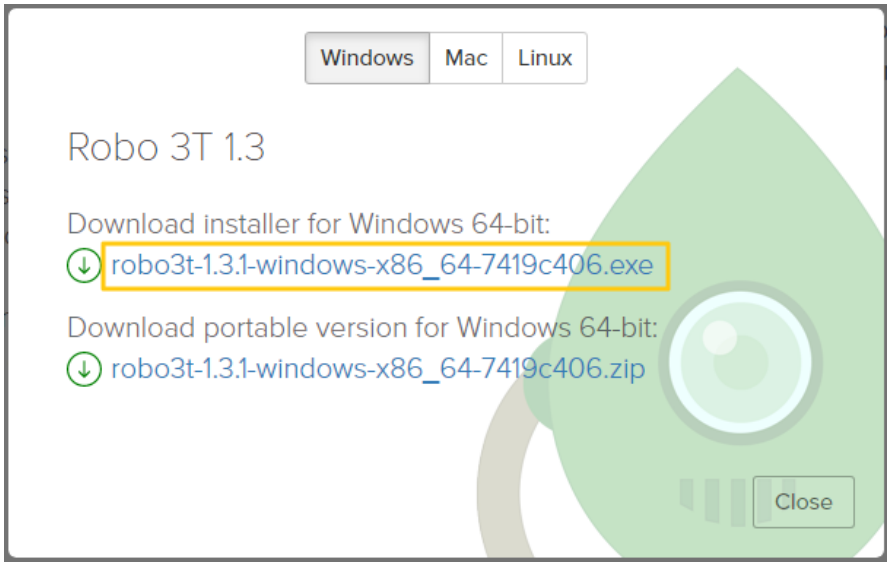
1. Download Robo 3T from **https://robomongo.org/download**.

Figure 7-9 Downloading Robo 3T



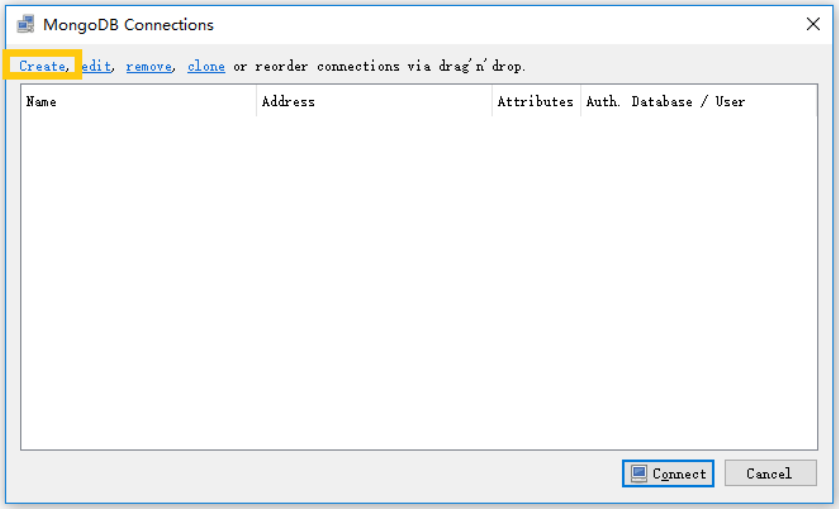
2. In the displayed dialog box, download **robo3t-1.3.1-windows-x86_64-7419c406.exe**.

Figure 7-10 Downloading Robo 3T



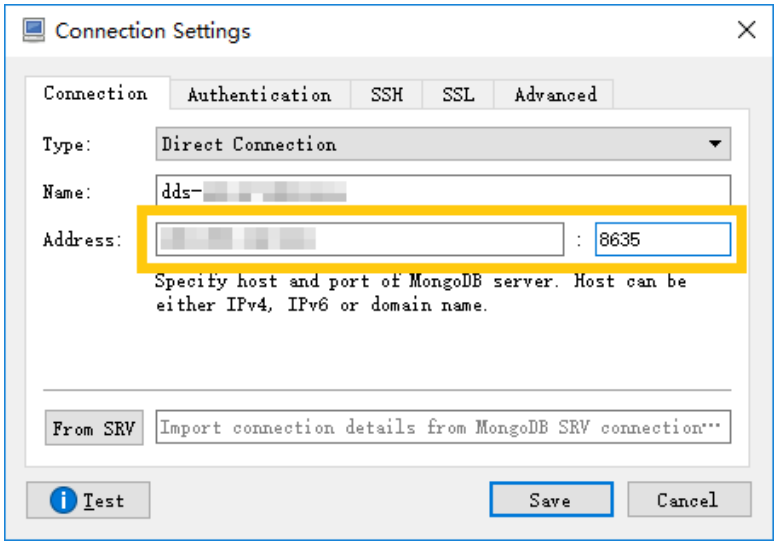
3. Double-click the **robo3t-1.3.1-windows-x86_64-7419c406.exe** file to start the installation.
4. Run the installed Robo 3T. On the connection information page, click **Create**.

Figure 7-11 Connection information



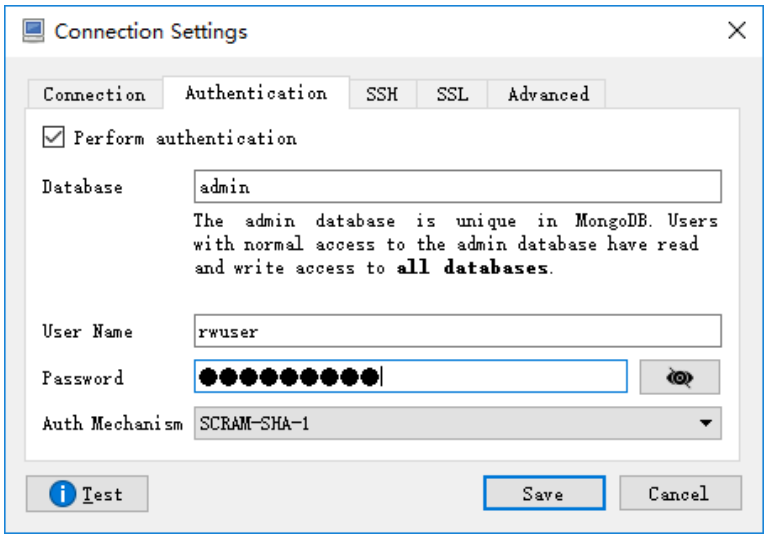
5. In the **Connection Settings** dialog box, set the parameters of the new connection.
   a. On the **Connection** tab, set **Name** to the name of the new connection and **Address** to the EIP and database port bound to the DDS DB instance.

Figure 7-12 Connection



b. On the **Authentication** tab, set **Database** to **admin**, **User Name** to **rwuser**, and **Password** to the administrator password you set during the creation of the DDS DB instance.

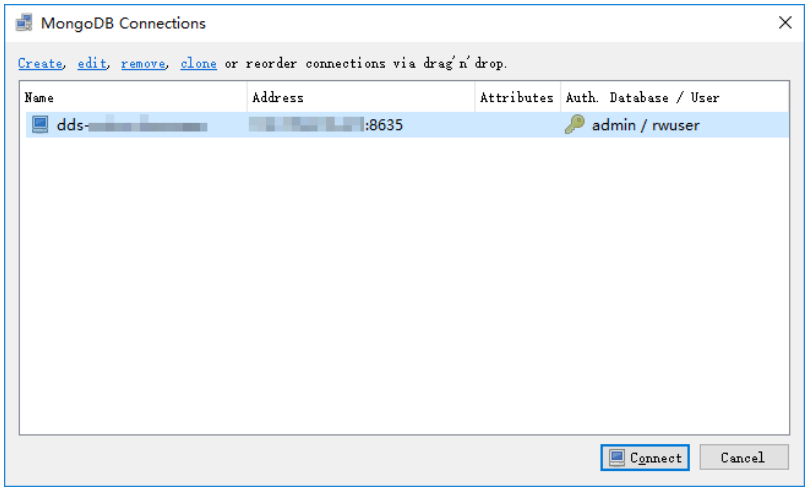
Figure 7-13 Authentication



c. Click **Save**.

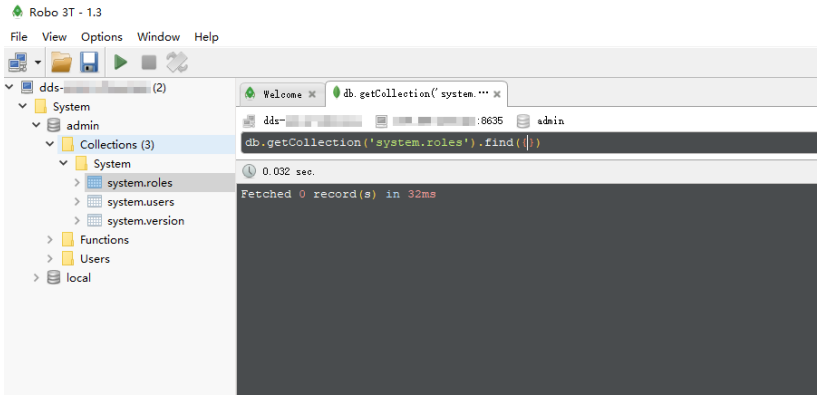
6. Click **Connect** to connect to the DDS DB instance.

Figure 7-14 Connection information



7. If the DDS DB instance is successfully connected, the following information is displayed:

Figure 7-15 Connection succeeded



8 Connecting to a DB Instance Through an ECS

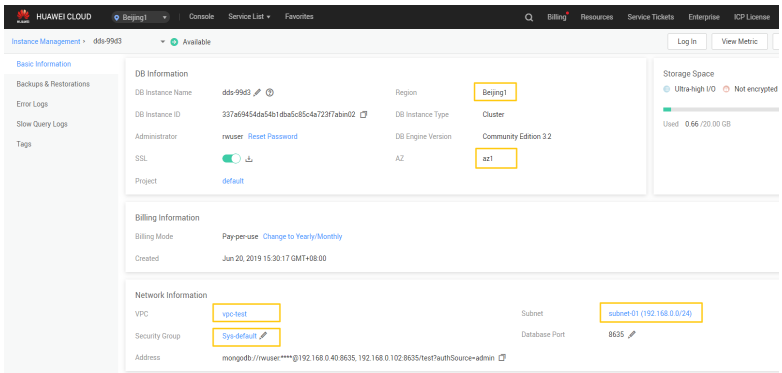
This section uses a DDS cluster instance and a Linux ECS as an example to describe how to connect to a DDS DB instance by logging in to an ECS with an EIP assigned using SSH. The procedures are as follows:

- **Step 1: Create an ECS**
- **Step 2: Log in to the ECS**
- **Step 3: Connect to a DDS DB Instance**

Step 1: Create an ECS

1. Log in to the **management console**.
2. Choose **Computing** > **Elastic Cloud Server**. On the ECS console, check whether there is an ECS meeting the following requirements:
   a. The region, VPC subnet, and security group of the ECS are the same as those of the DDS DB instance to be connected.

   Figure 8-1 DB instance basic information

   b. The ECS has had an EIP bound and the EIP meets the bandwidth requirements.

   – If an ECS that meets the preceding conditions is available, go to **Step 2: Log in to the ECS**.

- If you need to create an ECS, go to the next step.

3. On the page for buying ECSs, select the region, AZ, VPC subnet, and security group that are the same as those of the DDS DB instance to be connected.

Figure 8-2 Buying an ECS

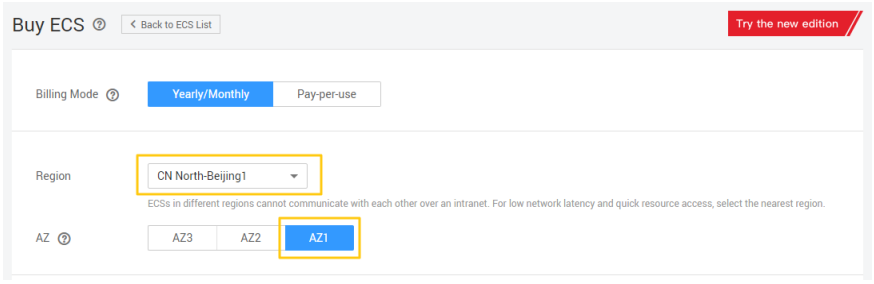
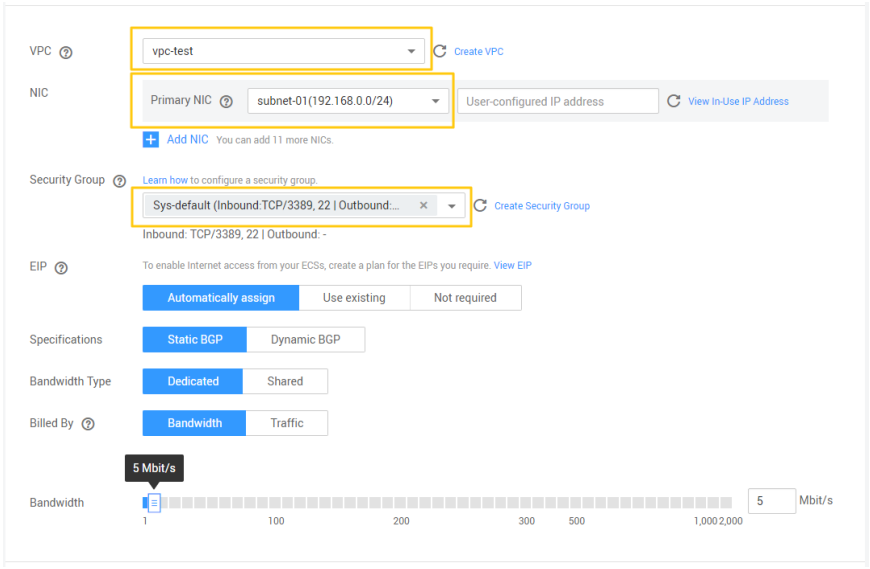


Figure 8-3 Network



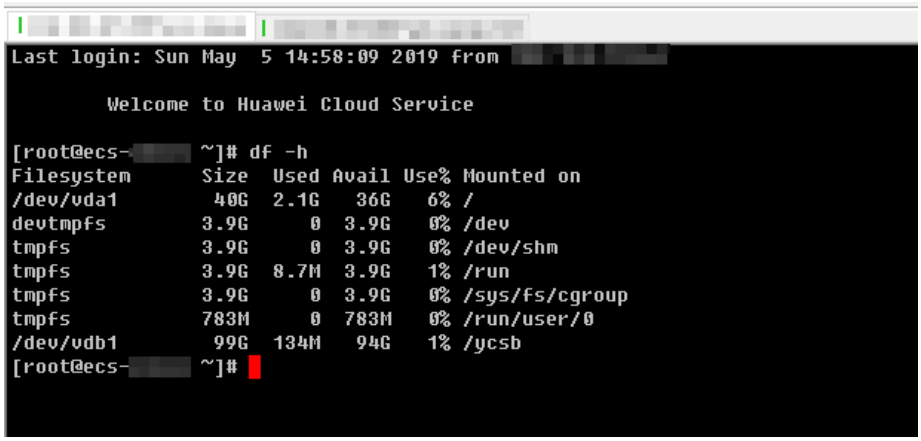
4. If you select **Automatically assign**, the system automatically assigns an EIP for the ECS. The EIP provides dedicated bandwidth that is configurable.

   For more details, see **Methods of Purchasing ECSs**.

Step 2: Log in to the ECS

Obtain the EIP bound to the ECS, and use the client tool to log in to the ECS in SSH mode.

Figure 8-4 Log in to the ECS.



For more details, see **Login Overview**.

Step 3: Connect to a DDS DB Instance

1. Create a file on the ECS to install mongo shell. The following uses the file **/ycsb** as an example.

   mkdir /ycsb

2. Open the **/ycsb** file and create the **mongo** directory.

   cd /ycsb

   mkdir mongo

   cd /ycsb/mongo

3. Download mongo shell.

   curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.4.17.tgz

4. Decompress the mongo shell package.

   tar -zxvf mongodb-linux-x86_64-3.4.17.tgz

5. Move the decompressed file to an existing directory, for example, **/usr/local/mongodb**.

   mv mongodb-linux-x86_64-3.4.17/* /usr/local/mongodb

6. Set environment variables.

   vi /etc/profile

   Add the following two lines to the end of the file and save the file:

   ```
   export MONGODB_HOME=/usr/local/mongodb
   export PATH=$PATH:$MONGODB_HOME/bin
   ```


Figure 8-5 Setting environment variables

```
[root@ecs-          ]# tail /etc/profile
            . "$i" >/dev/null
        fi
    fi
done

unset i
unset -f pathmunge
export MONGODB_HOME=/usr/local/mongodb
export PATH=$PATH:$MONGODB_HOME/bin
```

7. Make the configurations take effect.

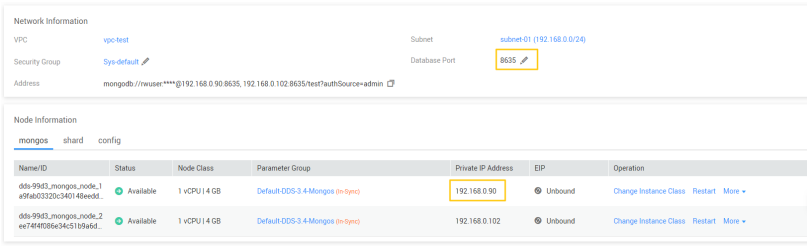
   source /etc/profile

8. Connect to the cluster instance using mong shell.

   mongo 192.168.0.90:8635 --authenticationDatabase admin -u rwuser -p *xxxxx*

   - The proceeding command is used to disable the SSL connection for a cluster instance and connect to a single mongos node.
   - **192.168.0.90** indicates the private IP address. Obtain the value from the **Private IP Address** column in the **Node Information** area on the **Basic Information** page of the DB instance.
   - **8635** indicates the port number. Obtain the value from the **Database Port** field in the **Network Information** area on the **Basic Information** page.
   - **rwuser** indicates the administrator account.
   - **xxxxx** indicates the password of the administrator.

   Figure 8-6 Connection information



9. If the DDS DB instance is successfully connected, the following information is displayed:

```
[root@ecs-dds-mongos~]# mongo 192.168.0.90:8635 --authenticationDatabase admin -u rwuser -p xxxxx
MongoDB shell version: 3.4.17
connecting to: 192.168.0.90:8635/test
mongos>
```

