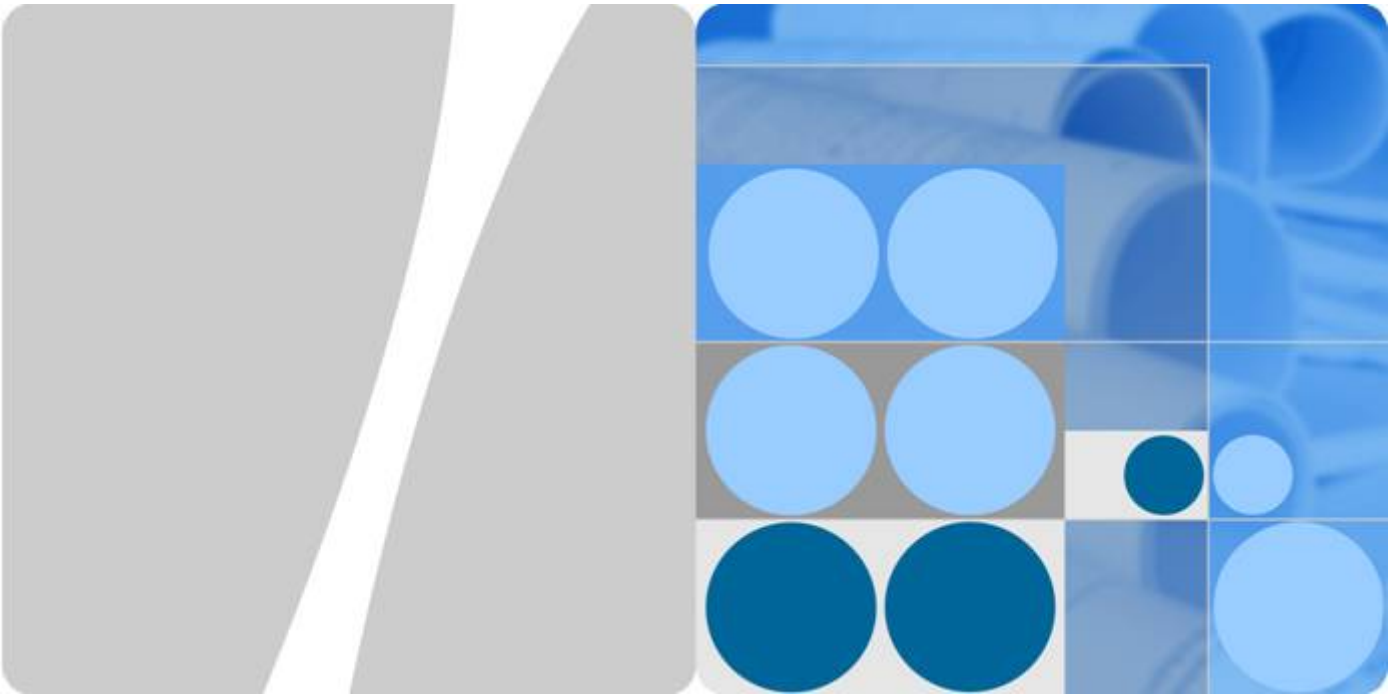


Document Database Service

Best Practices

Issue 01

Date 2025-11-14

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 Overview

This document provides best practices for Document Database Service (DDS) and guides you through using DDS to best suit your business needs.

Service	Reference	Overview
Document Database Service	**Common Methods for Connecting to a DDS Instance**	This section describes common DDS connection methods.
	How Do Replica Sets Achieve High Availability and Read/Write Splitting?	This section describes how to connect to a replica set instance to achieve high availability.
	Sharding	This section describes how to set cluster shards to improve database performance.
	How Do I Improve DDS Performance by Optimizing SQL Statements?	This section describes DDS usage suggestions.
	How Do I Prevent the dds mongos Cache Problem?	This section describes how to avoid the mongos route cache defect of the cluster.
	How Do I Solve the High CPU Usage Issue?	This section describes how to troubleshoot high CPU usage.
	Creating a User and Granting the Read-Only Permission to the User	This section describes how to use IAM to grant read-only permissions to DDS.
	How Is a DDS Node Going to Be Disconnected and What Can I Do?	This section describes the principles and workarounds of node disconnection.

Service	Reference	Overview
	Avoiding Cursor Invalidity Caused by hideIndex	This section describes how to prevent a cursor from becoming invalid due to hideIndex.
	Using DDS to Store and Analyze Log Data	This section describes how to use DDS to store and analyze application log data.
	DDS Query Plans and Query Replanning	This section describes the principles and scenarios of query plans and query replanning.
	DDS Transactions and Read/Write Concerns	This section describes transactions and read/write concerns.
	DDS Metric Alarm Configuration Suggestions	This section describes suggestions on configuring alarm rules of DDS metrics.
	Working with Indexes	This section describes suggestions on working with indexes.

2 Common Methods for Connecting to a DDS Instance

This section describes how to connect to a DDS instance using the following four methods:

- Mongo Shell
- Python Mongo
- Java Mongo
- Using Spring MongoTemplate to Perform MongoDB Operations

Mongo Shell

- Prerequisites
  a. To connect an ECS to a DDS instance, 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.
  b. Download the mongo shell package from the **MongoDB official website**. Decompress the package, obtain the **mongosh** file, and upload it to the ECS.
  c. If SSL is enabled, download the root certificate and upload it to the ECS.
- Connection commands
  – SSL is enabled.

    Method 1: **./mongosh** *ip:port* **--authenticationDatabase admin -u** *username* **-p** *password* **--ssl --sslCAFile $***path to certificate authority file* **--sslAllowInvalidHostnames**

    Method 2: **./mongosh "mongodb://***<username* >:<password>@*ip:port*/test?authSource=admin" **--ssl --sslCAFile $***path to certificate authority file* **--sslAllowInvalidHostnames**
  – SSL is disabled.

Method 1: **./mongosh** *ip:port* **--authenticationDatabase admin -u** *username* **-p** *password*

Method 2: **./mongosh "mongodb://***<username>*:<password>@*ip:port***/test?authSource=admin"**

Table 2-1 Parameter description

Parameter	Description
ip	If you access an instance from an ECS, *ip* is the private IP address of the instance.
	If you access an instance from a device over a public network, *ip* is the EIP bound to the instance.
port	Database port displayed on the **Basic Information** page. Default value: **8635**
username	Current username
password	Password for the current username. In the connection method 2, when connecting to a DDS instance, escape the at sign (@), percent sign (%), and exclamation mark (!) and replace them with hexadecimal URL codes (ASCII codes) %40, %25, and %21, respectively.
path to certificate authority file	Path of the SSL certificate

- Precautions
  a. If SSL is enabled, the connection command must contain **--ssl** and **--sslCAFile**.
  b. **--authenticationDatabase** must be set to **admin**. If you log in to the database as user **rwuser**, switch to **admin** for authentication.

For details, see **Connecting to an Instance** in *Getting Started with Document Database Service*.

Python Mongo

- Prerequisites
  a. To connect an ECS to a DDS instance, 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, download the root certificate and upload it to the ECS.

- Input the connection code.
  - SSL is enabled.

```
import ssl
import os
from pymongo import MongoClient
# There will be security risks if the username and password used for authentication are directly written into code. Store the username and password in ciphertext in the configuration file or environment variables.
# In this example, the username and password are stored in the environment variables. Before running this example, set environment variables EXAMPLE_USERNAME_ENV and EXAMPLE_PASSWORD_ENV as needed.
rwuser = os.getenv('EXAMPLE_USERNAME_ENV')
password = os.getenv('EXAMPLE_PASSWORD_ENV')
conn_urls="mongodb://%s:%s@ip:port/{mydb}?authSource=admin"
connection = MongoClient(conn_urls % (rwuser,
password),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
```

  - SSL is disabled.

```
import ssl
import os
from pymongo import MongoClient
# There will be security risks if the username and password used for authentication are directly written into code. Store the username and password in ciphertext in the configuration file or environment variables.
# In this example, the username and password are stored in the environment variables. Before running this example, set environment variables EXAMPLE_USERNAME_ENV and EXAMPLE_PASSWORD_ENV as needed.
rwuser = os.getenv('EXAMPLE_USERNAME_ENV')
password = os.getenv('EXAMPLE_PASSWORD_ENV')
conn_urls="mongodb://%s:%s@ip:port/{mydb}?authSource=admin"
connection = MongoClient(conn_urls % (rwuser, password),connectTimeoutMS=5000)
dbs = connection.database_names()
print "connect database success! database names is %s" % dbs
```

- Precautions

  a. *{mydb}* is the name of the database to be connected.

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

Java Mongo

- How to Use

  If you are connecting to an instance using Java, 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 newly created instances, but you can enable SSL by referring to **Enabling or Disabling SSL**. SSL encrypts connections to databases but it increases the connection response time and CPU usage. For this reason, enabling SSL is not recommended.

- Prerequisites

  You should be familiar with:

  - Computer basics
  - Java
- Obtaining and Using Java
  - Download the Jar driver from: **https://repo1.maven.org/maven2/org/mongodb/mongo-java-driver/3.0.4/**
  - To view the usage guide, visit **https://mongodb.github.io/mongo-java-driver/4.2/driver/getting-started/installation/**.
- Connecting to the Instance with an SSL Certificate

  NOTE

  - Download the SSL certificate and verify the certificate before connecting to databases.
  - On the **Instances** page, click the target DB instance name. In the **DB Information** area on the **Basic Information** page, click in the **SSL** field to download the root certificate or certificate bundle.
  - For details about the SSL connection guide, see the MongoDB Java Driver official document at **https://www.mongodb.com/docs/drivers/java/sync/v5.0/fundamentals/connection/tls/**.
  - Java Runtime Environment (JRE) earlier than Java 8 enables TLS 1.2 only in updated versions. If TLS 1.2 is not enabled for your JRE, upgrade it to a later version to use TLS 1.2 for connection.

  If you connect to a cluster instance using Java, the format of code is as follows:

```
mongodb://<username>:<password>@<instance_ip>:<instance_port>/<database_name>?authSource=admin&ssl=true
```

Table 2-2 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 shown on the **Basic Information** page of the DB instance.
	If you access an instance through an EIP, *instance_ip* is the EIP that has been bound to the instance.
	If there are multiple IP addresses, list the addresses in the format of <instance_ip1>:<instance_port1>,<instance_ip2>:<instance_port2>..... Example: mongodb://username:*****@127.***.***.1:8635,127.***.***.2:8635/?authSource=admin
<instance_port>	Database port displayed on the **Basic Information** page. Default value: **8635**
<database_name>	Name of the database to be connected.

Parameter	Description
authSource	Authentication database. The value is **admin**.
ssl	Connection mode. **true** indicates that SSL will be used.

Use the keytool to configure the CA certificate. For details about the parameters, see **Table 2-3**.

```
keytool -importcert -trustcacerts -file <path to certificate authority file> -keystore <path to trust store> -storepass <password>
```

Table 2-3 Parameter description

Parameter	Description
<path to certificate authority file>	Path for storing the SSL certificate.
<path to trust store>	Path for storing the truststore. Set this parameter as required, for example, **./trust/certs.keystore**.
<password>	Custom password.

Set the JVM system properties in the program to point to the correct truststore and keystore:

- System.setProperty("javax.net.ssl.trustStore","<path to trust store>");
- System.setProperty("javax.net.ssl.trustStorePassword","<password>");

The following shows an example:

```
public class Connector {
    public static void main(String[] args) {
        try {
            System.setProperty("javax.net.ssl.trustStore", "./trust/certs.keystore");
            System.setProperty("javax.net.ssl.trustStorePassword", "123456");
            ConnectionString connString = new ConnectionString("mongodb://
<username>:<password>@<instance_ip>:<instance_port>/<database_name>?
authSource=admin&ssl=true");
            MongoClientSettings settings = MongoClientSettings.builder()
                    .applyConnectionString(connString)
                    .applyToSslSettings(builder -> builder.enabled(true))
                    .applyToSslSettings(builder -> builder.invalidHostNameAllowed(true))
                    .build();
            MongoClient mongoClient = MongoClients.create(settings);
            MongoDatabase database = mongoClient.getDatabase("admin");
            //Ping the database. If the operation fails, an exception occurs.
            BsonDocument command = new BsonDocument("ping", new BsonInt64(1));
            Document commandResult = database.runCommand(command);
            System.out.println("Connect to database successfully");
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Test failed");
        }
    }
}
```

- Connecting to the Instance Without an SSL Certificate

  NOTE

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

  If you connect to a cluster instance using Java, the format of code is as follows:

  ```
  mongodb://<username>:<password>@<instance_ip>:<instance_port>/<database_name>?authSource=admin
  ```

  Table 2-4 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 shown on the **Basic Information** page of the DB instance.
	If you access an instance through an EIP, *instance_ip* is the EIP that has been bound to the instance.
	If there are multiple IP addresses, list the addresses in the format of <instance_ip1>:<instance_port1>,<instance_ip2>:<instance_port2>..... Example: mongodb://username:*****@127.***.***.1:8635,127.***.***.2:8635/?authSource=admin
<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**.

  The following shows an example:

  ```
  public class Connector {
      public static void main(String[] args) {
          try {
              ConnectionString connString = new ConnectionString("mongodb://<username>:<password>@<instance_ip>:<instance_port>/<database_name>?authSource=admin");
              MongoClientSettings settings = MongoClientSettings.builder()
                      .applyConnectionString(connString)
                      .retryWrites(true)
                      .build();
              MongoClient mongoClient = MongoClients.create(settings);
              MongoDatabase database = mongoClient.getDatabase("admin");
              //Ping the database. If the operation fails, an exception occurs.
              BsonDocument command = new BsonDocument("ping", new BsonInt64(1));
              Document commandResult = database.runCommand(command);
  ```



```
            System.out.println("Connect to database successfully");
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Test failed");
        }
    }
}
```

Using Spring MongoTemplate to Perform MongoDB Operations

- How to Use

  The following describes how to use Spring MongoTemplate to perform operations on MongoDB. For details, visit the **MongoDB official website**.

- Prerequisites

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-starter-logging</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

- Configuration Guide

```
spring:
  data:
    mongodb:          #MongoDB configuration, which is for reference only
      // There will be security risks if the username and password used for authentication are directly written into code. Store the username and password in ciphertext in the configuration file or environment variables.
      // In this example, the username and password are stored in the environment variables. Before running this example, set environment variables EXAMPLE_USERNAME_ENV and EXAMPLE_PASSWORD_ENV as needed.
      String userName = System.getenv("EXAMPLE_USERNAME_ENV");
      String rwuserPassword = System.getenv("EXAMPLE_PASSWORD_ENV");
      uri: mongodb://" + userName + ":" + rwuserPassword + "@192.***.***.***:8635,192.***.***.***:8635/${mongodb.database}
      database: ${mongodb.database}
```

- Development Guide

```
/**
 * MongoDB execution
 */
@Autowired
private MongoTemplate template;

/**
 * Log configuration
 */
@Autowired
private LoggingProperties properties;

@Override
public void write(BaseLog businessLog, LoggingOption option) {
    if (template != null) {
        LoggingConfig config = properties.getBusinessConfig(businessLog.getCategory());
        String collection = config.getMeta().get("collection");
        if (StringUtils.isNotEmpty(collection)) {
            Object data = mapping(businessLog, config);
            template.save(data, collection);
```

```
            if (log.isDebugEnabled()) {
                log.debug("save audit log to mongodb successfully!, message: {}",
StringEscapeUtils.escapeJava(TransformUtil.toJsonByJackson(businessLog)));
            }
        } else {
            log.warn("mongo log write log failed, mongoconfig is null");
        }
    } else {
        log.warn("mongo log write log failed, mongoTemplate is null");
    }
}
```

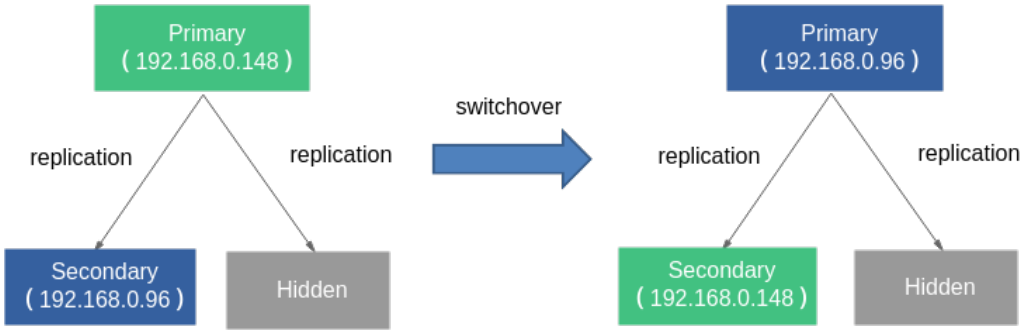
- Precautions
  a. In SSL mode, you need to manually generate the trustStore file.
  b. Change the authentication database to **admin**, and then switch to the service database after authentication.

3 How Do Replica Sets Achieve High Availability and Read/Write Splitting?

DDS replica set instances can store multiple duplicates to ensure data high availability and support the automatic switch of private IP addresses to ensure service high availability. To enhance the read and write performance of your client for connecting to the instance, you can use your client to read different data copies. You need to connect to replica set instances using HA connection addresses. You can also configure read/write splitting. Otherwise, the high availability and high read performance of replica set instances 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 3-1 Primary/Secondary switchover



Connecting to a Replica Set Instance (HA)

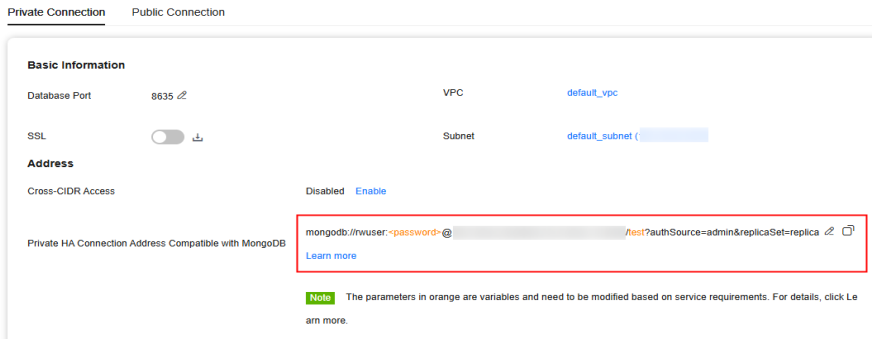
A DDS replica set consists of the primary, secondary, and hidden nodes. The hidden node is invisible to users. Read/Write splitting and HA can be realized only when you connect to the IP addresses and ports of the primary and secondary nodes of the replica set at the same time (in HA mode).

The following describes how to use URL and Java to connect to an instance in HA mode.

Method 1: Using a URL

On the **Instances** page, click the instance name. The **Basic Information** page is displayed. Choose **Connections**. Click the **Private Connection** tab and obtain the connection address of the current instance from the **Private HA Connection Address** field.

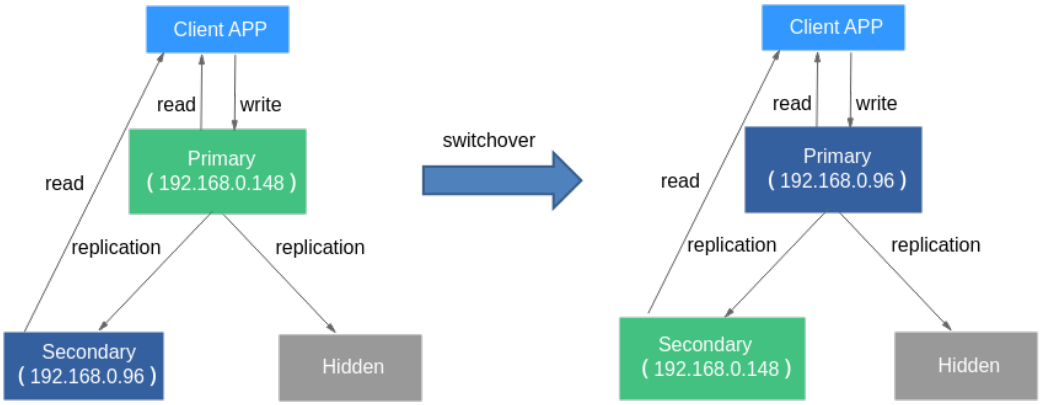
Figure 3-2 Obtaining the private HA connection address



Example: **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** are the IP addresses and ports of the primary and secondary nodes, respectively. If you use this address, the connection between your client and the instance can be ensured even when a primary/standby switchover occurs. In addition, using multiple IP addresses and port numbers can enhance the read and write performance of the entire database.

Figure 3-3 Data read and write process



Method 2: Using a Java Driver

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");
```

Table 3-1 Parameter description

Parameter	Description
rwuser:****	Username and password for starting authentication
192.168.0.148:8635, 192.168.0.96:8635	IP addresses and ports of the primary and secondary nodes in a replica set instance
test	Name of the database to be connected
authSource=admin	Database username for authentication
replicaSet=replica	Name of the replica set instance type

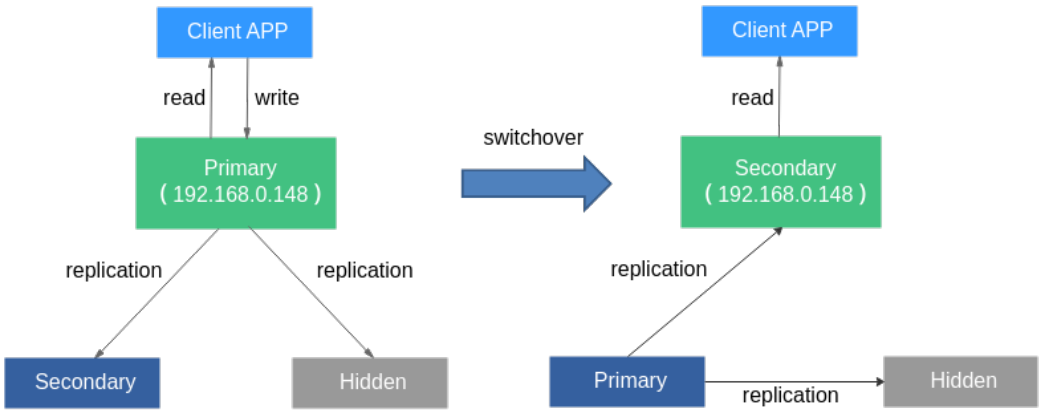
(Not Recommended) Connecting to a Replica Set Instance

Using the Connection Address

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

In the preceding URL, **192.168.0.148:8635** is 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 only be performed on a fixed primary node, so the read and write performance cannot be improved by adding nodes.

Figure 3-4 Data read and write process



Read/Write Splitting

The following HA connection address is used as an example to describe how to connect to a DDS replica set instance:

```
mongodb://rwuser:<password>@192.168.xx.xx:8635,192.168.xx.xx:8635/test?
authSource=admin&replicaSet=replica&readPreference=secondaryPreferred
```

The database account is **rwuser**, and the database is **admin**.

NOTE

After the DB instance is connected, read requests are preferentially sent to the secondary node to implement read/write splitting. If the relationship between the primary and secondary nodes changes, write operations are automatically switched to the new primary node to ensure high availability of DDS.

4 Sharding

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.

Number of Shards

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

Step 1 Log in to a sharded cluster instance using Mongo Shell.

Step 2 Check whether a collection has been sharded.

```
use <database>
db.<collection>.getShardDistribution()
```

Example:

```
use mytable
db.mycoll.getShardDistribution()
```

```
mongos> db.mycoll.getShardDistribution()
Collection test.mycoll is not sharded.
```

Step 3 Enable sharding for the databases that belong to the cluster instance.

- Method 1

  ```
  sh.enableSharding("<database>")
  ```

  Example:

  ```
  sh.enableSharding("mytable")
  ```

- Method 2

  ```
  use admin
  db.runCommand({enablesharding:"<database>"})
  ```

Step 4 Shard a collection.

- Method 1

  ```
  sh.shardCollection("<database>.<collection>",{"<keyname>":<value> })
  ```

  Example:

  ```
  sh.shardCollection("mytable.mycoll",{"name":"hashed"},false,{numInitialChunks:5})
  ```


- Method 2

```
use admin
db.runCommand({shardcollection:"<database>.<collection>",key:{"keyname":<value> }})
```

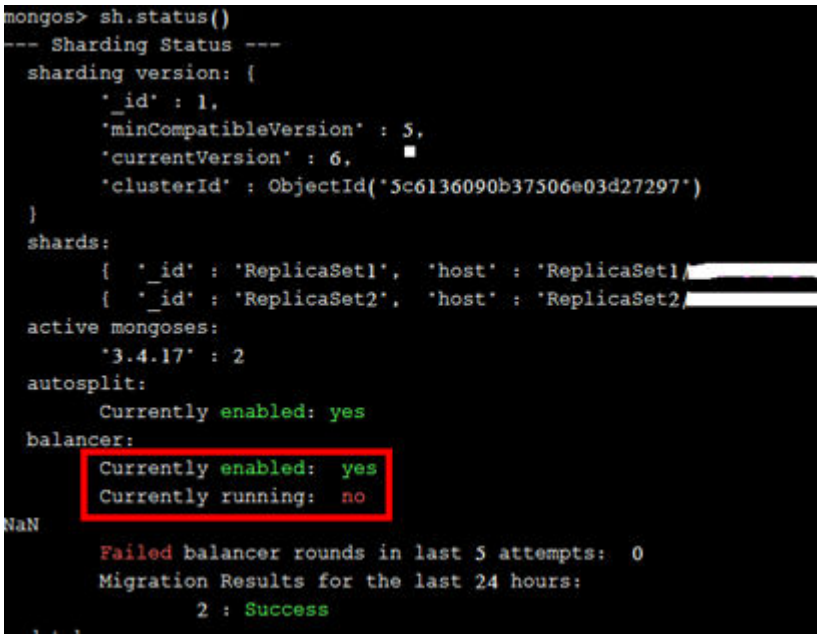
Table 4-1 Parameter description

Parameter	Description
<database>	Database name
<collection>	Collection name.
<keyname>	Shard key. Cluster instances are sharded based on the value of this parameter. Select a proper shard key for the collection based on your service requirements. For details, see **Selecting a Shard Key**.
<value>	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()**.
numInitialChunks	Optional. The minimum number of shards initially created is specified when an empty collection is sharded using a hashed shard key.

Step 5 Check the data storage status of the database on each shard.

```
sh.status()
```

Example:



----End

Selecting a Shard Key

- **Background**

  Each sharded cluster 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.

  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.

- **Characteristics of proper shard keys**
  - All inserts, updates, and deletes are evenly distributed to all shards in a 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 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.

  After existing data is sharded, if the **filter** field of the update request does not contain shard keys and **upsert:true** or **multi:false**, the update request will report an error and return message "An upsert on a sharded collection must contain the shard key and have the simple collation.".

- **Judgment criteria**

You can use the dimensions provided in **Table 4-2** to determine whether the selected shard keys meet your service requirements:

Table 4-2 Reasonable shard keys

Identification Criteria	Description
Cardinality	Cardinality refers to the capability of dividing chunks. For example, if you need to record the student information of a school and use the age as a shard key, data of students of the same age will be stored in only one data segment, which may affect 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 ranged 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 a cluster to fully utilize the computing performance of each shard.
Targeted read	The dds 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 dds 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.

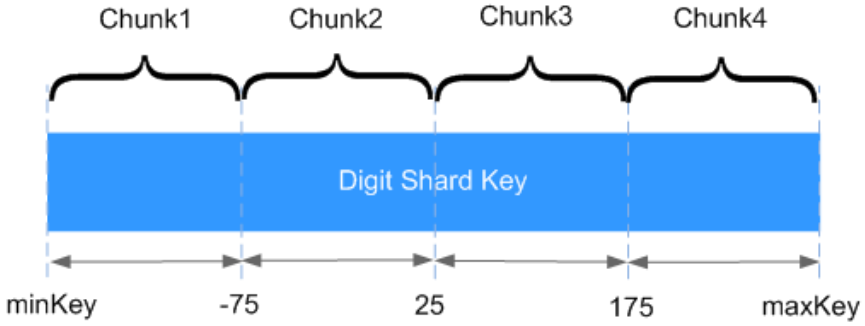
There are two data distribution policies: ranged sharding and hashed sharding. For details, see **Step 4**.

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 4-1 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 dds mongos based on the workload of each shard.

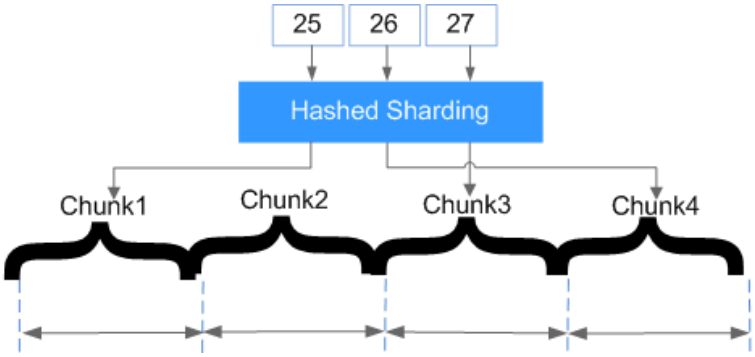
Ranged sharding can easily meet the requirements of query in a certain range. For example, if you need to query documents whose shard key is in range [-60,20], dds mongos only needs to forward the request to chunk 2.

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 4-2 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.

5 How Do I Improve DDS Performance by Optimizing SQL Statements?

DDS is inherently a NoSQL database with high performance and strong extensibility. Similar to relational databases, such as RDS for MySQL, RDS for SQL Server, and Oracle, DDS instance 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

1. Use short field names to save storage space. Different from an RDS database, each DDS document has its field names stored in the collection. Short name is recommended.
2. Limit the number of documents in a collection to avoid the impact on the query performance. Archive documents periodically if necessary.
3. Each document has a default **_id**. Do not change the value of this parameter.
4. Capped collections have a faster insertion speed than other collections and can automatically delete old data. You can create capped collections to improve performance based on your service requirements.

For details, see **Usage Suggestions** in the *Document Database Service Developer Guide*.

Query

Indexes

1. Create proper number of indexes for frequently queried fields based on service requirements. Indexes occupy some storage space, and the insert and indexing operations consume resources. It is recommended that the number of indexes in each collection should not exceed 5.

   If data query is slow due to lack of indexes, create proper indexes for frequently queried fields.
2. For a query that contains multiple shard keys, 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.

3. TTL indexes can be used to automatically filter out and delete expired documents. The index for creating TTL must be of type date. TTL indexes are single-field indexes.
4. You can create field indexes in a collection. However, if a large number of documents in the collection do not contain key values, you are advised to create sparse indexes.
5. 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 multiple fields.

Command usage

1. The findOne method returns the first document that satisfies the specified query criteria from the collection according to the natural order. To return multiple documents, use this method.
2. 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 **$project** to limit the returned field, reducing the network traffic and the memory usage of the client.
3. In addition to prefix queries, regular expression queries take longer to execute than using selectors, and indexes are not recommended.
4. Some operators that contain **$** 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 queries depend 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**.

Precautions

1. Indexes cannot be used in operators $where and $exists.
2. If the query results need to be sorted, control the number of result sets.
3. If multiple field indexes are involved, place the field used for exact match before the index.
4. 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

     Modifying 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

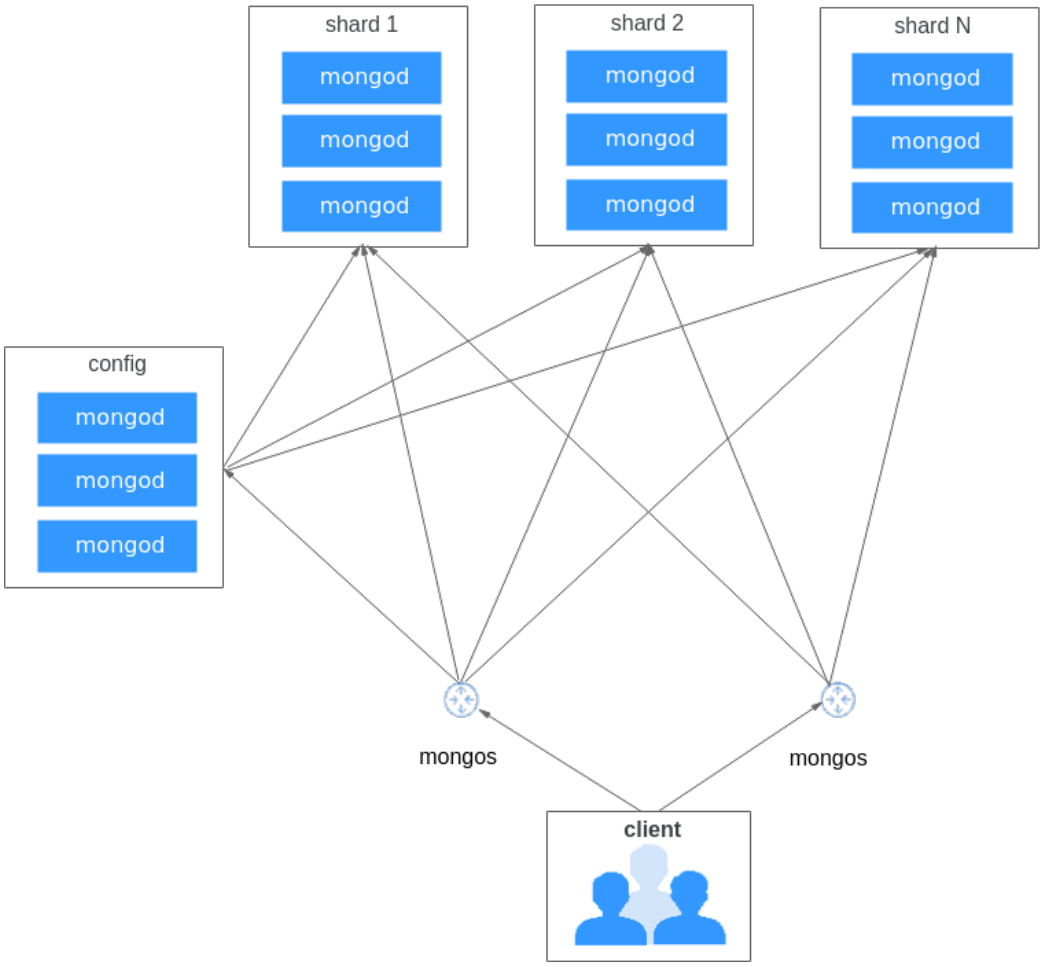
  During aggregation, $match must be placed before $group to reduce the number of documents to be processed by the $group operator.

6 How Do I Prevent the dds mongos Cache Problem?

Background

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

Figure 6-1 DDS cluster architecture



A cluster instance consists of the following three parts:

- dds 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 dds mongos nodes. 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 dds 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 dds mongos, dds 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, dds 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 dds 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 dds mongos caches the route information.

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

- If the dds mongos is started, it will obtain the latest route information from the config server and caches them locally.
- If the dds 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 dds mongos.

NOTE

Only the metadata related to the requested data is updated.

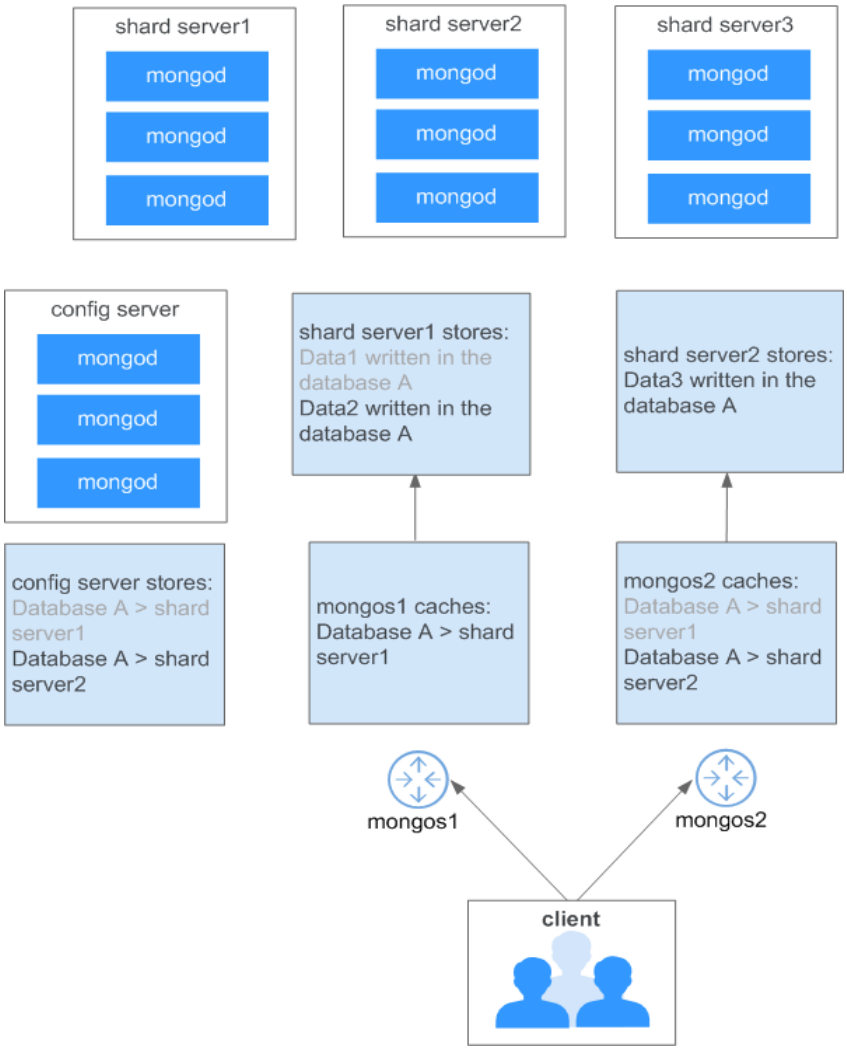
The data to be updated is in the unit of DB.

Scenarios

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

1. Create 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 database A.
2. If database A is deleted through mongos2, the information about database A in the config server and shard server1 is deleted. As a result, mongos1 cannot identify data1 because database A has been deleted.
3. When data2 is written to database A through 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 database A through mongos2, new information about database A will be generated again on the config server and shard server2 because mongos2 has identified that database A has been deleted.
4. In this case, the route information cached in the mongos1 and mongos2 is inconsistent. mongos1 and mongos2 are associated with different shard servers, and data is not shared between them. As a result, data inconsistency occurs.

Figure 6-2 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 databases or collections, run **db.adminCommand("flushRouterConfig")** 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 sharding disabled is deleted, do not create a database or collection with the same name as the deleted database or collection.

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

7 How Do I Solve the High CPU Usage Issue?

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

The following describes how to analyze current slow queries. After the analysis and optimization, queries will be processed better and indexes will be used more efficiently.

Analyzing Current Queries

1. Connect to an instance using Mongo Shell.

   To access an instance from the Internet

   For details, see

   - **Connecting to a Cluster Instance over a Public Network**
   - **Connecting to a Replica Set Instance over a Public Network**
   - **Connecting to a Single Node Instance over a Public Network**

   To access an instance that is not publicly accessible

   For details, see

   - **Connecting to a Cluster Instance over a Private Network**
   - **Connecting to a Replica Set Instance over a Private Network**
   - **Connecting to a Single Node Instance over a Private Network**

2. Run the following command to view the operations being performed 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 microseconds. If the returned value of this field is too large, check whether there is something wrong with the request.
- **op**: operation type. The operations can be query, insert, update, delete, or command.
- **ns**: 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 while services are being processed but then becomes high during just certain operations, analyze the requests that take a long time to execute.

   If an abnormal query is found, find the **opid** corresponding to the operation and run **db.killOp(*opid*)** to kill it.

Analyzing Slow Queries

Slow query profiling is enabled for DDS by default. The system automatically records any queries whose execution takes longer than 100 ms to the **system.profile** collection in the corresponding database. You can:

1. Connect to an instance using Mongo Shell.

   To access an instance from the Internet

   For details, see

   - **Connecting to a Cluster Instance over a Public Network**
   - **Connecting to a Replica Set Instance over a Public Network**
   - **Connecting to a Single Node Instance over a Public Network**

   To access an instance that is not publicly accessible

   For details, see

   - **Connecting to a Cluster Instance over a Private Network**
   - **Connecting to a Replica Set Instance over a Private Network**
   - **Connecting to a Single Node Instance over a Private Network**

2. Select a specific database (using the **test** database as an example):

   use test

3. Check whether slow SQL queries have been collected in **system.profile**.

   show collections;

   - If the command output includes **system.profile**, slow SQL queries have been generated. Go to the next step.
     ```
     mongos> show collectionssystem.profiletest
     ```
   - If the command output does not contain **system.profile**, no slow SQL queries have been generated, and slow query analysis is not required.
     ```
     mongos> show collectionstest
     ```

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 a request that scanned the entire table, including 1,561,632 documents and without using a search index.

   ```
   { "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"}
   ```

   The following stages can be causes for a slow query:

   - **COLLSCAN** involves a full collection (full table) scan.

     When a request (such as query, update, and delete) requires a full table scan, a large amount of CPU resources are occupied. If you find **COLLSCAN** in the slow query log, CPU resources may be occupied.

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

   - **docsExamined** involves a full collection (full table) scan.

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

- **IXSCAN** and **keysExamined** scan indexes.

  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 **keysExamined** to see how many indexes are scanned in a query. A larger value indicates a higher CPU usage.

  If the index is not appropriate or there are many matching results, the CPU usage may spike and the execution can slow down.

  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 more 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 effective because the b field has a few same values.
  ```

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

- **SORT** and **hasSortStage** may involve sorting a large amount of data.

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

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

Other operations, such as index creation and aggregation (combinations of traversal, query, update, and sorting), also apply to the preceding scenarios because they are also CPU intensive operations. For more information about profiling, see **official documents**.

Analysis Capability

After the analysis and optimization of the requests that are being executed and slow requests, all requests use proper indexes, and the CPU usage becomes stable. If the CPU usage remains high after the analysis and troubleshooting, the current instance may have reached the performance bottleneck and cannot meet service requirements. In this case, you can perform the following operations to solve the problem:

1. View monitoring information to analyze instance resource usage. For details, see **Viewing Monitoring Metrics**.
2. Change the DDS instance class or add shard nodes.

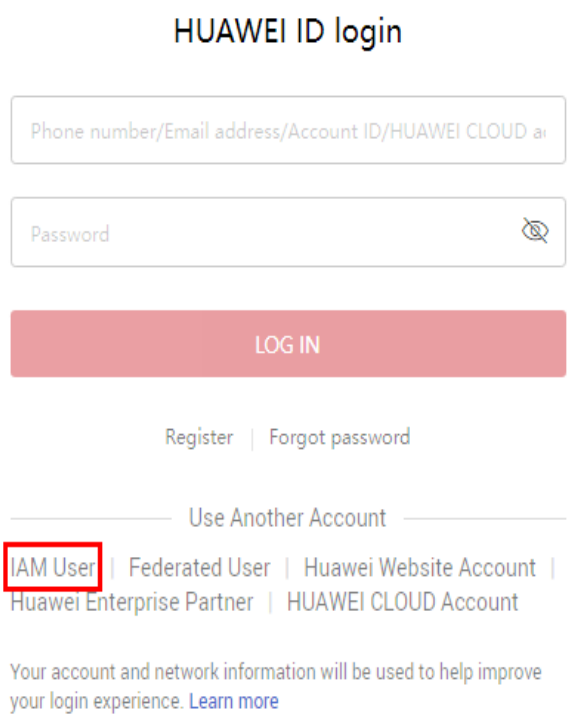
8 Creating a User and Granting the Read-Only Permission to the User

Step 1: Create a User Group and Grant Permissions

Users in the same user group have the same permissions. Users created in IAM inherit permissions from the groups to which they belong. Users created in IAM inherit permissions from the groups they belong to. To create a user group, perform the following steps:

Step 1 Log in to using your account.

Figure 8-1 HUAWEI ID Login



Step 2 On the management console, click the username in the upper right corner and then choose **Identity and Access Management**.

Step 3 On the IAM console, choose **User Groups** in the navigation pane. Then click **Create User Group**.

Step 4 Enter a user group name (for example, **test_01**), set the password, and click **OK**.

The user group is then displayed in the user group list.

Step 5 In the user group list, choose **Authorize** in the row that contains the **test_01** user group.

Step 6 Select **Document Database Service** from the drop-down list, select **DDS ReadOnlyAccess**, and click **Next**.

Step 7 Specify the scope and click **OK**.

- All resources
- Region-specific projects: The selected permissions will be applied to resources in the region-specific projects you select.

----End

Step 2: Create an IAM User

IAM users can be created for employees or applications of an enterprise. Each IAM user has their own security credentials, and inherits permissions from the groups it is a member of. To create an IAM user, perform the following steps:

Step 1 On the IAM console, choose **Users** in the navigation pane. Then click **Create User**.

Step 2 Specify the user information on the **Create User** page. To create more users, click **Add User**. You can add a maximum of 10 users at a time.

- **Username**: Used for logging in to . For this example, enter **James**.
- **Email Address**: Email address bound to the IAM user. This parameter is mandatory if the access type is specified as **Set by user**.
- (Optional) **Mobile Number**: Mobile number bound to the IAM user.
- (Optional) **Description**: Description of the user.

Step 3 Configure required parameters and click **Next**.

Table 8-1 Configuration items

Parameter	Description
Access Type	• **Programmatic access**: Select this option to allow the user to access cloud services using development tools, such as APIs, CLI, and SDKs. You can generate an access key or set a password for the user. • **Management console access**: Select this option to allow the user to access cloud services using the management console. You can set or generate a password for the user or request the user to set a password at first login.

Parameter	Description
Credential Type	• Access key: Download the access key after the user is created. • Password: If you create multiple users, set a password for the users and determine whether to require the users to reset the password at first login. If you create one user, you can select **Automatically generated** and the system automatically generates a login password for the user.
Login Protection	To ensure account security, you are advised to select **Enable**.

Step 4 Add the users to user group created in **Step 4** and click **Create User**.

Step 5 Check the created users in the user list. If you select **Access key** for **Credential Type**, you can download the access key after you create the user. You can also manage the access keys on the **My Credentials** page.

----End

Step 3: Log In and Verify Permissions

After the user is created, use the username and identity credential to log in to , and verify that the user has the permissions defined by the **DDS ReadOnlyAccess** policy.

Step 1 On the login page, click **IAM User** in the lower left corner.

Step 2 Enter the account name, username, and password, and click **Log In**.

- The account name is the name of the account that created the IAM user.
- The username and password are those set by the account when creating the IAM user.

If the login fails, contact the entity owning the account to verify the username and password. Alternatively, you can reset the password by following the procedure in "Resetting Password for an IAM User".

Step 3 After successful login, switch to a region where the user has been granted permissions on the management console.

Step 4 Choose **Service List** > **Document Database Service**. Then click **Buy DB Instance** on the DDS console. If a message appears indicating insufficient permissions to perform the operation, the **DDS ReadOnlyAccess** policy has already taken effect.

Step 5 Choose any other service in the **Service List**. If a message appears indicating insufficient permissions to access the service, the **DDS ReadOnlyAccess** policy has already taken effect.

----End

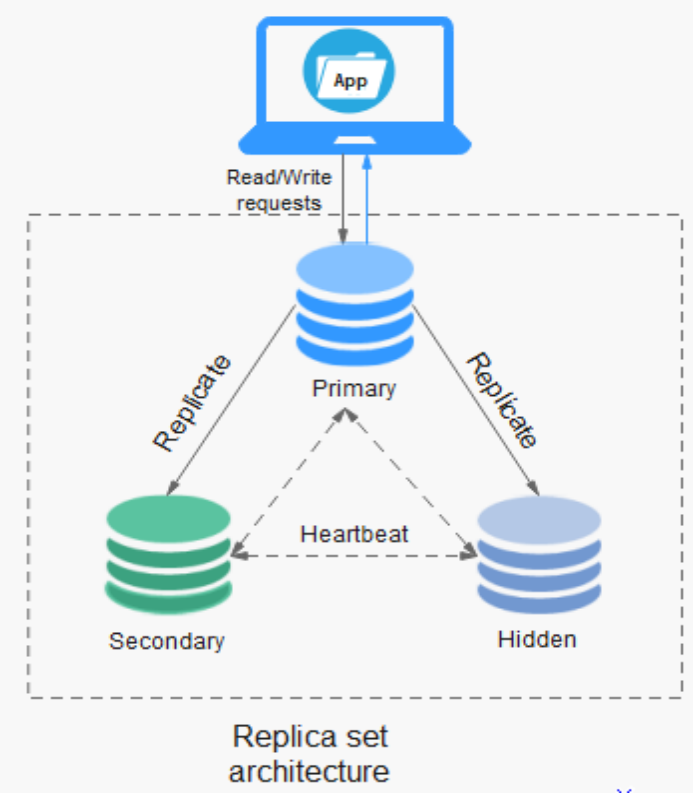
9 How Is a DDS Node Going to Be Disconnected and What Can I Do?

A replica set consists of three nodes: primary, secondary, and hidden. The three-node architecture is set up automatically, and the three nodes automatically synchronize data with each other to ensure data reliability. Replica sets are recommended for small- and medium-sized service systems that require high availability.

- Primary node: Primary nodes are used to process both read and write requests.
- Secondary node: Secondary nodes are used to process read requests only.
- Hidden node: Hidden nodes are used to back up service data.

You can perform operations on the primary and secondary nodes. If the primary node is faulty, the system automatically selects a new primary node. The following figure shows the replica set architecture.

Figure 9-1 Three-node replica set architecture



DDS can write data only on the primary node. When data is written to the primary node, oplogs are generated. The secondary and hidden nodes read oplogs from the primary node for replay to ensure data consistency.

The storage capacity of oplogs is determined by the value of **oplogSize** (10% of the default disk capacity).

- How is the primary/secondary latency generated?

  If the write speed of the primary node is too fast and exceeds the replay speed of the oplog read on the secondary node, the primary/secondary latency occurs.
- When is a node going to be disconnected?

  The storage capacity of oplogs is limited. If the capacity reaches the upper limit, the earliest oplog will be deleted. The secondary node reads oplogs and records the last oplog each time. If the primary/secondary latency reaches a certain value, the secondary node finds that the last oplog point has been deleted. In this case, the secondary node cannot continue to read oplogs, and the secondary node is disconnected.
- How to effectively prevent the secondary node from being disconnected?
  - Set **writeConcern** to **majority**. In this way, data is written to a majority of nodes, ensuring data consistency.
  - Increase the oplog storage capacity by changing the value of **oplogSizePercent** on the console. For details, see .
  - Perform time-consuming DDL operations (such as index creation) and data backup operations during off-peak hours to avoid burst addition, deletion, and modification operations.

NOTE

If **writeConcern** is not set to **majority**, the data that is not synchronized to the secondary node may be lost when a primary/secondary switchover occurs.

10 Avoiding Cursor Invalidity Caused by hideIndex

Scenarios

In DDS, some operations may cause cursors to be invalid. The behaviors of DDS are the same as those of MongoDB Community Edition. The following are some common examples:

Invalid cursors caused by hideIndex

Cause:

The **hideIndex** operation changes the metadata of an index, invalidating the existing cursor.

Sample code:

```
(function() {
   "use strict";

   let collName = "test_coll";
   let coll = db.getCollection(collName);
   coll.drop();

      // Create an index and insert data.
   coll.createIndex({x: 1}, {background: false});
   for (var i = 0; i < 1000; ++i) {
      coll.insert({x: i});
   }

      // Obtain a cursor.
   let cursor = coll.find({x: {$gte: 1}});

      // Use the cursor, but the cursor is not exhausted.
   for (var i = 0; i < 101; ++i) {
      let doc = cursor.next();
      print(tojson(doc));
   }

      // Hide an index before the cursor is exhausted.
   coll.hideIndex("x_1");
```



```
    // Traverse the cursor.
  cursor.forEach((doc) => {
            // An error is reported for the cursor.
    // "errmsg" : "definition of index 'x_1' changed"
    // "codeName" : "QueryPlanKilled"
    print(tojson(doc));
  });
})();
```

Result:

The cursor is invalid. Errors may be reported.

```
2024-12-24T16:26:42.445+0800 E QUERY    [js] Error: getMore command failed: {
  "ok" : 0,
  "errmsg" : "definition of index 'x_1' changed",
  "code" : 175,
  "codeName" : "QueryPlanKilled"
} :
```

Invalid cursors caused by dropIndex

Cause:

After an index is deleted, DDS needs to recalculate query plans, invalidating the existing cursor.

Sample code:

```
(function() {
  "use strict";

  let collName = "test_coll";
  let coll = db.getCollection(collName);
  coll.drop();

    // Create an index and insert data.
  coll.createIndex({x: 1}, {background: false});
  for (var i = 0; i < 1000; ++i) {
    coll.insert({x: i});
  }

    // Obtain a cursor.
  let cursor = coll.find({x: {$gte: 1}});

    // Use the cursor, but the cursor is not exhausted.
  for (var i = 0; i < 101; ++i) {
    let doc = cursor.next();
    print(tojson(doc));
  }

    // Delete an index before the cursor is exhausted.
  coll.dropIndex("x_1");

    // Traverse the cursor.
  cursor.forEach((doc) => {
            // An error is reported for the cursor.
    // "errmsg" : "index 'x_1' dropped",
    // "codeName" : "QueryPlanKilled"
    print(tojson(doc));
  });
})();
```


Result:

The cursor is invalid because query plans need to be recalculated after an index is deleted.

```
2024-12-25T10:34:29.948+0800 E QUERY    [js] Error: getMore command failed: {
    "ok" : 0,
    "errmsg" : "index 'x_1' dropped",
    "code" : 175,
    "codeName" : "QueryPlanKilled"
}
```

Invalid cursors caused by renameCollection

Cause:

The renameCollection operation also causes cursors to become invalid.

Sample code:

```
(function() {
    "use strict";

    let collName = "test_coll";
    let coll = db.getCollection(collName);
    coll.drop();

        // Create an index and insert data.
    coll.createIndex({x: 1}, {background: false});
    for (var i = 0; i < 1000; ++i) {
        coll.insert({x: i});
    }

        // Obtain a cursor.
    let cursor = coll.find({x: {$gte: 1}});

        // Use the cursor, but the cursor is not exhausted.
    for (var i = 0; i < 101; ++i) {
        let doc = cursor.next();
        print(tojson(doc));
    }

        // Rename a collection before the cursor is exhausted.
    coll.renameCollection("test_coll_reaname");

        // Traverse the cursor.
    cursor.forEach((doc) => {
                // An error is reported for the cursor.
        // "errmsg" : "collection dropped between getMore calls",
        // "codeName" : "OperationFailed"
        print(tojson(doc));
    });
})();
```

Result:

The cursor is invalid because the name of the collection is changed and the cursor loses the context.

```
2024-12-25T10:39:28.624+0800 E QUERY    [js] Error: getMore command failed: {
    "ok" : 0,
    "errmsg" : "collection dropped between getMore calls",
```



```
    "code" : 96,
    "codeName" : "OperationFailed"
} :
```

Invalid cursors caused by dropCollection

Cause:

Deleting a collection causes cursors to lose the context. As a result, cursors become invalid.

Sample code:

```
(function() {
    "use strict";

    let collName = "test_coll";
    let coll = db.getCollection(collName);
    coll.drop();

        // Create an index and insert data.
    coll.createIndex({x: 1}, {background: false});
    for (var i = 0; i < 1000; ++i) {
        coll.insert({x: i});
    }

        // Obtain a cursor.
    let cursor = coll.find({x: {$gte: 1}});

        // Use the cursor, but the cursor is not exhausted.
    for (var i = 0; i < 101; ++i) {
        let doc = cursor.next();
        print(tojson(doc));
    }

        // Delete a collection before the cursor is exhausted.
    coll.drop();

        // Traverse the cursor.
    cursor.forEach((doc) => {
                // An error is reported for the cursor.
        // "errmsg" : "collection dropped between getMore calls",
        // "codeName" : "OperationFailed"
        print(tojson(doc));
    });
})();
```

Result:

The cursor is invalid because the collection is deleted and the cursor loses the context.

```
2024-12-25T10:40:33.737+0800 E QUERY    [js] Error: getMore command failed: {
    "ok" : 0,
    "errmsg" : "collection dropped between getMore calls",
    "code" : 96,
    "codeName" : "OperationFailed"
} :
```


Solution

- Process cursors before an index operation: Ensure that a cursor is exhausted before the **hideIndex** or **dropIndex** operation.
- Re-obtain cursors: If an index changes, perform the query operation again to obtain the new cursor.
- Plan in advance: Do not perform index operations during long-time queries.

Sample code:

```
(function() {
    "use strict";

    let collName = "test_coll";
    let coll = db.getCollection(collName);
    coll.drop();

        // Create an index and insert data.
    coll.createIndex({x: 1}, {background: false});
    for (var i = 0; i < 1000; ++i) {
        coll.insert({x: i});
    }

        // Obtain a cursor.
    let cursor = coll.find({x: {$gte: 1}});

        // Use the cursor, but the cursor is not exhausted.
    for (var i = 0; i < 101; ++i) {
        let doc = cursor.next();
        print(tojson(doc));
    }

        // Hide an index before the cursor is exhausted.
    coll.hideIndex("x_1");

        // Traverse the cursor.
    // cursor.forEach((doc) => {
                // An error is reported for the cursor.
        // "errmsg" : "definition of index 'x_1' changed"
        // "codeName" : "QueryPlanKilled"
        // print(tojson(doc));
    // });

        // Obtain the cursor again.
    cursor = coll.find({x: {$gte: 1}});
    cursor.forEach((doc) => {
        print(tojson(doc));
    });
})();
```


11 Using DDS to Store and Analyze Log Data

Log data plays an important role in the O&M and analysis of modern applications. It not only helps monitor the health status of applications, but also provides valuable data for service insight, user experience optimization and business decision-making. However, as the data volume increases sharply, traditional log storage and analysis become insufficient. This document describes how to use Document Database Service (DDS) to efficiently store and analyze log data. DDS 4.2 and later versions use RocksDB as the underlying storage engine. RocksDB has good performance in scenarios where write operations are more than read operations, such as log storage and analysis.

By properly designing a log storage structure, optimizing write and query policies, and using data sharding, DDS can efficiently store and analyze massive amounts of log data, providing strong support for application O&M and service analysis. In addition to powerful data storage, DDS provides flexible configuration and optimization policies to help users easily cope with challenges and mine infinite data value in the era of data explosion.

Log Data Storage

In the Internet of Things (IoT) field, device log data plays an important role. It not only helps monitor the status of devices, but also provides detailed information about device usage modes and fault prediction. For example, a log record of a smart home security system may include information such as a device ID, a timestamp, an event type (such as lock opening status or motion detection), a device status, and a possible error code.

- Log data example

  The following is a typical IoT device log record:

  ```
  DeviceID: 001, Timestamp: 2023-04-05T14:30:00Z, Event: DoorLockOpened, DeviceStatus: Active, Error: None
  ```

- Storage mode optimization

  When data is stored in DDS, logs are converted into structured documents to extract key fields. For example:

  ```
  { _id: ObjectId('5f442120eb03305789000000'),
    device_id: "001",
    timestamp: ISODate("2023-04-05T14:30:00Z"),
  ```



```
    event: "DoorLockOpened",
    device_status: "Active",
    error: "None"
}
```

Irrelevant fields are filtered out based on analysis requirements to save storage space. For example, if you want to analyze the error status of a device that is not concerned, you can omit the **error** field.

Log Writing and Performance Optimization

DDS replica set instances provide data redundancy and highly reliable storage. To support high-concurrency log writing, you can customize the **writeConcern** parameter. For example, using {w: 0} can achieve the highest write throughput, but at the cost of data durability. For key device logs, a more secure level, such as {w: 1} or {w: "majority"}, is recommended. This setting allows writing multiple logs in batches and can significantly improve efficiency.

```
replica:PRIMARY> log = {
...     "device_id": "001",
...     "timestamp": ISODate("2023-04-05T14:30:00Z"),
...     "event": "DoorLockOpened",
...     "device_status": "Active",
...     "error": "None" ... };
// Write using {w: 0}.
replica:PRIMARY> db.getSiblingDB("iot_logs").events.insert(log, {w: 0})
WriteResult({ "nInserted" : 1 })
// Write using {w: 1}.
replica:PRIMARY> db.getSiblingDB("iot_logs").events.insert(log, {w: 1})
WriteResult({ "nInserted" : 1 })
// Write using {w: "majority"}.
replica:PRIMARY> db.getSiblingDB("iot_logs").events.insert(log, {w: "majority"})
WriteResult({ "nInserted" : 1 })
// Batch write at a time:
replica:PRIMARY> logs = [
... { "device_id": "002", "timestamp": ISODate("2023-04-06T09:45:00Z"), "event": "MotionDetected",
"device_status": "Active", "error": "None"},
... { "device_id": "003", "timestamp": ISODate("2023-04-07T16:15:00Z"), "event": "TemperatureAlarm",
"device_status": "Active", "error": "None"},
... {"device_id": "004", "timestamp": ISODate("2023-04-08T20:30:00Z"), "event": "WindowOpened",
"device_status": "Active", "error": "None"}
... ]
replica:PRIMARY> db.getSiblingDB("iot_logs").events.insertMany(logs, {w: 0})
```

Log Query and Analysis

- Log query example

  Query all events of a specified device in a specified time range.

  ```
  db.getSiblingDB("iot_logs").events.find({"device_id": "001", "timestamp": {"$gte":
  ISODate("2023-04-05T00:00:00Z"), "$lt": ISODate("2023-04-06T00:00:00Z")}})
  ```

- Create indexes to improve query efficiency.

  Creating indexes for the **device_id** and **timestamp** fields can accelerate the query in the preceding scenario.

  ```
  replica:PRIMARY> db.getSiblingDB("iot_logs").events.createIndex({"device_id": 1, "timestamp": 1})
  ```

- Complex analysis.

  The DDS aggregation framework can be used for more complex query and analysis. For details, see **Aggregation Operations**.

Data Sharding and Scaling

As logs spike, the log data volume increases exponentially. A single database node cannot meet the storage and query requirements of massive data. DDS uses the

sharding technology to provide the horizontal scalability, effectively share data storage and query stress, and ensure high availability and high performance of the system.

- Data sharding

  Data sharding means dividing a dataset into multiple parts and store them on different database nodes (shards). Each shard stores a part of a dataset, and data is distributed based on the shard key. A shard key policy determines the data distribution mode and affects the write and query performance.

- Shard key policies

  When selecting a shard key, consider the following points:

  - **Even distribution**: Shard keys must ensure that data is evenly distributed among shards to avoid hotspots.
  - **Query mode**: Shard keys must match common query modes to improve query efficiency.
  - **Data access mode**: If the data access mode is time-based, you can use timestamps or time-based fields as shard keys.

- Device ID-based sharding

  Assume that there are a large number of IoT devices and each device generates a large number of logs. You can use **device_id** as the shard key to ensure that the log data of each device is stored on different shards, achieving even data distribution. Example code:

  ```
  db.getSiblingDB("iot_logs").events.createIndex({"device_id": 1})
  sh.enableSharding("iot_logs")
  sh.shardCollection("iot_logs.events", {"device_id": 1 })
  ```

  NOTE

  - When creating a sharded collection, ensure that the shard key is unique in the data to achieve even data distribution.
  - When selecting a shard key, consider the data access mode to optimize the query performance.
  - Sharded cluster management, such as adding or deleting shards, should be adjusted based on data growth and query requirements.

Automatically Deleting Expired Data

- **TTL indexes**: Expired documents are deleted automatically, for example, you can set the period to 30 days.

  ```
  db.eventlog.createIndex( { "timestamp": 1 }, { expireAfterSeconds: 30 * 24 * 60 * 60 } )
  ```

- **Capped collections**: The collection size is limited, and the earliest document is automatically deleted. In capped collections, documents are inserted and retrieved according to the insertion order. Capped collections work similarly to circular buffers: Once a collection is full, it makes space for new documents by overwriting the earliest document in the collection.

  ```
  // Create a capped collection when creating a collection.
  db.getSiblingDB("iot_logs").dropDatabase()
  db.getSiblingDB("iot_logs").createCollection( "events", { capped: true, size: 5242880 } )
  ```

- **Periodic archiving**: Log data is archived by month to facilitate historical data management and query.

  ```
  db.getSiblingDB("iot_logs").events.renameCollection("events202301")
  ```


NOTICE

In a DB instance, the total number of databases cannot exceed 200, and the total number of collections cannot exceed 500. If there are too many collections, the memory may be overloaded. In addition, the performance for restarting a DB instance and performing a primary/standby switchover may deteriorate due to too many collections, which affects the high availability performance in emergencies. You are advised to periodically delete unnecessary collections.

12 DDS Query Plans and Query Replanning

A DDS query plan is a detailed process that determines how a query is executed. The DDS query planner chooses an execution plan based on the query criteria, data distribution, and index information. However, in some cases, DDS re-evaluates and generates a new query plan, that is, query replanning. Learning query replanning helps improve database performance and effectively reduce resource waste.

The following describes how DDS query plans work, query replanning scenarios, and best practices for handling and optimizing query replanning.

Query Plan Overview

A query plan is an execution path selected by the DDS query planner when a query is executed. This execution path defines how to scan data, whether to use indexes, and how to process each query phase. DDS generates a query plan in the following steps:

1. Parsing a query: DDS parses query criteria and identifies fields and operators.
2. Selecting indexes: Based on the query criteria, DDS checks available indexes and chooses the most appropriate index to accelerate the query.
3. Evaluating execution paths: DDS evaluates possible execution paths based on the data volume, field selectivity, and index type, and chooses the optimal execution path to generate a new query plan.
4. Executing the query: DDS executes the query based on the selected query plan and returns the result.

Query Replanning Overview

Query replanning refers to the process in which DDS re-evaluates and generates a new query plan based on data or query structure changes during query execution. When query replanning occurs, DDS discards the previously selected query plan and uses the new execution path.

The main purpose of query replanning is to ensure that the query can adapt to new data distribution, index changes, and load changes when the data environment changes, thereby maintaining stable query performance.

Query Replanning Scenarios

The following lists some common query replanning scenarios:

- Index changes

  Adding or deleting indexes: When indexes of a field change (for example, an index is added or deleted), DDS may replan the query execution plan. The query may re-evaluate the validity of existing indexes and then chooses a new index, or decide to use a full table scan.

- Data distribution changes

  If data distribution changes (for example, the selectivity of a field changes significantly), DDS may re-evaluate the query plan to ensure that the optimal execution path can be chosen. For example, a field becomes sparser or denser, resulting in a change in index efficiency.

- Query criteria changes

  When query criteria change, DDS may need to re-generate a query plan. For example, new filter criteria are added to some queries, or different operators are used in the query criteria (for example, from **$eq** to **$in**).

- Aggregation pipeline changes

  During an aggregation operation, if the data volume, field, or pipeline sequence changes, DDS may re-evaluate the execution plan of the aggregation pipeline. Especially when the data filter criteria in the pipeline phase change, DDS may reselect an execution path.

- Environment resource load changes

  If an instance load changes (for example, node resources are used up, the storage space is insufficient, or the network latency increases), DDS may adjust the query plan to optimize the query performance under the current resource conditions.

- Invalid query cache

  DDS caches the optimal query plan that is executed previously. In some cases, DDS may use the query cache. If the query cache becomes invalid, a new query plan may be regenerated.

Impacts of Query Replanning

When query replanning occurs, the keyword **replanned:1** is displayed in the slow query log, indicating that the database cannot provide an always valid plan for the query conditions of the current query.

Frequent query replanning has the following impacts:

- **Increased CPU overhead**: Frequent replanning consumes more CPU resources and affects the overall performance.
- **Increased memory usage**: Frequent replanning increases the memory usage and may affect other operations.
- **Increased query delay**: Each replanning increases the query execution time. As a result, the response becomes slow.

Handling Suggestions for Query Replanning

- Periodically monitor query plans.

  You are advised to periodically monitor query execution plans, especially after a system changes, for example, adding indexes, adding data, or modifying the query structure. Using either of the following methods:

  - Use the **explain()** method to analyze the execution plan of a query.
  - Periodically check slow query logs to learn about possible query plan changes.

- Check the cause of a query plan change.

  When query replanning occurs, you need to determine which factors cause the change of a query plan. Check the following causes:

  - Index changes: Run **db.collection.getIndexes()** to check whether indexes are added or deleted.
  - Data distribution: Check whether data changes significantly based on statistics. For example, you can run **db.collection.stats()** to view collection statistics, such as the number of documents, data size, and index size.
  - Query criteria: Check whether the query criteria are changed, whether a new field is added, or whether the query operators are changed.

- Optimize query replanning.

  The following are some suggestions for optimizing query replanning:

  - [Recommended] Ensure that the database environment has sufficient resources (such as CPUs, memory, and storage) to handle the load, reduce query replanning caused by resource limitations, and upgrade the instance specifications to relieve the database load.
  - [Recommended] Predict data distribution. During index design, you can use data distribution prediction to choose proper fields for indexing to prevent frequent query replanning caused by data distribution changes.
  - [Recommended] Use stable indexes. Ensure that stable and proper indexes are provided for common queries. Especially when there is a large volume of data, indexes can effectively reduce the occurrence of query replanning.
  - [Recommended] Avoid frequently changing query structures. Frequent changes of query criteria may cause DDS to frequently re-generate query plans. You are advised to ensure the consistency of query criteria based on service requirements.
  - [Recommended] Optimize aggregation pipelines. If your query involves a complex aggregation pipeline, you are advised to periodically analyze the execution plan of the pipeline and optimize it. You can use **$match** to filter data in the early stage of the aggregation pipeline to reduce the amount of data processed in subsequent stages.
  - Use **hint()** to specify an index for query criteria that are replanned. Calling this method for a query can overwrite the default index selection and query optimization process of DDS. Example:

    ```
    db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1})
    ```

  - Use index filtering to restrict indexes that you want to use for query criteria that are replanned. Index filters override the expected behavior of the query planner in selecting query plans. If you specify a hint and an

index filter for a query, the index filter overwrites the specified hint. Therefore, use the index filter with caution. The following example creates an index filter for the **orders** collection. This filter is suitable for queries whose predicate is an equivalent match of the **item** field, where only the **quantity** field is projected, and sorting is specified by **order_date** in ascending order.

```
db.runCommand(
  {
    planCacheSetFilter: "orders",
    query: { item: "ABC" },
    projection: { quantity: 1, _id: 0 },
    sort: { order_date: 1 },
    indexes: [
      { item: 1, order_date: 1 , quantity: 1 }
    ]
  }
)
```

For plan cache query structures, the query planner will only consider index plans that use index { item: 1, order_date: 1, quantity: 1 }.

Evaluating Query Replanning and Taking Measures

Query replanning may change an execution plan, which affects the query performance. When query replanning occurs, you need to evaluate the change of the query performance, check whether the query performance is improved or decreased, and take measures accordingly.

- Performance deterioration: If the query performance deteriorates due to query replanning, you can add or adjust indexes to improve the performance.
- Performance improvement: If a query execution after replanning is more efficient, you can keep the current execution plan and monitor whether there are other potential performance bottlenecks.

13 DDS Transactions and Read/Write Concerns

Transactions

- Overview

  In DDS, an operation on a single document is atomic. Because you can use embedded documents and arrays to capture relationships between data in a single document structure instead of normalizing across multiple documents and collections, this single-document atomicity obviates the need for multi-document transactions for many practical use cases.

  For situations that require atomicity of multiple document updates or consistency between multiple document reads: Starting from version 4.0, DDS is able to execute multi-document transactions for replica sets.

  Multi-document transactions can be used across multiple operations, collections, databases, and documents, providing an "All-Or-Nothing" proposition. Transactions either apply all data changes or roll back the changes. If a transaction commits, all data changes made in the transaction are saved. If any operation in the transaction fails, the transaction aborts and all data changes made in the transaction are discarded without ever becoming visible. Until a transaction commits, write operations in the transaction are not visible outside the transaction.

  NOTICE

  In most cases, a multiple-document transaction incurs a greater performance cost over single document writes, and the availability of multiple-document transactions should not be a replacement for effective schema design. For many scenarios, the denormalized data model (embedded documents and arrays) will continue to be optimal for your data and use cases. That is, for many scenarios, modeling your data appropriately will minimize the need for multi-document transactions.

- Transaction APIs

  The following mongo shell code shows the key APIs for using DDS transactions:


```
// Runs the txnFunc and retries if TransientTransactionError encountered
function runTransactionWithRetry(txnFunc, session) {
    while (true) {
      try {
          txnFunc(session);  // performs transaction
          break;
      } catch (error) {
          // If transient error, retry the whole transaction
          if ( error.hasOwnProperty("errorLabels") &&
error.errorLabels.includes("TransientTransactionError")  ) {
              print("TransientTransactionError, retrying transaction ...");
              continue;
          } else {
              throw error;
          }
      }
    }
}

// Retries commit if UnknownTransactionCommitResult encountered
function commitWithRetry(session) {
    while (true) {
      try {
          session.commitTransaction(); // Uses write concern set at transaction start.
          print("Transaction committed.");
          break;
      } catch (error) {
          // Can retry commit
          if (error.hasOwnProperty("errorLabels") &&
error.errorLabels.includes("UnknownTransactionCommitResult") ) {
              print("UnknownTransactionCommitResult, retrying commit operation ...");
              continue;
          } else {
              print("Error during commit ...");
              throw error;
          }
      }
    }
}

// Updates two collections in a transactions
function updateEmployeeInfo(session) {
    employeesCollection = session.getDatabase("hr").employees;
    eventsCollection = session.getDatabase("reporting").events;

    session.startTransaction( { readConcern: { level: "snapshot" }, writeConcern: { w: "majority" } } );

    try{
        employeesCollection.updateOne( { employee: 3 }, { $set: { status: "Inactive" } } );
        eventsCollection.insertOne( { employee: 3, status: { new: "Inactive", old: "Active" } } );
        print(tojson(employeesCollection.find({ employee: 3 }).toArray()));
        print(tojson(eventsCollection.find({ employee: 3 }).toArray()));
        print("countDocuments = " + eventsCollection.countDocuments({}));
    } catch (error) {
        print("Caught exception during transaction, aborting.", error);
        session.abortTransaction();
        throw error;
    }

    commitWithRetry(session);
}

// insert data
employeesCollection = db.getSiblingDB("hr").employees; eventsCollection =
db.getSiblingDB("reporting").events;
employeesCollection.drop();
eventsCollection.drop();
for (var i = 0; i < 10; ++i) {
    employeesCollection.insertOne({employee: i});
```



```
    eventsCollection.insertOne({employee: i});
}

// Start a session.
session = db.getMongo().startSession( { readPreference: { mode: "primary" } } );
try{
    runTransactionWithRetry(updateEmployeeInfo, session);
} catch (error) {
   // Do something with error
} finally {
   session.endSession();
}
```

Transactions are associated with a session. You must start a session before starting a transaction. You can have at most one open transaction at a time for a session. If a session ends and it has an open transaction, the transaction aborts.

NOTICE

When using the drivers, each operation in the transaction must be associated with the session.

- Transactions and Atomicity

  Multi-document transactions are atomic.

  - If a transaction commits, all data changes made in the transaction are saved and are visible outside the transaction. Until a transaction commits, the data changes made in the transaction are not visible outside the transaction.
  - When a transaction aborts, all data changes made in the transaction are discarded without ever becoming visible. For example, if any operation in the transaction fails, the transaction aborts and all data changes made in the transaction are discarded without ever becoming visible.

- Restrictions on Transaction Operations

  For transactions:

  - You can specify Create/Retrieve/Update/Delete (CRUD) operations on existing collections. The collections used in a transaction can be in different databases.
  - You cannot read from or write to collections in the **config**, **admin**, or **local** databases.
  - You cannot write to **system.*** collections.
  - You cannot return the supported operation's query plan using **explain**.
  - For cursors created outside of a transaction, you cannot call **getMore** inside the transaction.
  - For cursors created in a transaction, you cannot call **getMore** outside the transaction.
  - You cannot execute non-CURD commands, including **listCollections**, **listIndexes**, **createUser**, **getParameter**, and **count**.
  - To perform a count operation within a transaction, use the **$count** aggregation stage or the **$group** (with a **$sum** expression) aggregation stage. Starting from DDS 4.0, mongo shell provides the

db.collection.countDocuments() method that uses the **$group** with a **$sum** expression to perform a count.

- A transaction cannot contain an insert operation that would result in the creation of a new collection, for example, an operation for implicitly creating a collection. A transaction cannot contain operations that affect the database catalog, for example, creating or dropping collections or indexes.

- Precautions for Using Transactions
  - Runtime Limit

    By default, a transaction must have a runtime of less than 1 minute. You can modify this limit using **transactionLifetimeLimitSeconds** for the DDS instances. Transactions that exceed this limit are considered expired and will be aborted by a periodic cleanup process.
  - Oplog Size Limit

    If a committed transaction contains any write operations, a single oplog entry is created. That is, each operation in the transaction does not have a corresponding oplog entry. Instead, a single oplog entry encapsulates all write operations in a transaction. Each oplog entry must be within the BSON document size limit of 16 MB.
  - Cache

    To prevent storage cache pressure from negatively impacting the performance:

    - When you abandon a transaction, abort the transaction.
    - When you encounter an error during individual operation in the transaction, abort and retry the transaction.

    The **transactionLifetimeLimitSeconds** parameter also ensures that expired transactions are aborted periodically to relieve storage cache pressure.
  - Transactions and Locks

    By default, transactions wait up to 5 milliseconds to acquire locks required by the operations in the transaction. If the transaction cannot acquire its required locks within the 5 milliseconds, the transaction aborts. Transactions release all locks upon abort or commit.

    You can use the **maxTransactionLockRequestTimeoutMillis** parameter to adjust how long transactions wait to acquire locks. Increasing **maxTransactionLockRequestTimeoutMillis** allows operations in the transactions to wait the specified time to acquire the required locks. This can help obviate transaction aborts on momentary concurrent lock acquisitions, like fast-running metadata operations. However, this could possibly delay the abort of deadlocked transaction operations.
  - Pending DDL Operations and Transactions

    If a multi-document transaction is in progress, new DDL operations that affect the same database wait behind the transaction. While these pending DDL operations exist, new transactions that access the same database as the pending DDL operations cannot obtain the required locks and will abort after waiting **maxTransactionLockRequestTimeoutMillis**. In addition, new non-transaction operations that access the same database will block until they reach their **maxTimeMS** limit.

Consider the following scenarios:

While an in-progress transaction is performing various CRUD operations on the **employees** collection in the **hr** database, you can start and complete a separate transaction to access the **foobar** collection in the **hr** database.

While an in-progress transaction is performing various CRUD operations on the **employees** collection in the **hr** database and a separate DDL operation is issued to create an index on the **employees** collection in the **hr** database, the DDL operation must wait for the transaction to complete.

When the DDL operation is pending, a new transaction attempts to access the **foobar** collection in the **hr** database. If the DDL operation remains pending for more than **maxTransactionLockRequestTimeoutMillis**, the new transaction will abort.

- **In-progress Transactions and Write Conflicts**

  If a multiple-document transaction is in progress and a write outside the transaction modifies a document that an operation in the transaction later tries to modify, the transaction aborts because of a write conflict. If a multi-document transaction is in progress and has taken a lock to modify a document, when a write outside the transaction tries to modify the same document, the write waits until the transaction ends.

- **In-progress Transactions and Stale Reads**

  Read operations inside a transaction can return old data, which is known as a stale read. Read operations inside a transaction are not guaranteed to see writes performed by other committed transactions or non-transactional writes.

  For example, consider the following sequence:

  - A transaction is in-progress.
  - A write outside the transaction deletes a document.
  - A read operation inside the transaction can read the now-deleted document since the operation uses a snapshot from before the write operation.

  To avoid stale reads inside transactions for a single document, you can use the **db.collection.findOneAndUpdate()** method. Example:

```
// insert data
employeesCollection = db.getSiblingDB("hr").employees;
employeesCollection.drop();
employeesCollection.insertOne({ _id: 1, employee: 1, status: "Active" });

// Start a session.
session = db.getMongo().startSession( { readPreference: { mode: "primary" } } );

session.startTransaction( { readConcern: { level: "snapshot" }, writeConcern: { w:
"majority" } } );

employeesCollection = session.getDatabase("hr").employees;

employeeDoc = employeesCollection.findOneAndUpdate(
  { _id: 1, employee: 1, status: "Active" },
  { $set: { employee: 1 } },
  { returnNewDocument: true }
```



```
);
print(tojson(employeeDoc));
```

NOTICE

- If the **employee** document is changed outside the transaction, the transaction aborts.
- If the **employee** document is not changed, the transaction returns and locks the document.

- Best Practices for Using Multi-Document Transactions
  - By default, DDS automatically aborts multiple-document transactions that run for more than 60 seconds. Note that if a server write volume is low, you can flexibly adjust a transaction to obtain a longer execution time. To resolve timeout issues, split a transaction into smaller parts to ensure that it can be executed within a specified period of time. Make sure that query statements are optimized and provide appropriate index coverage to quickly access data in transactions.
  - There is no limit on the number of documents that can be read in a transaction. In this best practice, no more than 1,000 documents should be modified in a transaction. If you want to modify more than 1,000 documents in a transaction, we recommend that you split the transaction into multiple parts and they can be executed in batches.
  - In DDS 4.0, a transaction is represented by a single oplog entry. The entry must fall within 16 MB in size. In DDS, oplogs record the incremental content in the case of an update operation, and record the entire document in the case of an insert operation. Therefore, all oplog entries for all statements in a transaction must fall within 16 MB in size. If this limit is exceeded, the transaction is aborted and completely rolled back. We recommend that you split a large transaction into smaller operation sets that each are 16 MB or less in size.
  - When a transaction is abnormally aborted, an exception is returned to the driver and the transaction is fully rolled back. You can add application logic to catch and retry transactions that are aborted due to temporary exceptions, such as primary/secondary switchovers or network faults. Drivers provided by DDS use retryable writes to automatically retry to commit transactions.
  - DDL operations, such as createIndex or dropDatabase, block transactions running on a namespace. All transactions that attempt to access the namespace during DDL suspension cannot obtain a lock within a specified time, which causes new transactions to be aborted.

Transactions and Read Concern

Operations in a transaction use the transaction-level read concern. This means a read concern set at the collection or database level is ignored inside the transaction. You can set the transaction-level read concern at the transaction start.

If the transaction-level read concern is unset, the transaction-level read concern defaults to the session-level read concern.

If the transaction-level and session-level read concerns are unset, the transaction-level read concern defaults to the client-level read concern. By default, the client-level read concern is **"local"** for reads on the primary node.

- Multi-document transactions support the following read concern levels:
  - **"local"**

    In most cases, the **"majority"** read concern is recommended.
  - "majority"

    If the transaction commits with write concern **"majority"**, read concern **"majority"** returns data that has been acknowledged by a majority of the replica set members and cannot be rolled back.

    If the transaction does not commit with write concern **"majority"**, read concern **"majority"** provides no guarantees that read operations read majority-committed data.
  - "snapshot"

    Read concern **"snapshot"** returns data from a snapshot of majority committed data if the transaction commits with write concern **"majority"**.

    If the transaction does not use write concern **"majority"** for the commit, the **"snapshot"** read concern provides no guarantee that read operations used a snapshot of majority-committed data.
- Suggestions on Using Transactions and Read Concern
  - In most cases, the **"majority"** read concern is recommended.

    This setting ensures data isolation and consistency. Applications can read data only when the data is replicated to most nodes in a replica set. In this way, data will not be rolled back even if a new primary node is elected.
  - In scenarios where the "read your own write" function is required, you need to read data directly from the primary node and use the **"local"** read concern.

    In this way, the latest update can be read as soon as possible after the write operation is complete. If the transaction commits with write concern **"majority"**, read concern **"majority"** can also be used.

Transactions and Write Concern

Transactions use the transaction-level write concern to commit the write operations. Any write concerns set inside a transaction are ignored. Do not explicitly set the write concern for the individual write operations inside a transaction. You can set the transaction-level write concern at the transaction start.

If the transaction-level write concern is unset, the transaction-level write concern defaults to the session-level write concern for the commit.

If the transaction-level and session-level write concerns are unset, the transaction-level write concern defaults to the client-level write concern. By default, the client-level write concern is **w: 1**.

- Multi-document transactions support all write concern **w** values, including:

  - **w: 1**

    Write concern **w: 1** returns acknowledgment after the commit is applied to the primary node. When you commit with **w: 1**, your transaction can be rolled back if there is a failover.

    When you commit with **w: 1** write concern, transaction-level **"majority"** read concern provides no guarantees that read operations in the transaction read majority-committed data.

    When you commit with **w: 1** write concern, transaction-level **"snapshot"** read concern provides no guarantee that read operations in the transaction used a snapshot of majority-committed data.

  - w: "majority"

    Write concern **w: "majority"** returns acknowledgment after the commit has been applied to a majority of (*M*) voting members. That is, the commit has been applied to the primary node and (*M*-1) voting secondary nodes.

    When you commit with **w: "majority"** write concern, transaction-level **"majority"** read concern guarantees that operations have read majority-committed data.

    When you commit with **w: "majority"** write concern, transaction-level **"snapshot"** read concern guarantees that operations have read from a synchronized snapshot of majority-committed data.

- Suggestions on Using Transactions and Write Concern
  - In most cases, the **"majority"** write concern is recommended.

    This setting ensures that write operations are acknowledged by a majority of nodes in a replica set, thereby avoiding data loss or rollback risks even in the event of node failures or abnormal switchovers.

  - In scenarios where high write performance is required, you can use write concern **w: 1** and pay close attention to the replication delay of secondary nodes.

    Write concern **w: 1** can provide better write performance and is applicable to write-intensive scenarios. In addition, you must monitor the replication delay of secondary nodes. If the delay is too long, the primary node may be rolled back unexpectedly. If the replication delay of a secondary node exceeds the oplog retention period, the secondary node may enter the **RECOVERING** state and cannot be automatically restored, reducing the instance availability. Therefore, you should pay attention to and handle this problem first.

  - Set an appropriate write concern based on different operation requirements.

    You can flexibly adjust the write concern to meet diversified service requirements. For example, financial transaction data can be written using transactions with write concern to ensure atomicity. Core player data of gaming services can be written using **w: "majority"** write concern to ensure that the data will not be rolled back. Log data can be written using the default or **w: 1** write concern to acquire better write performance.

14 DDS Metric Alarm Configuration Suggestions

Scenarios

You can set alarm rules on Cloud Eye to customize the monitored objects and notification policies and keep track of the instance status. DDS allows you to set threshold rules for instance metrics. If the value of a metric exceeds the threshold, an alarm is triggered. The system automatically sends an alarm notification to the cloud account contact through SMN, helping you learn about the running status of your DDS instance in a timely manner.

This section describes how to configure DDS metric alarm rules.

Creating an Alarm Rule

Step 1 Log in to the management console.

Step 2 Under **Management & Governance**, click **Cloud Eye**.

Step 3 In the navigation pane, choose **Alarm Management** > **Alarm Rules**.

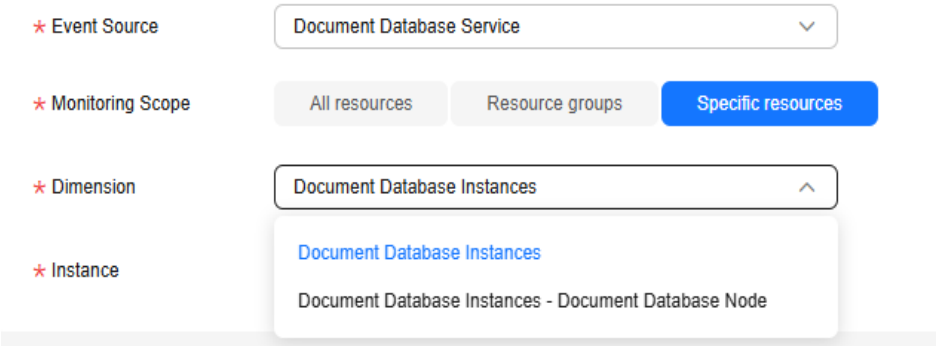
Step 4 On the displayed **Alarm Rules** page, click **Create Alarm Rule**.

Step 5 On the displayed page, set parameters as prompted.

Pay attention to the following parameters:

- **Event Source**: Select **Document Database Service**.
- **Dimension**: DDS supports instance- and node-level monitoring. Different monitoring metrics support different monitoring dimensions. For details, see **DDS Metrics**.

Figure 14-1 Configuring a monitoring dimension



- For details about other parameters, see **Creating an Alarm Rule** in the *Cloud Eye User Guide*.

----End

Table 14-1 Suggestions on configuring alarm rules of DDS metrics

Metric ID	Metrics Name	Dimension	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
mongo007_connections_usage	Percentage of Active Node Connections	Node	Raw data > 80% for three consecutive periods	Major	• Check whether the connection pool is properly used. • Check whether the timeout and other parameters are properly configured. For details, see **Common Parameter Configuration on the Driver Side**. • Increase the maximum number of connections. – For a replica set instance whose maximum number of connections is less than 16,000, you can increase the maximum number of connections by changing the instance class. – For a cluster instance, you can increase the maximum number of connections by adding mongos nodes.

Metric ID	Metrics Name	Dimension	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
mongo031_cpu_usage	CPU Usage	Node	Raw data > 80% for three consecutive periods	Major	• Identify why the CPU usage is high. For details, see **How Do I Solve the High CPU Usage Issue?** • If the CPU usage remains high, upgrade the CPU specifications. For details, see **Changing an Instance Class**. • If the CPU usage remains high and workloads cannot be completely stopped, the specification change may fail. In this case, choose Service Tickets > Create Service Ticket to submit a service ticket and ask engineers to limit the number of connections in the background. After the specification change is complete, change back the number of connections to the original value.

Metric ID	Metrics Name	Dimension	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
mongo035_disk_usage	Storage Space Usage	Node	Raw data > 80% for three consecutive periods	Major	• Identify why the storage usage is high. For details, see **High Storage Usage**. • If the storage usage remains high, scale up storage space. For details, see **Scaling Up Storage Space**.
mongo032_mem_usage	Memory Usage	Node	Raw data > 90% for three consecutive periods	Major	• Identify why the memory usage is high. For details, see **High Memory Usage**. • If the memory usage remains high for a long time, upgrade the instance class. For details, see **Changing an Instance Class**.

Metric ID	Metrics Name	Dimension	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
mongo039_avg_disk_sec_per_read	Average Time per Disk Read	Node	Raw data ≥ 0.1s for three consecutive periods	Major	• Check whether the instance has performance bottlenecks in CPU, memory, or connections. If yes, solve the bottlenecks based on the related suggestions. • If workloads cannot be optimized, upgrade the instance class or choose specifications with better disk performance. For details, see **Changing an Instance Class** and **Instance Specifications**.

Metric ID	Metrics Name	Dimension	Threshold in Best Practices	Alarm Severity in Best Practices	Handling Suggestion
mongo040_avg_disk_sec_per_write	Average Time per Disk Write	Node	Raw data ≥ 0.1s for three consecutive periods	Major	• Check whether the instance has performance bottlenecks in CPU, memory, or connections. If yes, solve the bottlenecks based on the related suggestions. • If workloads cannot be optimized, upgrade the instance class or choose specifications with better disk performance. For details, see **Changing an Instance Class** and **Instance Specifications**. • Perform a primary/secondary switchover on the console.

15 Working with Indexes

Scenarios

In database management, indexes play a crucial role in enhancing query performance. You can create both single-field indexes and composite indexes to optimize queries involving multiple fields.

Periodic analysis and optimization of index performance are essential. The database management system usually provides tools and commands to evaluate index usage and identify performance issues. It's important to note that maintaining indexes comes with a significant cost. Operations such as insertion, update, and deletion can trigger index updates, which may affect performance. It is essential to choose an index strategy that fits the specific application scenario, balancing query performance and maintenance costs.

Key to Creating Indexes

- Do not use aggregation or query statements without filter criteria. Ensure that there are indexes on the fields used in the filter criteria (for example, the fields specified in the **$match**, **find**, **update**, and **remove** statements in aggregations). This prevents full table scans and ensures more efficient query performance.
- Use high-selectivity filter criteria to narrow down the query scope and ensure that indexes exist on high-selectivity fields. It is recommended to create indexes only on fields where the number of duplicate values is less than 1% of the total number of documents in the collection. For example, if your collection contains 100,000 documents, create indexes only on fields where the number of duplicate values is no more than 1,000.
- In composite indexes, it is recommended to place high-selectivity fields that can narrow down the query scope at the beginning.
- In scenarios where a large amount of data needs to be sorted, you should create indexes on the sort field based on the exact match filter criteria. For example, if the filter condition is {a:xx, b:xx, c:xx, f:xx } and the sort condition is {e:1}, you should create composite index {a:1,b:1,c:1,f:1,e:1} to avoid in-memory sorts.
- For statements that include equality, range, and sort, the optimal composite index follows the ESR principle: equality fields are placed at the beginning, sort fields are placed in the middle, and range fields are placed at the end.

- If a query includes a collation, the collation option must be specified when creating an index for the query field.
- Before deleting an index, ensure that no index is being created on the secondary node (you can use the **currentOp** command to check the ongoing operations). If an index is being created on the secondary node, do not delete the index immediately. Otherwise, the secondary node may enter a lock wait state, leading to unexpected issues.

Impact of Index Creation

After an index creation request is submitted, the existing documents in the current data are scanned to establish the mapping between index fields and their disk locations. This process consumes I/O and compute resources. Therefore, consider the following:

- If an index to be created involves a large amount of data, create it during off-peak hours to minimize impact on production services.
- If multiple indexes need to be created for a given collection, use the **createIndexes** method instead of **createIndex**. This allows multiple indexes to be created with a single scan, reducing the compute resources and I/O overhead associated with index creation.
- Plan the indexes required for core service tables in advance.

Impact of Indexes on Writing Data

While indexes can enhance query performance by reducing the need to scan every document in a collection, they come with certain trade-offs. Each index on a collection requires the database to update both the collection and the relevant index fields whenever a document is inserted, updated, or deleted. For example, if a collection has nine indexes, the database must execute ten write operations before confirming the operation to the client. As a result, each additional index increases write latency, I/O operations, and overall storage usage. For optimal performance, review and minimize the number of indexes in your collections, and add only those that are necessary to enhance performance for common queries. You are advised to keep the number of indexes per collection to 10 or fewer.

