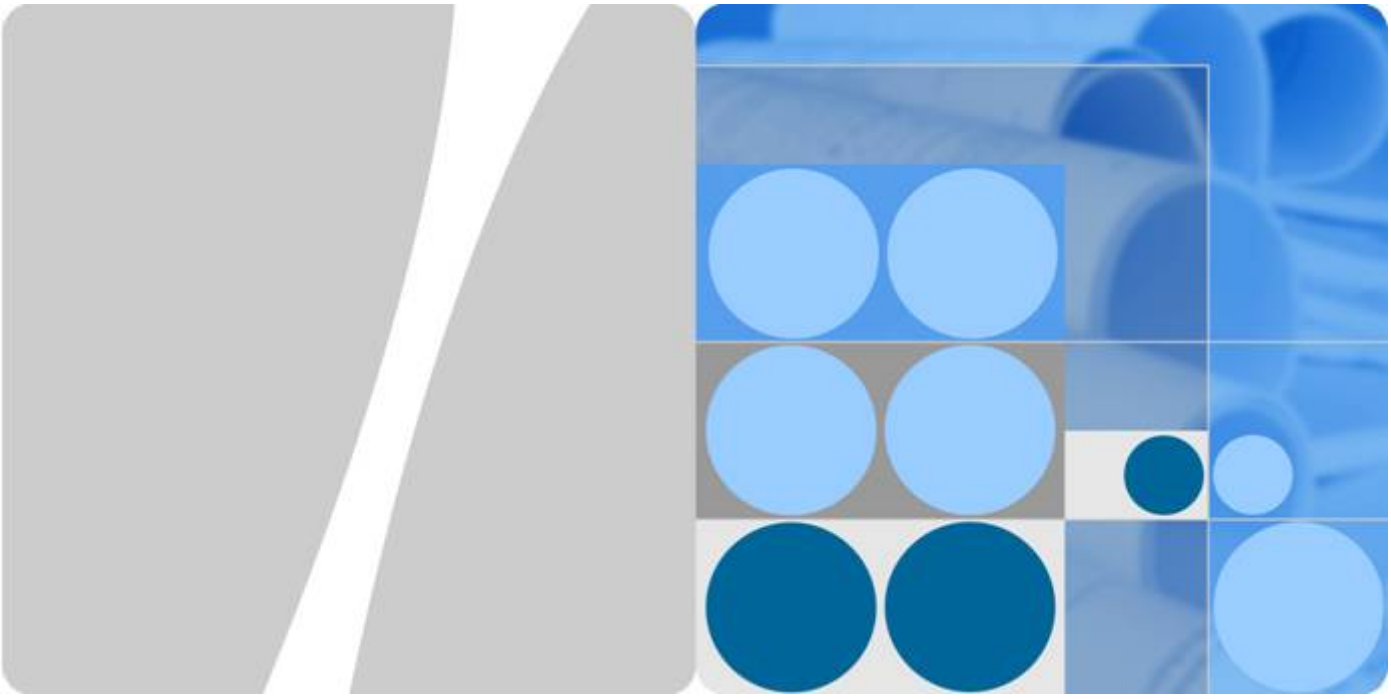


Document Database Service

Best Practices

Issue 01

Date 2026-06-25

HUAWEI TECHNOLOGIES CO., LTD.

HUAWEI

Trademarks and Permissions


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

Notice

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

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

Contents

1 Overview

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

Reference	Overview
Security Best Practices	This section provides actionable guidance for enhancing the overall security of using DDS.
Common Methods for Connecting to a DDS Instance	This section describes common DDS connection methods.
From Other Cloud MongoDB to DDS	These sections describe how to migrate DDS data.
From On-Premises MongoDB to DDS	
From ECS-hosted MongoDB to DDS	
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.

Reference	Overview
How Do I Troubleshoot High Memory Usage of DDS DB Instances?	This section describes how to troubleshoot high memory usage.
What Can I Do If the Number of Connections of an Instance Reaches Its Maximum?	This section describes how to troubleshoot the fault that there are too many connections to a DB instance.
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.
Proper Use of Data Definition Languages (DDL) Statements	This section describes DDL statements used to create, modify, and delete the structure of databases and collections.
How Is a DDS Node Going to Be Disconnected and What Can I Do?	This section describes the principles and workarounds of node disconnection.
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.
Oplog Size Planning and Optimization	This section describes oplog size planning and optimization.

2 Security Best Practices

Security is a shared responsibility between Huawei Cloud and you. Huawei Cloud is responsible for the security of cloud services to provide a secure cloud. As a tenant, you should properly use the security capabilities provided by cloud services to protect data, and securely use the cloud. For details, see **Shared Responsibilities**.

This section provides actionable guidance for enhancing the overall security of using DDS. You can continuously evaluate the security status of your DDS DB instances and enhance their overall security defense by combining different security capabilities provided by DDS. By doing this, data stored in DDS DB instances can be protected from leakage and tampering both at rest and in transit.

Make security configurations from the following dimensions to meet your business needs.

- **No Binding an EIP to Access DDS over the Internet**
- **No Using Weak Passwords**
- **No Using the Default Port**
- **Limiting the Maximum Number of DDS Connections**
- **Disabling IPv6**
- **Disabling Script Execution**
- **Configuring the Audit Policy**
- **Configuring Instance Access Logs**
- **Enabling SSL**
- **Enabling Disk Encryption**
- **Enabling Data Backups**
- **Configuring Monitoring by Seconds and Alarm Rules**
- **Upgrading the Version of a DB Instance**
- **Checking Roles**

No Binding an EIP to Access DDS over the Internet

Do not deploy DDS on the Internet or DMZ. Instead, deploy DDS on the internal network of your company and use routers or firewalls to protect DDS. Do not bind an EIP to access DDS from the Internet. By doing so, DDS can be protected from

unauthorized access and DDoS attacks. You are not advised **binding an EIP** to access DDS from the Internet. If necessary, you must **set security group rules**.

No Using Weak Passwords

When creating or changing an account password, ensure that the password meets the password complexity requirements and do not use weak passwords. By doing so, passwords can be protected from hacker and rainbow table attacks. For details about how to check for weak passwords, see **Checking for Weak Passwords**.

No Using the Default Port

The default port for MongoDB is 27017. If the default port is used, it is easy to be listened on, which poses security risks. You are advised to use a non-default port. For details, see **Changing a Database Port**.

Limiting the Maximum Number of DDS Connections

Excessive DDS connections will consume excessive server resources, leading to sluggish response of the OPS operations (such as **query**, **insert**, **update**, and **delete**). Also, you need to set **net.maxIncomingConnections** to an appropriate value based on the operating system environment. If the value is greater than the maximum number of threads received by the operating system, the setting is invalid. For details, see **Parameters**.

Disabling IPv6

IPv6 subnets are not supported. You are advised to select an IPv4 subnet when creating a DDS DB instance.

Disabling Script Execution

If the **security.javascriptEnabled** parameter is enabled, JavaScript scripts can be executed on mongod, which poses security risks. If the **javascriptEnabled** option is disabled, the **mapreduce** and **group** commands cannot be used. If your application does not require operations such as MapReduce, you are advised to disable **javascriptEnabled**. For details, see **Parameters**.

Configuring the Audit Policy

The audit function can be used to record all database operations performed by users. Auditing logs can enhance your database security and help you analyze the cause of failed operations to improve system O&M. For details, see **Audit Logs**.

Configuring Instance Access Logs

If you enable log reporting for your DDS instance, new audit, error, and slow query logs generated for the instance will be uploaded to Log Tank Service (LTS). You can view, search for, and download audit, error, and slow query logs of DDS instances. The log data is graphically displayed to make it easier to analyze and understand. For details, see **Log Reporting**.

Enabling SSL

If SSL is not configured, data transmitted between the MongoDB client and server is in plaintext, which is vulnerable to eavesdropping, tampering, and man-in-the-middle attacks. To improve the security of data transmission, you are advised to enable SSL. For details, see **Enabling or Disabling SSL**.

Enabling Disk Encryption

Enabling disk encryption improves data security. For details, see "Disk Encryption" in **Custom Config**.

Enabling Data Backups

DDS supports automated and manual backups. You can periodically back up databases. If a database is faulty or data is damaged, you can restore the database using backups to ensure data reliability. For details, see **Data Backups**.

Configuring Monitoring by Seconds and Alarm Rules

DDS DB instances can be monitored by default. 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 status of your DDS instance in a timely manner. Configure proper monitoring and alarm rules based on service requirements. For details, see **Monitoring and Alarm Reporting**.

Upgrading the Version of a DB Instance

DDS supports **minor version upgrade** and **major version update**. You can upgrade your DB instance to the latest version to add new functions, fix problems, and improve security and performance. You are advised to upgrade the version of a DB instance in a timely manner.

Checking Roles

DDS allows you to grant role-based permissions to a database account for data and command access. You are advised to create **user-defined roles** based on service requirements and grant the minimum permission to a database account. You can also **update** or **delete** a database user as needed.

Table 2-1 DDS role-based permissions

No.	Check Item	Description
1	The user with the userAdmin role	After the user with the userAdmin role is defined in the admin database, the user is provided with all permissions of all users. That is, the user with this role can define their own permissions on any database.

No.	Check Item	Description
2	The user with the userAdminAnyDatabase role	After the user with the userAdminAnyDatabase role is defined, the user is provided with all permissions of all users. That is, the user with this role can define their own permissions on any database.
3	The role with the anyAction action	After the role with the anyAction action is defined, the role-based user is provided with all operation permissions on the corresponding database, which affects permission management.
4	The role with the anyResource action	After the role with the anyResource action is defined, the role-based user is provided with all resource permissions on the corresponding database, which affects permission management.
5	The role with the changeCustomData action	After the role with the changeCustomData action is defined, the role-based user is provided with the permission to modify all user-defined information in the corresponding database, which affects permission management.
6	The role with the changePassword action	After the role with the changePassword action is defined, the role-based user is provided with the permission to change the passwords of all users in the corresponding database, which affects permission management.
7	The role with the createRole action	After the role with the createRole action is defined, the role-based user is provided with the permission to create all roles in the corresponding database, which affects permission management.
8	The role with the createUser action	After the role with the createUser action is defined, the role-based user is provided with the permission to create all users in the corresponding database, which affects permission management.
9	The role with the dropRole action	After the role with the dropRole action is defined, the role-based user is provided with the permission to delete any role in the corresponding database, which affects permission management.

No.	Check Item	Description
10	The role with the dropUser action	After the role with the dropUser action is defined, the role-based user is provided with the permission to delete any user in the corresponding database, which affects permission management.
11	The role with the grantRole action	After the role with the grantRole action is defined, the role-based user is provided with the permission to grant all role permissions to all users in the corresponding database, which affects permission management.
12	The role with the revokeRole action	After the role with the revokeRole action is defined, the role-based user is provided with the permission to revoke any role permission from any user in the corresponding database, which affects permission management.
13	The role with the authSchemaUpgrade action	After the role with the authSchemaUpgrade action is defined, the role-based user is provided with the permission to execute **authschemaupgrade**, which affects permission management. The **authschemaupgrade** command is used to modify the user authentication conversion format.
14	The role with the closeAllDatabases action	After the role with the closeAllDatabases action is defined, the role-based user is provided with the permission to run the **closeAllDatabases** command, which affects permission management. The **closeAllDatabases** command is used to close all databases and release the memory occupied by MongoDB.
15	The role with the dropDatabase action	After the role with the dropDatabase action is defined, the role-based user is provided with the permission to run the **dropDatabase** command to delete any database, which affects permission management.
16	The role with the getParameter action	After the role with the getParameter action is defined, the role-based user is provided with the permission to run the **getParameter** command to view the values of all command line options, which affects permission management.

No.	Check Item	Description
17	The role with the setParameter action	After the role with the setParameter action is defined, the role-based user is provided with the permission to run the **setParameter** command to change the value of any command line option, which affects permission management.
18	The role with the shutdown action	After the role with the shutdown action is defined, the role-based user is provided with the permission to run the **shutdown** command to clear all database resources and stop processes, which affects permission management.
19	The role with the getCmdLineOpts action	After the role with the getCmdLineOpts action is defined, the role-based user is provided with the permission to run the **getCmdLineOpts** command to obtain the **argv** and **parsed** fields, which affects permission management. The **argv** field contains the mongod or mongos command string, and the **parsed** field contains all runtime options.
20	The role with the internal action	After the role with the internal action is defined, the role-based user is provided with the permission to perform all operations on the corresponding database, which affects permission management.
21	The user with the readWrite role	After the user with the readWrite role is defined, the role-based user is provided with the read permission and data modification permission on the corresponding database, which affects permission management.
22	The user with the backup role	After the user with the backup role is defined, the role-based user is provided with the insert and update permissions in the **mms.bak** file in the admin database, which affects permission management.
23	The user with the clusterAdmin role	After the user with the clusterAdmin role is defined, the role-based user is provided with the highest cluster management permission, which affects permission management. The role has the permissions of the clusterManager, clusterMonitor, and hostManager roles.

No.	Check Item	Description
24	The user with the clusterManager role	After the user with the clusterManager role is defined to manage and monitor operations in a cluster, the role-based user is provided with the permission to manage local databases that are shared and replicated, which affects permission management.
25	The user with the clusterMonitor role	After the user with clusterMonitor role is defined, the role-based user is provided with the read-only permission for the monitoring tool, which affects permission management.
26	The user with the dbAdmin role	After the user with the dbAdmin role is defined, the role-based user is provided with the administrator permissions on the corresponding database, which affects permission management.
27	The user with the dbAdminAnyDatabase role	After the user with the dbAdminAnyDatabase role is defined, the role-based user is provided with the same permissions as the dbAdmin role, which affects permission management. The role applies to all databases in a cluster, and also has the listDatabases operation permission on the cluster.
28	The user with the dbOwner role	After the user with the dbOwner role is defined, the role-based user is provided with the permission to perform all database management operations, which affects permission management. This role has the permissions of the readWrite, dbAdmin, and userAdmin roles.
29	The user with the hostManager role	After the user with hostManager role is defined, the role-based user is provided with the permission to monitor and manage servers, which affects permission management.
30	The user with the readAnyDatabase role	After the user with readAnyDatabase role is defined, the role-based user is provided with the permission to read data from all databases, which affects permission management. This role also has the listdatabases operation permission on the cluster.

No.	Check Item	Description
31	The user with the readWriteAnyDatabase role	After the user with readWriteAnyDatabase role is defined, the role-based user is provided with the permission to read data from and write data to all databases, which affects permission management. This role also has the listdatabases operation permission on the cluster.
32	The user with the restore role	After the user with restore role is defined, the role-based user is provided with the permission required for restoring backups, which affects permission management.
33	The user with the root role	After the user with root role is defined, the role-based user is provided with all operation permissions on all resources, which affects permission management. The role has the permissions of the readWriteAnyDatabase, dbAdminAnyDatabase, userAdminAnyDatabases, clusterAdmin, and restore roles.
34	The user with the userAdmin role	After the user with the userAdmin role is defined, the role-based user is provided with the permissions of all users, including their own permissions, which affects permission management.
35	The user with the userAdminAnyDatabase role	After the user with the userAdminAnyDatabase role is defined, the role-based user is provided with the permissions of all users on all databases, including their own permissions, which affects permission management.

3 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 3-1 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 *Document Database Service User Guide*.

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/current/security/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 3-2 Parameter description

Parameter	Description
<username>	Current username.
<password>	Password for the current username.
<instance_ip>	If you access an instance from an ECS, *instance_ip* is the private IP address 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 3-3**.

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

Table 3-3 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 3-4 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.

4 From Other Cloud MongoDB to DDS

DRS helps you migrate MongoDB databases from other cloud platforms to DDS on the current cloud. With DRS, you can migrate databases online with zero downtime and your services and databases can remain operational during migration.

This section describes how to use DRS to migrate MongoDB databases from another cloud to DDS on the current cloud. Migration scenarios include:

- Migrating MongoDB databases from another cloud to DDS on the current cloud.
- Migrating self-built MongoDB databases from servers on another cloud to DDS on the current cloud.

Resource Planning

Table 4-1 Resource planning

Category	Subcategory	Planned Value	Description
VPC	VPC name	vpc-dds	Specify a name that is easy to identify.
	Region	AP-Singapore	To achieve lower network latency, select the region nearest to you.
	AZ	AZ1	-
	Subnet	10.0.0.0/24	Select a subnet with sufficient network resources.
	Subnet name	subnet-default	Specify a name that is easy to identify.

Category	Subcategory	Planned Value	Description
MongoDB database on other clouds	Database version	MongoDB 4.4	-
	IP address	192.168.0.1	This is only an example.
	Port	8635	This is only an example.
DDS instance	Instance name	dds-test	Specify a name that is easy to identify.
	Database version	DDS 4.4	-
	AZ type	Single-AZ	This is only an example.
	AZ	AZ1	AZ1 is selected in this example.
	Instance specifications	Enhanced II 4 vCPUs | 16 GB	The specifications in the left column are selected in this example. You need to select specifications meeting your workload requirements.
DRS migration task	Task name	DRS-test-migrate	Specify a name that is easy to identify.
	Source DB engine	MongoDB	-
	Destination DB engine	DDS	-
	Network type	Public network	Public network is used in this example.

Diagram

Figure 4-1 Migrating MongoDB databases from other clouds

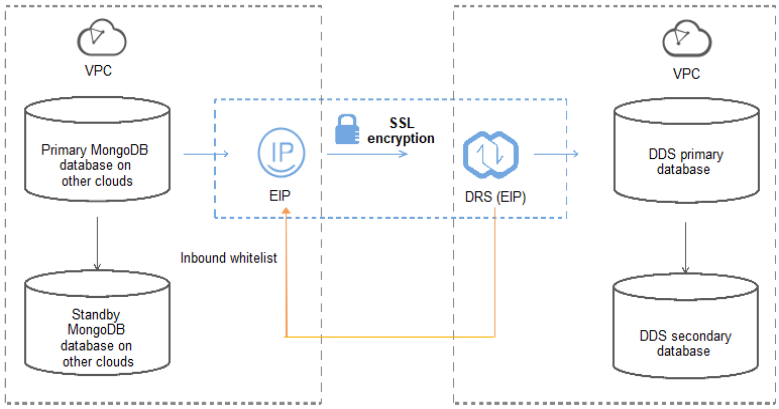
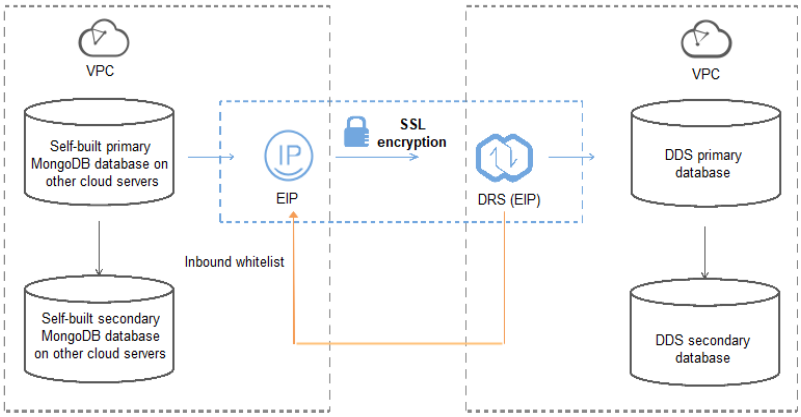
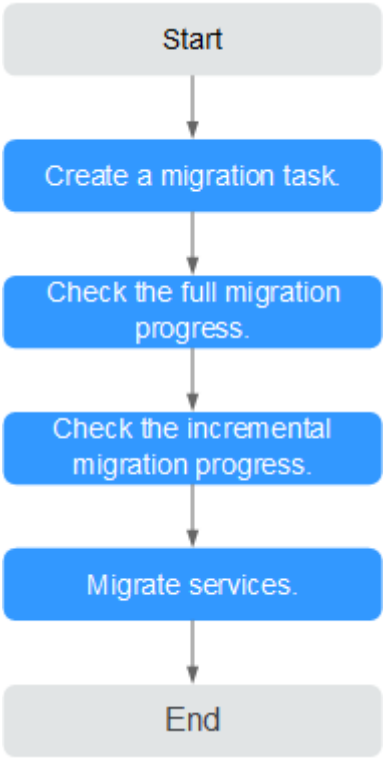


Figure 4-2 Migrating MongoDB databases from other cloud servers



Migration Process

Figure 4-3 Flowchart



Migration Suggestions (Important)

- Database migration is closely impacted by a wide range of environmental and operational factors. To ensure the migration goes smoothly, perform a test run before the actual migration to help you detect and resolve any potential issues in advance. Recommendations on how to minimize any potential impacts on your data base are provided in this section.
- It is strongly recommended that you start your migration task during off-peak hours. A less active database is easier to migrate successfully. If the data is fairly static, there is less likely to be any severe performance impacts during the migration.

Notes on Migration (Important)

NOTICE

Before creating a migration task, read the migration notes carefully.

For details, see **precautions** on using specific migration tasks in *Data Replication Service Real-Time Migration*.

Preparations

1. Permissions

   Table 4-2 lists the permissions required for the source and destination databases when migrating a MongoDB database from another cloud to DDS on the current cloud.

Table 4-2 Required permissions

Database	Full Migration Permission	Full+Incremental Migration Permission
Source	• Replica set: The source database user must have the read permission for the database to be migrated. • Single node: The source database user must have the read permission for the database to be migrated. • Cluster: The source database user must have the read permission for the databases to be migrated and the config database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.	• Replica set: The source database user must have the read permission for the databases to be migrated and the local database. • Single node: The source database user must have the read permission for the databases to be migrated and the local database. • Cluster: The source mongos node user must have the readAnyDatabase permission for the databases to be migrated and the config database. The source shard node user must have the readAnyDatabase permission for the admin database and the read permission for the local database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.
Destination	The destination database user must have the dbAdminAnyDatabase permission for the admin database and the readWrite permission for the destination database. If the destination database is a cluster instance, the migration account must have the read permission for the config database.	

- Source database permissions:

  The source MongoDB database user must have all the required permissions listed in **Table 4-2**. If the permissions are insufficient, create a user that has all of the permissions on the source database.

- Destination database permissions:

  If the destination database is a DDS database, the initial account can be used.

2. Network settings

   Enable public accessibility for the source database.

   - Source database network settings:

     Any source database MongoDB instances will need to be accessible from the Internet.

   - Destination database network settings: No settings are required.

3. Security rules

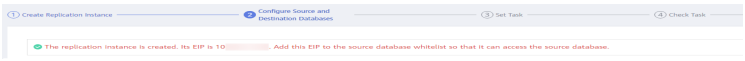
   - Source database security group settings:

     The replication instance needs to be able to access the source MongoDB instance. That means that the EIP of the replication instance must be on the whitelist of the source MongoDB instance.

     Before configuring the network whitelist, you need to obtain the EIP of the replication instance.

     - After creating a replication instance on the DRS console, you can find the EIP on the **Configure Source and Destination Databases** page as shown in **Figure 4-4**.

       Figure 4-4 EIP of the replication instance

     You can also add 0.0.0.0/0 to the source database whitelist to allow any IP address to access the source database but this action may result in security risks.

     If you do take this step, then once the migration is complete, you should delete this item from the whitelist or your system will insecure.

   - Destination database security group settings:

     By default, the destination database and the DRS replication instance are in the same VPC and can communicate with each other. No further configuration is required.

4. Other

   You need to export the user information of the MongoDB database first and manually add it to the destination DDS DB instance because the user information will not be migrated.

Migration Procedure

Step 1 Create a migration task.

1. Log in to the management console and choose **Databases** > **Data Replication Service** to go to the DRS console.
2. On the **Online Migration Management** page, click **Create Migration Task**.
3. On the **Replication Instance Information** page, configure the task details, description, and replication instance details and click **Next**.

Figure 4-5 Replication instance information

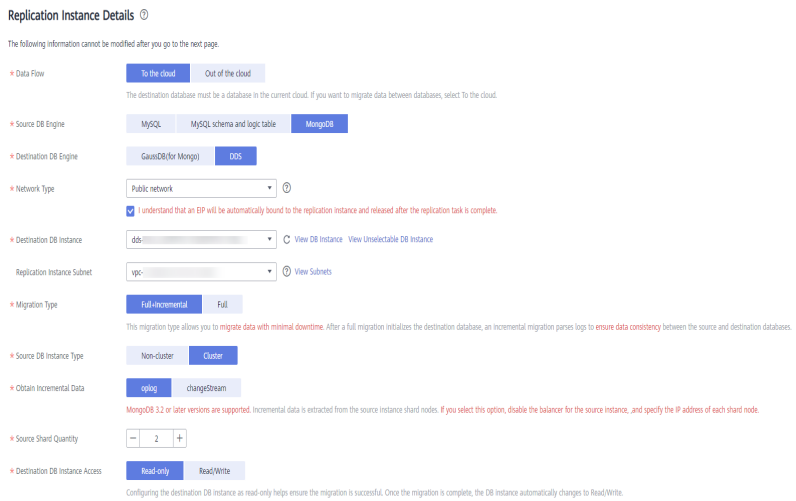


Table 4-3 Task settings

Parameter	Description
Region	The region where the replication instance is deployed. You can change the region. To reduce latency and improve access speed, select the region closest to your workloads.
Project	The project corresponds to the current region and can be changed.
Task Name	The task name consists of 4 to 50 characters, starts with a letter, and can contain only letters (case-insensitive), digits, hyphens (-), and underscores (_).
Description	The description consists of a maximum of 256 characters and cannot contain the following special characters: =<>&'\"

Table 4-4 Replication instance settings

Parameter	Description
Data Flow	**To the cloud**

Parameter	Description
Source DB Engine	Select **MongoDB**.
Destination DB Engine	Select **DDS**.
Network Type	Select **Public network**.
Destination DB Instance	The DDS DB instance you purchased.
Replication Instance Subnet	The subnet where the replication instance resides. You can also click **View Subnet** to go to the network console to view the subnet where the instance resides. By default, the DRS instance and the destination DB instance are in the same subnet. You need to select the subnet where the DRS instance resides, and there are available IP addresses for the subnet. To ensure that the replication instance is successfully created, only subnets with DHCP enabled are displayed.
Migration Type	– **Full** This migration type is suitable for scenarios where service interruption is acceptable. All objects in non-system databases are migrated to the destination database at one time. The objects include collections and indexes. – **Full+Incremental** The full+incremental migration type allows you to migrate data without interrupting services. After a full migration initializes the destination database, an incremental migration parses logs to ensure data consistency between the source and destination databases.
Source DB Instance Type	If you select **Full+Incremental** for **Migration Type**, set this parameter based on the source database. – If the source database is a cluster instance, set this parameter to **Cluster**. – If the source database is a replica set or a single node instance, set this parameter to **Non-cluster**.

Parameter	Description
Obtain Incremental Data	This parameter is available for configuration if **Source DB Instance Type** is set to **Cluster**. You can determine how to capture data changes during the incremental synchronization. – oplog: For MongoDB 3.2 or later, DRS directly connects to each shard of the source DB instance to extract data. If you select this mode, you must disable the balancer of the source instance. When testing the connection, you need to enter the connection information of each shard node of the source instance. – changeStream: This method is recommended. For MongoDB 4.0 and later, DRS connects to mongos nodes of the source instance to extract data. If you select this method, you must enable the WiredTiger storage engine of the source instance. **NOTE** Only whitelisted users can use **changeStream**. To use this function, submit a service ticket. In the upper right corner of the management console, choose **Service Tickets** > **Create Service Ticket** to submit a service ticket.
Source Shard Quantity	If **Source DB Instance Type** is set to **Cluster** and **Obtain Incremental Data** is set to **oplog**, enter the number of source shard nodes. The default minimum number of source DB instances is 2 and the maximum number is 32. You can set this parameter based on the number of source database shards.

4. On the **Configure Source and Destination Databases** page, wait until the replication instance is created. Then, specify source and destination database information and click **Test Connection** for both the source and destination databases to check whether they have been connected to the replication instance. After the connection tests are successful, select the check box before the agreement and click **Next**.

Figure 4-6 Source database information

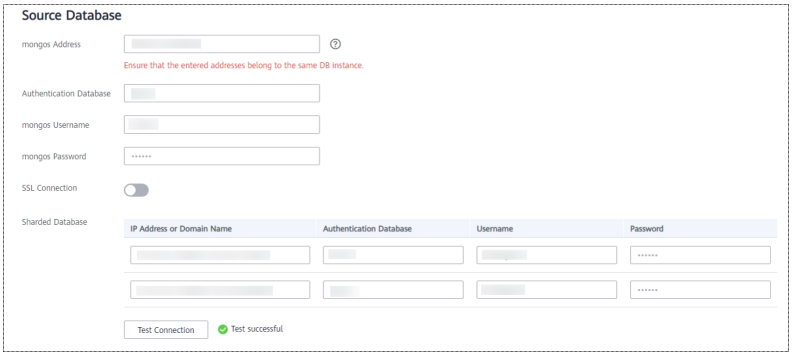


Table 4-5 Source database settings

Parameter	Description
mongos Address	IP address or domain name of the source database in the **IP address/Domain name:Port** format. The port of the source database. Range: 1 - 65534 You can enter a maximum of three groups of IP addresses or domain names of the source database. Separate multiple values with commas (,). For example: 192.168.0.1:8080,192.168.0.2:8080. Ensure that the entered IP addresses or domain names belong to the same sharded cluster. **NOTE** If multiple IP addresses or domain names are entered, the test connection is successful as long as one IP address or domain name is accessible. Therefore, you must ensure that the IP address or domain name is correct.
Authentication Database	The name of the authentication database. For example: The default authentication database of Huawei Cloud DDS instance is **admin**.
mongos Username	A username for the source database.
mongos Password	The password for the source database username.
SSL Connection	SSL encrypts the connections between the source and destination databases. If SSL is enabled, upload the SSL CA root certificate.
Sharded Database	Enter the information about the sharded databases in the source database.

– Destination database configuration

Figure 4-7 Destination database information

Table 4-6 Destination database settings

Parameter	Description
DB Instance Name	The DB instance you selected when creating the migration task and cannot be changed.
Database Username	The username for accessing the destination database.
Database Password	The password for the database username.

5. On the **Set Task** page, select migration objects and click **Next**.

Figure 4-8 Migration object

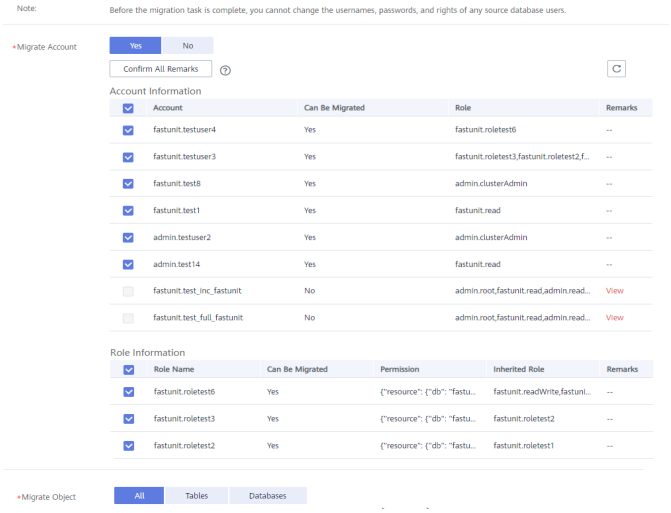


Table 4-7 Migration object

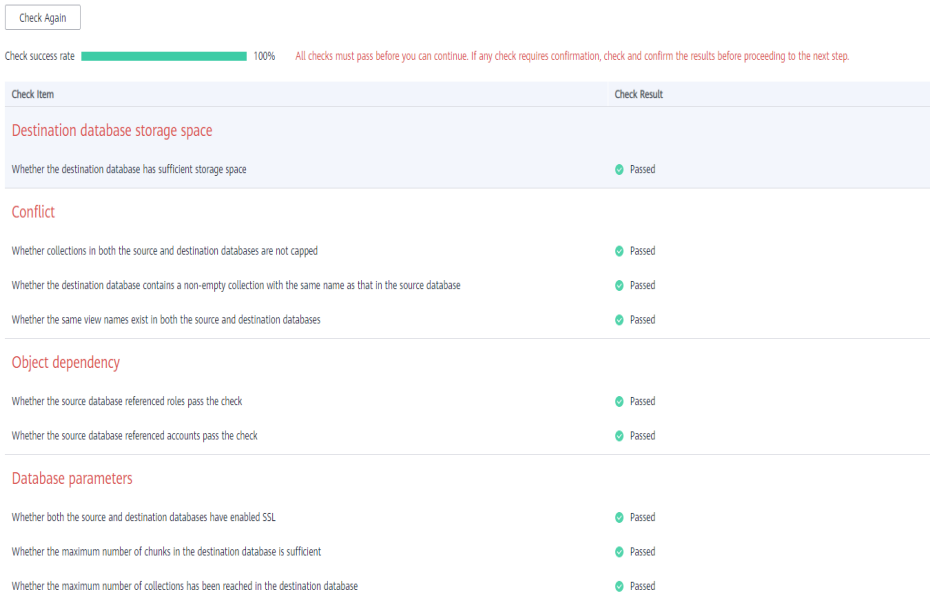
Parameter	Description
Migrate Account	There are accounts that can be migrated completely and accounts that cannot be migrated. You can choose whether to migrate the accounts. Accounts that cannot be migrated or accounts that are not selected will not exist in the destination database. Ensure that your services will not be affected by these accounts. – **Yes** If you choose to migrate accounts, see **Migrating Accounts** in *Data Replication Service User Guide* to migrate database users and roles. – **No** During the migration, accounts and roles are not migrated.

Parameter	Description
Migrate Object	You can choose to migrate all objects, tables, or databases based on your service requirements. – **All**: All objects in the source database are migrated to the destination database. After the migration, the object names will remain the same as those in the source database and cannot be modified. – **Tables**: The selected table-level objects will be migrated. – **Databases**: The selected database-level objects will be migrated. If the source database is changed, click in the upper right corner before selecting migration objects to ensure that the objects to be selected are from the changed source database. **NOTE** – If you choose not to migrate all of the databases, the migration may fail because the objects, such as stored procedures and views, in the database to be migrated may have dependencies on other objects that are not migrated. To ensure a successful migration, you are advised to migrate all of the databases. – When you select an object, the spaces before and after the object name are not displayed. If there are two or more consecutive spaces in the middle of the object name, only one space is displayed. – The search function can help you quickly select the required database objects.

6. On the **Check Task** page, check the migration task.
   - If any check fails, review the cause and rectify the fault. After the fault is rectified, click **Check Again**.

     For details about how to handle check failures, see **Checking Whether the Source Database Is Connected** in *Data Replication Service User Guide*.
   - If all check items are successful, click **Next**.

Figure 4-9 Task Check



NOTE

You can proceed to the next step only when all check items are successful. If any alarms are generated, view and confirm the alarm details first before proceeding to the next step.

7. On the displayed page, specify **Start Time**, **Send Notification**, **SMN Topic**, **Synchronization Delay Threshold**, and **Stop Abnormal Tasks After** and confirm that the configured information is correct and click **Submit** to submit the task.

Figure 4-10 Task startup settings

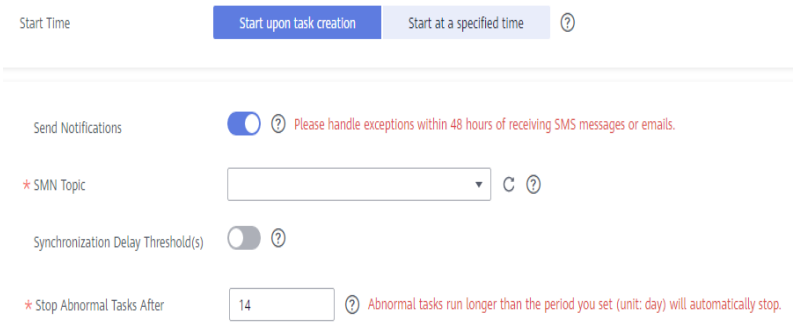


Table 4-8 Task startup settings

Parameter	Description
Start Time	Set **Start Time** to **Start upon task creation** or **Start at a specified time** based on site requirements. The **Start at a specified time** option is recommended. **NOTE** The migration task may affect the performance of the source and destination databases. You are advised to start the task in off-peak hours and reserve two to three days for data verification.
Send Notifications	SMN topic. This parameter is optional. If an exception occurs during migration, the system will send a notification to the specified recipients.
SMN Topic	This parameter is available only after you enable **Send Notification** and create a topic on the SMN console and add a subscriber. For details, see **Simple Message Notification User Guide**.
Synchronization Delay Threshold	During an incremental migration, a synchronization delay indicates a time difference (in seconds) of synchronization between the source and destination database. If the synchronization delay exceeds the threshold you specify, DRS will send alarms to the specified recipients. The value ranges from 0 to 3,600. To avoid repeated alarms caused by the fluctuation of delay, an alarm is sent only after the delay has exceeded the threshold for six minutes. **NOTE** – In the early stages of an incremental migration, there is more delay because more data is waiting to be synchronized. In this situation, no notifications will be sent. – Before setting the delay threshold, enable **Send Notification**. – If the delay threshold is set to 0, no notifications will be sent to the recipient.
Stop Abnormal Tasks After	Number of days after which an abnormal task is automatically stopped. The value must range from 14 to 100. The default value is **14**. **NOTE** Tasks in the abnormal state are still charged. If tasks remain in the abnormal state for a long time, they cannot be resumed. Abnormal tasks run longer than the period you set (unit: day) will automatically stop to avoid unnecessary fees.

8. After the task is submitted, go back to the **Online Migration Management** page to view the task status.

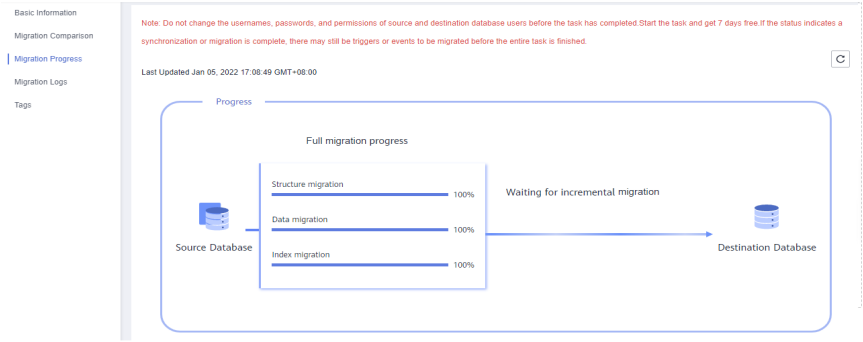
Step 2 Manage the migration task.

The migration task contains two phases: full migration and incremental migration. You can manage them in different phases.

- Full migration

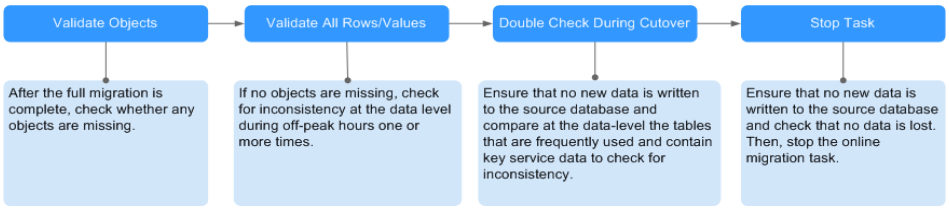
  - Viewing the migration progress: Click the target full migration task, and on the **Migration Progress** tab, you can see the migration progress of the structure, data, indexes, and migration objects. When the progress reaches 100%, the migration is complete.
  - Viewing migration details: In the migration details, you can view the migration progress of a specific object. If the number of objects is the same as that of migrated objects, the migration is complete. You can view the migration progress of each object in detail. Currently, this function is available only to whitelisted users. You can submit a service ticket to apply for this function.
- Incremental Migration Permission
  - Viewing the synchronization delay: After the full migration is complete, an incremental migration starts. On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Progress** to view the synchronization delay of the incremental migration. If the synchronization delay is 0s, the destination database is being synchronized with the source database in real time. You can also view the data consistency on the **Migration Comparison** tab.

    Figure 4-11 Viewing the synchronization delay



  - Viewing the migration results: On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Comparison** and perform a migration comparison in accordance with the comparison process, which should help you determine an appropriate time for migration to minimize service downtime.

    Figure 4-12 Database comparison process



    For details, see **Comparing Migration Items** in *Data Replication Service User Guide*.

Step 3 Cut over services.

You are advised to start the cutover process during off-peak hours. At least one complete data comparison is performed during off-peak hours. To obtain accurate

comparison results, start data comparison at a specified time point during off-peak hours. If it is needed, select **Start at a specified time** for **Comparison Time**. Due to slight time difference and continuous operations on data, inconsistent comparison results may be generated, reducing the reliability and validity of the results.

1. Interrupt services first. If the workload is not heavy, you may not need to interrupt the services.
2. Run the following statement on the source database and check whether any new sessions execute SQL statements within the next 1 to 5 minutes. If there are no new statements executed, the service has been stopped.
   ```
   db.currentOp()
   ```

   NOTE

   The process list queried by the preceding statement includes the connection of the DRS replication instance. If no additional session executes SQL statements, the service has been stopped.
3. On the **Migration Progress** page, view the synchronization delay. When the delay is displayed as 0s and remains stable for a period, then you can perform a data-level comparison between the source and destination databases. For details about the time required, refer to the results of the previous comparison.
   - If there is enough time, compare all objects.
   - If there is not enough time, use the data-level comparison to compare the tables that are frequently used and that contain key business data or inconsistent data.
4. Determine an appropriate time to cut the services over to the destination database. After services are restored and available, the migration is complete.

Step 4 Stop or delete the migration task.

1. Stopping the migration task. After databases and services are migrated to the destination database, to prevent operations on the source database from being synchronized to the destination database to overwrite data, you can stop the migration task. This operation only deletes the replication instance, and the migration task is still displayed in the task list. You can view or delete the task. DRS will not charge for this task after you stop it.
2. Delete the migration task. After the migration task is complete, you can delete it. After the migration task is deleted, it will no longer be displayed in the task list.

----End

5 From On-Premises MongoDB to DDS

DRS helps you migrate data from on-premises MongoDB databases to DDS on the current cloud. With DRS, you can migrate databases online with zero downtime and your services and databases can remain operational during migration.

This section describes how to use DRS to migrate an on-premises MongoDB database to DDS on the current cloud. The following network types are supported:

- VPN
- Public network

Resource Planning

Table 5-1 Resource planning

Category	Subcategory	Planned Value	Description
VPC	VPC name	vpc-dds	Specify a name that is easy to identify.
	Region	AP-Singapore	To achieve lower network latency, select the region nearest to you.
	AZ	AZ1	-
	Subnet	10.0.0.0/24	Select a subnet with sufficient network resources.
	Subnet name	subnet-default	Specify a name that is easy to identify.
On-premises MongoDB database	Database version	MongoDB 4.4	Specify a name that is easy to identify.
DDS	DDS instance name	dds-test	Specify a name that is easy to identify.
	DB engine	DDS	-
	DB engine version	4.4	-

Category	Subcategory	Planned Value	Description
	AZ type	Single-AZ	-
	AZ	AZ1	-
	Instance specifications	Enhanced II	-
	CPU architecture	x86 8 vCPUs | 32 GB	-
DRS migration task	Task name	DRS-dds	Specify a name that is easy to identify.
	Source DB engine	MongoDB	In this example, the source is an on-premises MongoDB database.
	Destination DB engine	DDS	In this example, a DDS instance serves as the destination database.
	Network type	Public network	Public network is used in this example.

Diagram

Figure 5-1 VPN

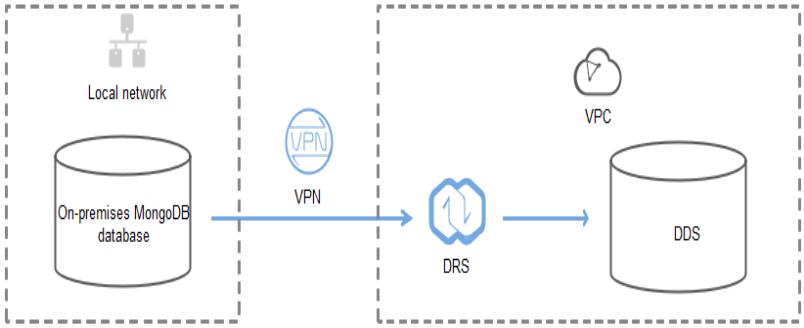
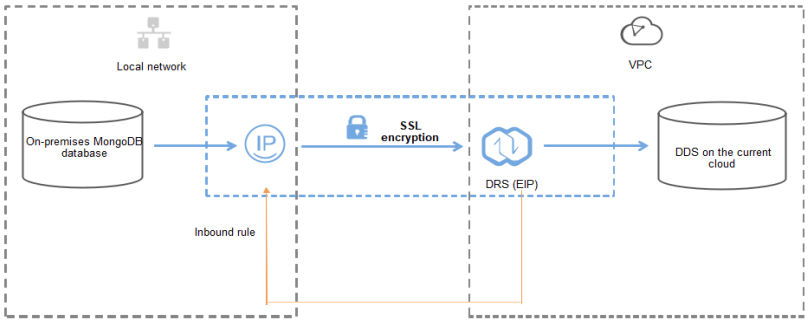
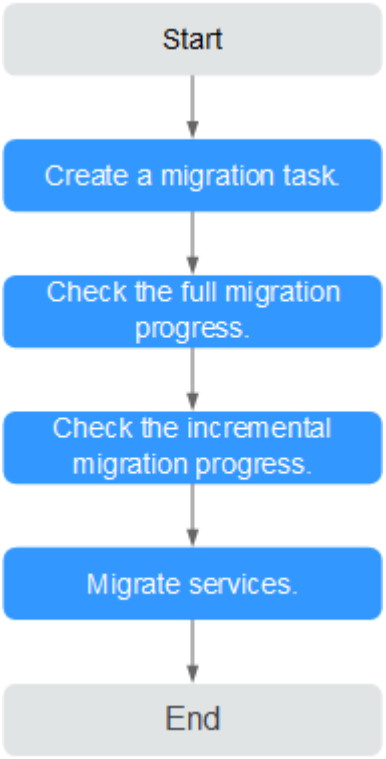


Figure 5-2 Public network+SSL connection



Migration Process

Figure 5-3 Flowchart



Migration Suggestions (Important)

- Database migration is closely impacted by a wide range of environmental and operational factors. To ensure the migration goes smoothly, perform a test run before the actual migration to help you detect and resolve any potential issues in advance. Recommendations on how to minimize any potential impacts on your data base are provided in this section.
- It is strongly recommended that you start your migration task during off-peak hours. A less active database is easier to migrate successfully. If the data is fairly static, there is less likely to be any severe performance impacts during the migration.

Notes on Migration (Important)

NOTICE

Before creating a migration task, read the migration notes carefully.

For details, see **precautions** on using specific migration tasks in *Data Replication Service Real-Time Migration*.

Preparations

1. Permissions

   Table 5-2 lists the permissions required for the source and destination databases when migrating data from on-premises MongoDB databases to DDS DB instances.

   Table 5-2 Required permissions

Database	Full Migration Permission	Full+Incremental Migration Permission
Source	• Replica set: The source database user must have the read permission for the database to be migrated. • Single node: The source database user must have the read permission for the database to be migrated. • Cluster: The source database user must have the read permission for the databases to be migrated and the config database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.	• Replica set: The source database user must have the read permission for the databases to be migrated and the local database. • Single node: The source database user must have the read permission for the databases to be migrated and the local database. • Cluster: The source mongos node user must have the readAnyDatabase permission for the databases to be migrated and the config database. The source shard node user must have the readAnyDatabase permission for the admin database and the read permission for the local database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.

Database	Full Migration Permission	Full+Incremental Migration Permission
Destination	The destination database user must have the dbAdminAnyDatabase permission for the admin database and the readWrite permission for the destination database. If the destination database is a cluster instance, the migration account must have the read permission for the config database.	

- Source database permissions:

  The source database user must have all the required permissions listed in **Table 5-2**. If the permissions are insufficient, create a user that has all of the permissions on the source database.
- Destination database permissions:

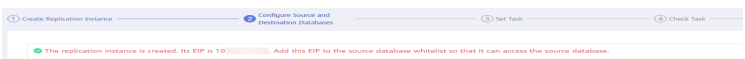
  If the destination database is a DDS database, the initial account can be used.

2. Network settings
   - Source database network settings:

     You can migrate on-premises MongoDB databases to DDS through a VPN or public network. Enable public accessibility or establish a VPN for local MongoDB databases based on the site requirements. You are advised to migrate data through a public network, which is more convenient and cost-effective.
   - Destination database network settings:
     - If the source database accesses the destination database through a VPN, enable the VPN service first so that the source database can communicate with the destination DDS network.
     - If you access the DDS DB instance through a public network, no network settings are required.
3. Security rules
   a. Source database network settings:
      - The replication instance needs to be able to access the source DB. That means that the EIP of the replication instance must be on the whitelist of the source MongoDB instance. Before configuring the network whitelist for the source database, you need to obtain the EIP of the DRS replication instance.

        After creating a replication instance on the DRS console, you can find the EIP on the **Configure Source and Destination Databases** page as shown in **Figure 5-4**.

        Figure 5-4 EIP of the replication instance

You can also add 0.0.0.0/0 to the source database whitelist to allow any IP address to access the source database but this action may result in security risks.

- If the migration is performed over a VPN network, add the private IP address of the DRS replication instance to the whitelist of the source database to enable the source database to communicate with the destination database.

  If you do take this step, then once the migration is complete, you should delete this item from the whitelist or your system will insecure.

b. Destination database security group settings:

   By default, the destination database and the DRS replication instance are in the same VPC and can communicate with each other. No further configuration is required.

4. Other

   You need to export the user information of the MongoDB database first and manually add it to the destination DDS DB instance because the user information will not be migrated.

Migration Procedure

The following describes how to use DRS to migrate an on-premises MongoDB database to a DDS DB instance.

Step 1 Create a migration task.

1. Log in to the management console and choose **Databases** > **Data Replication Service** to go to the DRS console.
2. On the **Online Migration Management** page, click **Create Migration Task**.
3. On the **Create Replication Instance** page, configure the task details, recipient, and replication instance and click **Next**.

Figure 5-5 Replication instance information

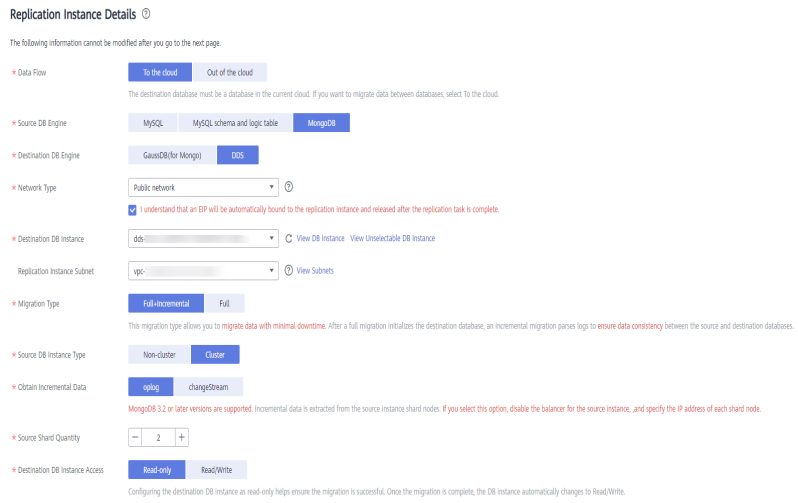


Table 5-3 Task settings

Parameter	Description
Region	The region where the replication instance is deployed. You can change the region. To reduce latency and improve access speed, select the region closest to your workloads.
Project	The project corresponds to the current region and can be changed.
Task Name	The task name consists of 4 to 50 characters, starts with a letter, and can contain only letters (case-insensitive), digits, hyphens (-), and underscores (_).
Description	The description consists of a maximum of 256 characters and cannot contain the following special characters: =<>&'\"

Table 5-4 Replication instance settings

Parameter	Description
Data Flow	Select **To the cloud**.
Source DB Engine	Select **MongoDB**.
Destination DB Engine	Select **DDS**.
Network Type	Select **Public network**. Enabling SSL is recommended. It may slow down the migration by 20% to 30% but it ensures data security.
Destination DB Instance	The DDS DB instance you purchased.
Migration Type	– **Full** It migrates all data at one time. If you perform a full migration, you are advised to stop operations on the source database. Otherwise, data generated in the source database during the migration will not be synchronized to the destination database. – **Full+Incremental** An incremental migration can keep data consistency after a full migration is complete.
Source DB Instance Type	If you select **Full+Incremental** for **Migration Type**, set this parameter based on the source database. – If the source database is a cluster instance, set this parameter to **Cluster**. – If the source database is a replica set or a single node instance, set this parameter to **Non-cluster**.

Parameter	Description
Obtain Incremental Data	This parameter is available for configuration if **Source DB Instance Type** is set to **Cluster**. You can determine how to capture data changes during the incremental synchronization. – oplog: For MongoDB 3.2 or later, DRS directly connects to each shard of the source DB instance to extract data. If you select this mode, you must disable the balancer of the source instance. When testing the connection, you need to enter the connection information of each shard node of the source instance. – changeStream: This method is recommended. For MongoDB 4.0 and later, DRS connects to mongos nodes of the source instance to extract data. If you select this method, you must enable the WiredTiger storage engine of the source instance. **NOTE** Only whitelisted users can use **changeStream**. To use this function, submit a service ticket. In the upper right corner of the management console, choose **Service Tickets** > **Create Service Ticket** to submit a service ticket.
Source Shard Quantity	If **Source DB Instance Type** is set to **Cluster** and **Obtain Incremental Data** is set to **oplog**, enter the number of source shard nodes. The default minimum number of source DB instances is 2 and the maximum number is 32. You can set this parameter based on the number of source database shards.

4. On the **Configure Source and Destination Databases** page, wait until the replication instance is created. Then, specify source and destination database information and click **Test Connection** for both the source and destination databases to check whether they have been connected to the replication instance. After the connection tests are successful, select the check box before the agreement and click **Next**.

Figure 5-6 Source database information

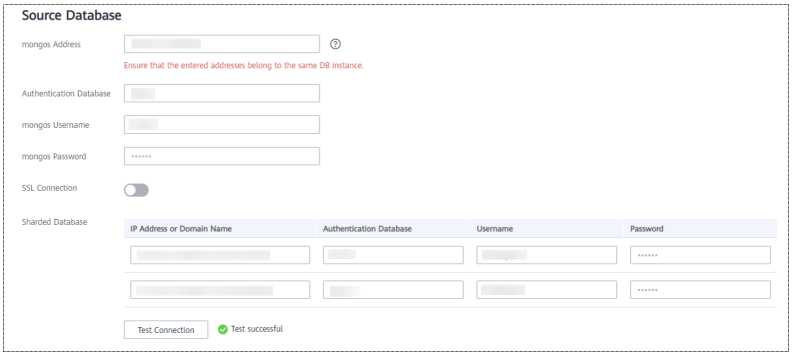


Table 5-5 Source database settings

Parameter	Description
mongos Address	IP address or domain name of the source database in the **IP address/Domain name:Port** format. The port of the source database. Range: 1 - 65534 You can enter a maximum of three groups of IP addresses or domain names of the source database. Separate multiple values with commas (,). For example: 192.168.0.1:8080,192.168.0.2:8080. Ensure that the entered IP addresses or domain names belong to the same sharded cluster. **NOTE** If multiple IP addresses or domain names are entered, the test connection is successful as long as one IP address or domain name is accessible. Therefore, you must ensure that the IP address or domain name is correct.
Authentication Database	The name of the authentication database. For example: The default authentication database of Huawei Cloud DDS instance is **admin**.
mongos Username	A username for the source database.
mongos Password	The password for the source database username.
SSL Connection	SSL encrypts the connections between the source and destination databases. If SSL is enabled, upload the SSL CA root certificate.
Sharded Database	Enter the information about the sharded databases in the source database.

– Destination database configuration

Figure 5-7 Destination database information

Table 5-6 Destination database settings

Parameter	Description
DB Instance Name	The DB instance you selected when creating the migration task and cannot be changed.
Database Username	The username for accessing the destination database.
Database Password	The password for the database username.

5. On the **Set Task** page, select migration objects and click **Next**.

Figure 5-8 Migration object

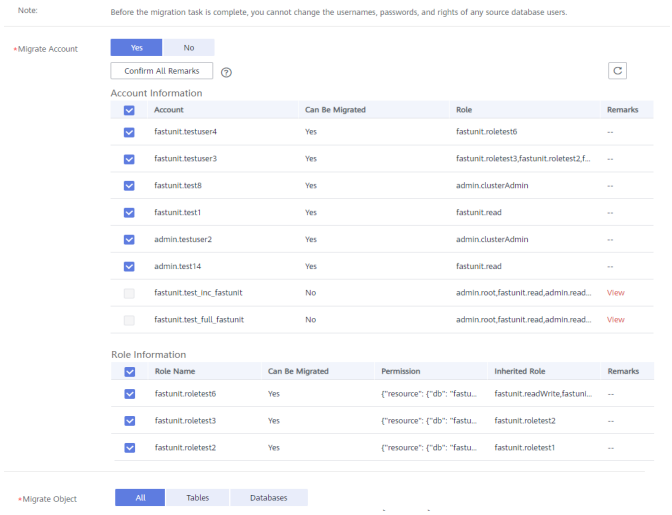


Table 5-7 Migration object

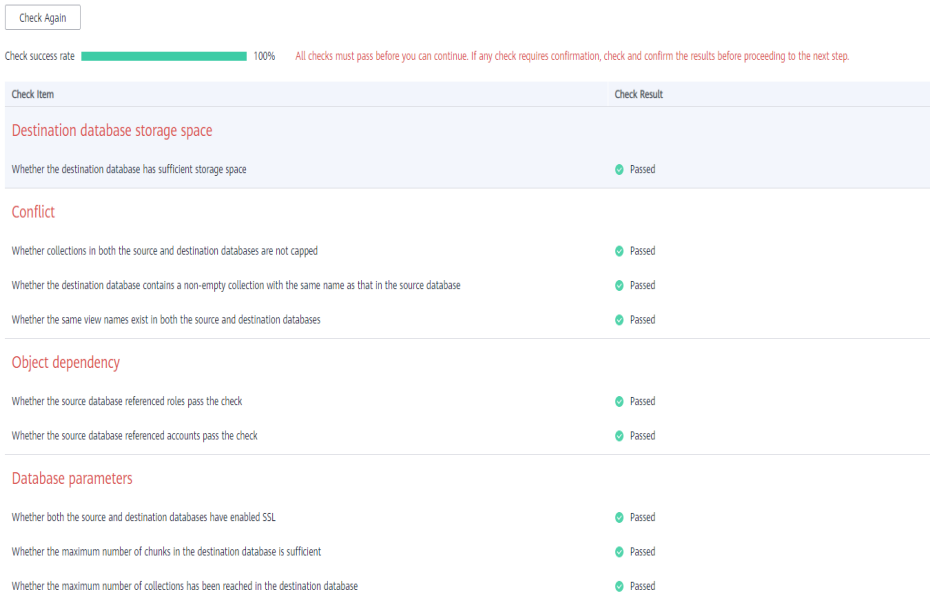
Parameter	Description
Migrate Account	There are accounts that can be migrated completely and accounts that cannot be migrated. You can choose whether to migrate the accounts. Accounts that cannot be migrated or accounts that are not selected will not exist in the destination database. Ensure that your services will not be affected by these accounts. – **Yes** If you choose to migrate accounts, see **Migrating Accounts** in *Data Replication Service User Guide* to migrate database users and roles. – **No** During the migration, accounts and roles are not migrated.

Parameter	Description
Migrate Object	You can choose to migrate all objects, tables, or databases based on your service requirements. – **All**: All objects in the source database are migrated to the destination database. After the migration, the object names will remain the same as those in the source database and cannot be modified. – **Tables**: The selected table-level objects will be migrated. – **Databases**: The selected database-level objects will be migrated. If the source database is changed, click in the upper right corner before selecting migration objects to ensure that the objects to be selected are from the changed source database. **NOTE** – If you choose not to migrate all of the databases, the migration may fail because the objects, such as stored procedures and views, in the database to be migrated may have dependencies on other objects that are not migrated. To ensure a successful migration, you are advised to migrate all of the databases. – When you select an object, the spaces before and after the object name are not displayed. If there are two or more consecutive spaces in the middle of the object name, only one space is displayed. – The search function can help you quickly select the required database objects.

6. On the **Check Task** page, check the migration task.
   - If any check fails, review the cause and rectify the fault. After the fault is rectified, click **Check Again**.

     For details about how to handle check failures, see **Checking Whether the Source Database Is Connected** in *Data Replication Service User Guide*.
   - If all check items are successful, click **Next**.

Figure 5-9 Task Check



NOTE

You can proceed to the next step only when all check items are successful. If any alarms are generated, view and confirm the alarm details first before proceeding to the next step.

7. On the displayed page, specify **Start Time**, **Send Notification**, **SMN Topic**, **Synchronization Delay Threshold**, and **Stop Abnormal Tasks After** and confirm that the configured information is correct and click **Submit** to submit the task.

Figure 5-10 Task startup settings

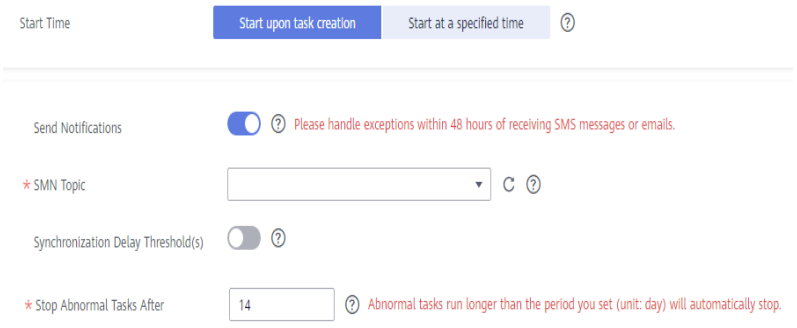


Table 5-8 Task startup settings

Parameter	Description
Start Time	Set **Start Time** to **Start upon task creation** or **Start at a specified time** based on site requirements. The **Start at a specified time** option is recommended. **NOTE** The migration task may affect the performance of the source and destination databases. You are advised to start the task in off-peak hours and reserve two to three days for data verification.
Send Notifications	SMN topic. This parameter is optional. If an exception occurs during migration, the system will send a notification to the specified recipients.
SMN Topic	This parameter is available only after you enable **Send Notification** and create a topic on the SMN console and add a subscriber. For details, see **Simple Message Notification User Guide**.
Synchronization Delay Threshold	During an incremental migration, a synchronization delay indicates a time difference (in seconds) of synchronization between the source and destination database. If the synchronization delay exceeds the threshold you specify, DRS will send alarms to the specified recipients. The value ranges from 0 to 3,600. To avoid repeated alarms caused by the fluctuation of delay, an alarm is sent only after the delay has exceeded the threshold for six minutes. **NOTE** – In the early stages of an incremental migration, there is more delay because more data is waiting to be synchronized. In this situation, no notifications will be sent. – Before setting the delay threshold, enable **Send Notification**. – If the delay threshold is set to 0, no notifications will be sent to the recipient.
Stop Abnormal Tasks After	Number of days after which an abnormal task is automatically stopped. The value must range from 14 to 100. The default value is **14**. **NOTE** Tasks in the abnormal state are still charged. If tasks remain in the abnormal state for a long time, they cannot be resumed. Abnormal tasks run longer than the period you set (unit: day) will automatically stop to avoid unnecessary fees.

8. After the task is submitted, go back to the **Online Migration Management** page to view the task status.

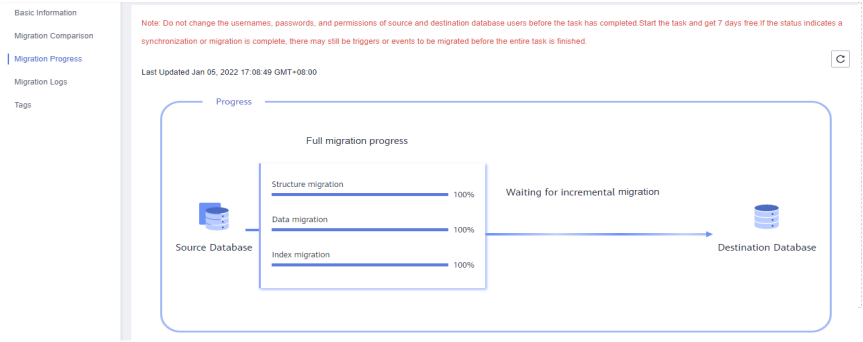
Step 2 Manage the migration task.

The migration task contains two phases: full migration and incremental migration. You can manage them in different phases.

- Full migration

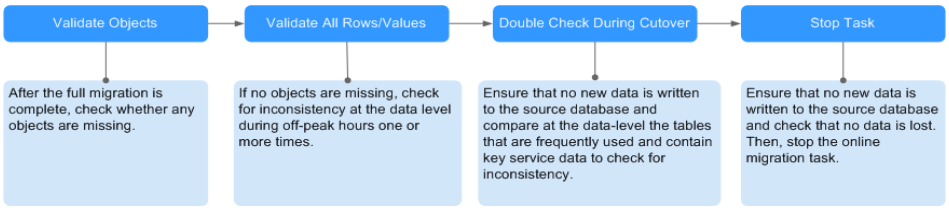
  - Viewing the migration progress: Click the target full migration task, and on the **Migration Progress** tab, you can see the migration progress of the structure, data, indexes, and migration objects. When the progress reaches 100%, the migration is complete.
  - Viewing migration details: In the migration details, you can view the migration progress of a specific object. If the number of objects is the same as that of migrated objects, the migration is complete. You can view the migration progress of each object in detail. Currently, this function is available only to whitelisted users. You can submit a service ticket to apply for this function.
- Incremental Migration Permission
  - Viewing the synchronization delay: After the full migration is complete, an incremental migration starts. On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Progress** to view the synchronization delay of the incremental migration. If the synchronization delay is 0s, the destination database is being synchronized with the source database in real time. You can also view the data consistency on the **Migration Comparison** tab.

    Figure 5-11 Viewing the synchronization delay



  - Viewing the migration results: On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Comparison** and perform a migration comparison in accordance with the comparison process, which should help you determine an appropriate time for migration to minimize service downtime.

    Figure 5-12 Database comparison process



    For details, see **Comparing Migration Items** in *Data Replication Service User Guide*.

Step 3 Cut over services.

You are advised to start the cutover process during off-peak hours. At least one complete data comparison is performed during off-peak hours. To obtain accurate

comparison results, start data comparison at a specified time point during off-peak hours. If it is needed, select **Start at a specified time** for **Comparison Time**. Due to slight time difference and continuous operations on data, inconsistent comparison results may be generated, reducing the reliability and validity of the results.

1. Interrupt services first. If the workload is not heavy, you may not need to interrupt the services.
2. Run the following statement on the source database and check whether any new sessions execute SQL statements within the next 1 to 5 minutes. If there are no new statements executed, the service has been stopped.
   ```
   db.currentOp()
   ```

   NOTE

   The process list queried by the preceding statement includes the connection of the DRS replication instance. If no additional session executes SQL statements, the service has been stopped.
3. On the **Migration Progress** page, view the synchronization delay. When the delay is displayed as 0s and remains stable for a period, then you can perform a data-level comparison between the source and destination databases. For details about the time required, refer to the results of the previous comparison.
   - If there is enough time, compare all objects.
   - If there is not enough time, use the data-level comparison to compare the tables that are frequently used and that contain key business data or inconsistent data.
4. Determine an appropriate time to cut the services over to the destination database. After services are restored and available, the migration is complete.

Step 4 Stop or delete the migration task.

1. Stopping the migration task. After databases and services are migrated to the destination database, to prevent operations on the source database from being synchronized to the destination database to overwrite data, you can stop the migration task. This operation only deletes the replication instance, and the migration task is still displayed in the task list. You can view or delete the task. DRS will not charge for this task after you stop it.
2. Delete the migration task. After the migration task is complete, you can delete it. After the migration task is deleted, it will no longer be displayed in the task list.

----End

6 From ECS-hosted MongoDB to DDS

Scenarios

Data Replication Service (DRS) helps you migrate data from ECS-hosted MongoDB databases to DDS instances on the current cloud. With DRS, you can migrate databases online with zero downtime and your services and databases can remain operational during migration.

This section describes how to use DRS to migrate data from an ECS database to a DDS instance on the current cloud. The following network scenarios are supported:

- Source and destination databases are in the same VPC.
- Source and destination databases are in different VPCs.

Resource Planning

Table 6-1 Resource planning

Category	Subcategory	Planned Value	Description
VPC	VPC name	vpc-dds	Specify a name that is easy to identify.
	Region	AP-Singapore	To achieve lower network latency, select the region nearest to you.
	AZ	AZ1	-
	Subnet	10.0.0.0/24	Select a subnet with sufficient network resources.
	Subnet name	subnet-default	Specify a name that is easy to identify.
ECS	ECS name	ecs-mongodb	Specify a name that is easy to identify.
	Specifications	s6.xlarge.2 4vCPUs|8GiB	The specifications in the left column are selected in this example. In your project, select specifications meeting your workload requirements. For details, see **ECS Specifications**.

Category	Subcategory	Planned Value	Description
	OS	CentOS 7.6 64	-
	System disk	General purpose SSD 40 GiB	-
	Data disk	Ultra-high I/O, 100 GiB	-
	EIP	Buy now	Buy an EIP because the public network is selected for the migration task.
DDS	DDS instance name	dds-test	Specify a name that is easy to identify.
	DB engine	DDS	-
	DB engine version	4.4	-
	AZ type	Single-AZ	-
	AZ	AZ1	-
	Instance specifications	Enhanced II	-
	CPU architecture	x86 8 vCPUs | 32 GB	-
DRS migration task	Task name	DRS-dds	Specify a name that is easy to identify.
	Source DB engine	MongoDB	In this example, the source is a MongoDB database built on an ECS.
	Destination DB engine	DDS	In this example, a DDS instance serves as the destination database.
	Network type	Public network	Public network is used in this example.

Diagram

Figure 6-1 Source and destination databases in the same VPC

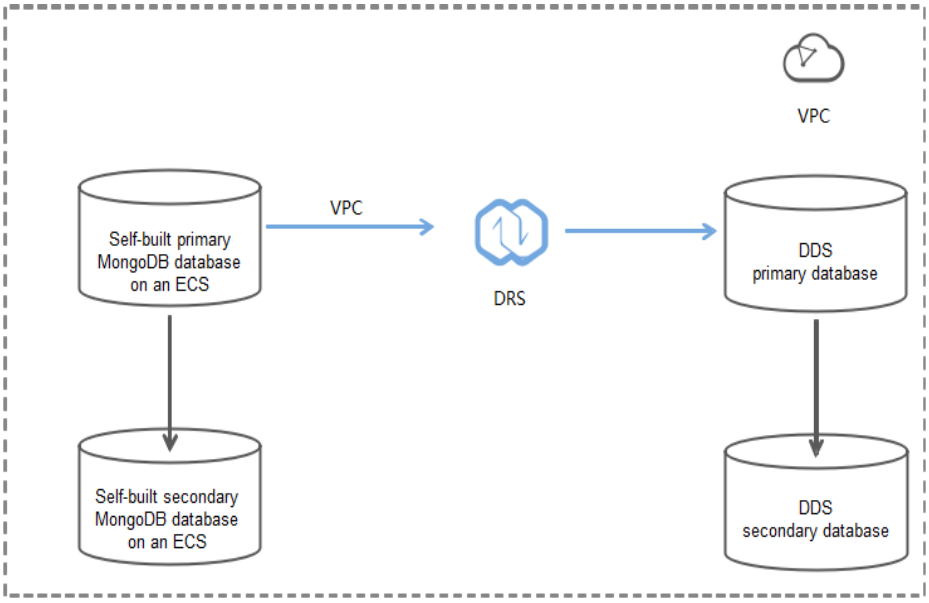
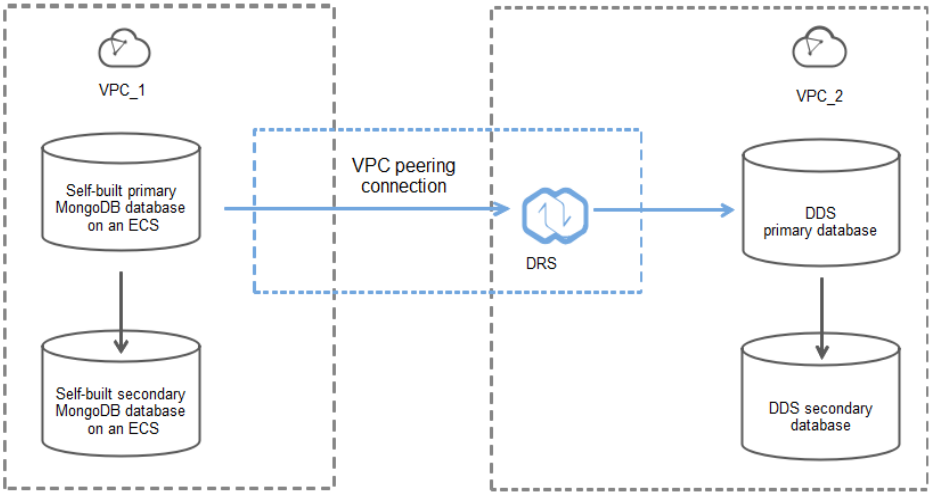
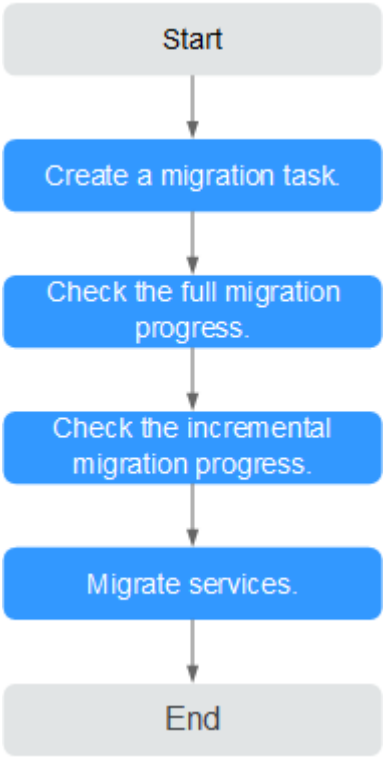


Figure 6-2 Source and destination databases in the same region and different VPCs



Migration Process

Figure 6-3 Flowchart



Migration Suggestions (Important)

- Database migration is closely impacted by a wide range of environmental and operational factors. To ensure the migration goes smoothly, perform a test run before the actual migration to help you detect and resolve any potential issues in advance. Recommendations on how to minimize any potential impacts on your data base are provided in this section.
- It is strongly recommended that you start your migration task during off-peak hours. A less active database is easier to migrate successfully. If the data is fairly static, there is less likely to be any severe performance impacts during the migration.

Notes on Migration (Important)

NOTICE

Before creating a migration task, read the migration notes carefully.

- Supported source and destination databases

Table 6-2 Supported databases

Source DB	Destination DB
• On-premises MongoDB (versions 3.2, 3.4, 3.6, 4.0, 4.2, 4.4, and 5.0) • ECS-hosted MongoDB (versions 3.2, 3.4, 3.6, 4.0, 4.2, 4.4, and 5.0) • Other cloud MongoDB (versions 3.2, 3.4, 3.6, 4.0, 4.2, 4.4, and 5.0) • DDS DB instances (versions 3.2, 3.4, 4.0, 4.2, 4.4, and 5.0) **NOTE** If the source database is a cluster instance of DDS 3.2, only full migration is supported. DDS 5.0 supports replica sets only. 1. If the source database is a DDS DB instance, the source DB engine is DDS. Otherwise, the source DB engine is MongoDB.	• DDS DB instances (versions 3.4, 4.0, 4.2, 4.4, and 5.0) **NOTE** The destination database version must be the same as or later than the source database version. DDS 5.0 supports replica sets only.

You can run the following command to query a database version.

```
db.version()
```

- If you select **Full+Incremental** for **Migration Type**, Oplog of the source database must be enabled. This log occupies certain disk space. Ensure that the source disk has sufficient space. You need to set the retention period of Oplog to at least three days based on the amount of data to be migrated.

For details, see **precautions** on using specific migration tasks in *Data Replication Service Real-Time Migration*.

Preparations

1. Permissions:

   Table 6-3 lists the permissions required for the source and destination database users when migrating data from an ECS-hosted MongoDB database to a DDS instance on the current cloud.

Table 6-3 Required permissions

Database	Full Migration Permission	Full+Incremental Migration Permission
Source	• Replica set: The source database user must have the read permission for the database to be migrated. • Single node: The source database user must have the read permission for the database to be migrated. • Cluster: The source database user must have the read permission for the databases to be migrated and the config database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.	• Replica set: The source database user must have the read permission for the databases to be migrated and the local database. • Single node: The source database user must have the read permission for the databases to be migrated and the local database. • Cluster: The source mongos node user must have the readAnyDatabase permission for the databases to be migrated and the config database. The source shard node user must have the readAnyDatabase permission for the admin database and the read permission for the local database. • To migrate accounts and roles of the source database, the source database user must have the read permission for the **system.users** and **system.roles** system tables of the admin database.
Destination	The destination database user must have the dbAdminAnyDatabase permission for the admin database and the readWrite permission for the destination database. If the destination database is a cluster instance, the migration account must have the read permission for the config database.	

- Source database permissions:

  The source MongoDB database user must have all the required permissions listed in **Table 6-3**. If the permissions are insufficient, create a user that has all of the permissions on the source database.
- Destination database permissions:

  The initial account of the DDS instance has the required permissions.

2. Network settings
   - The source database and destination DDS DB instance must be in the same region.
   - The source database and destination DDS DB instance can be either in the same VPC or different VPCs.
     - If the source and destination databases are in different VPCs, the subnets of the source and destination databases are required to be in different CIDR blocks. You need to create a VPC peering connection between the two VPCs.

       For details, see **VPC Peering Connection Overview** in the *Virtual Private Cloud User Guide*.
     - If the source and destination databases are in the same VPC, the networks are interconnected by default.
3. Security rules
   - In the same VPC, the network is connected by default. You do not need to set a security group.
   - In different VPCs, establish a VPC peering connection between the two VPCs. You do not need to set a security group.
4. Other

   You need to export the parameter information of the MongoDB database first and manually add it to the destination DDS DB instance because the parameter information will not be migrated.

Migration Procedure

Step 1 Create a migration task.

1. Log in to the management console and choose **Databases** > **Data Replication Service** to go to the DRS console.
2. On the **Online Migration Management** page, click **Create Migration Task**.
3. On the **Create Replication Instance** page, configure the task details, recipient, and replication instance and click **Next**.

Figure 6-4 Replication instance information

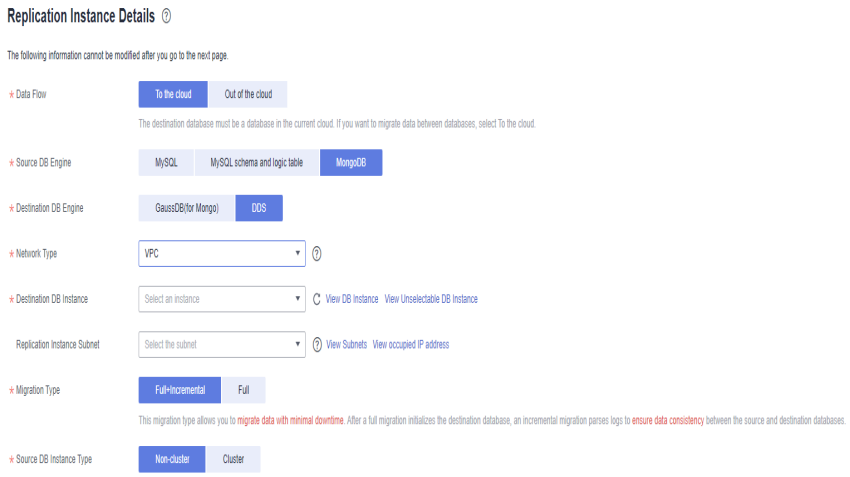


Table 6-4 Task settings

Parameter	Description
Region	The region where the replication instance is deployed. You can change the region. To reduce latency and improve access speed, select the region closest to your workloads.
Project	The project corresponds to the current region and can be changed.
Task Name	The task name consists of 4 to 50 characters, starts with a letter, and can contain only letters (case-insensitive), digits, hyphens (-), and underscores (_).
Description	The description consists of a maximum of 256 characters and cannot contain the following special characters: =<>&'\"

Table 6-5 Replication instance information

Parameter	Description
Data Flow	**To the cloud**
Source DB Engine	Select **MongoDB**.
Destination DB Engine	Select **DDS**.
Network Type	Select **VPC**.
Destination DB Instance	The DDS DB instance you purchased.

Parameter	Description
Migration Type	Select **Full+Incremental** as an example: – **Full**: This migration type is suitable for scenarios where a service interruption is acceptable. All objects and data in non-system databases are migrated to the destination database at one time. The objects include tables, views, and stored procedures. **NOTE** If you perform a full migration, you are advised to stop operations on the source database. Otherwise, data generated in the source database during the migration will not be synchronized to the destination database. – **Full+Incremental**: This migration type allows you to migrate data without interrupting services. After a full migration initializes the destination database, an incremental migration initiates and parses logs to ensure data consistency between the source and destination databases. **NOTE** If you select the **Full+Incremental** migration type, data generated during the full migration will be synchronized to the destination database with zero downtime, ensuring that both the source and destination databases remain accessible.
Source DB Instance Type	If you select **Full+Incremental** for **Migration Type**, set this parameter based on the source database. **Non-cluster** is selected as an example. – If the source database is a cluster instance, set this parameter to **Cluster**. – If the source database is a replica set or a single node instance, set this parameter to **Non-cluster**.

Parameter	Description
Obtain Incremental Data	This parameter is available for configuration if **Source DB Instance Type** is set to **Cluster**. You can determine how to capture data changes during the incremental synchronization. – oplog: For MongoDB 3.2 or later, DRS directly connects to each shard of the source DB instance to extract data. If you select this mode, you must disable the balancer of the source instance. When testing the connection, you need to enter the connection information of each shard node of the source instance. – changeStream: This method is recommended. For MongoDB 4.0 and later, DRS connects to mongos nodes of the source instance to extract data. If you select this method, you must enable the WiredTiger storage engine of the source instance. **NOTE** Only whitelisted users can use **changeStream**. To use this function, submit a service ticket. In the upper right corner of the management console, choose **Service Tickets** > **Create Service Ticket** to submit a service ticket.
Source Shard Quantity	If **Source DB Instance Type** is set to **Cluster** and **Obtain Incremental Data** is set to **oplog**, enter the number of source shard nodes. The default minimum number of source DB instances is 2 and the maximum number is 32. You can set this parameter based on the number of source database shards.

4. On the **Configure Source and Destination Databases** page, wait until the replication instance is created. Then, specify source and destination database information and click **Test Connection** for both the source and destination databases to check whether they have been connected to the replication instance. After the connection tests are successful, select the check box before the agreement and click **Next**.

Figure 6-5 Source and destination database details

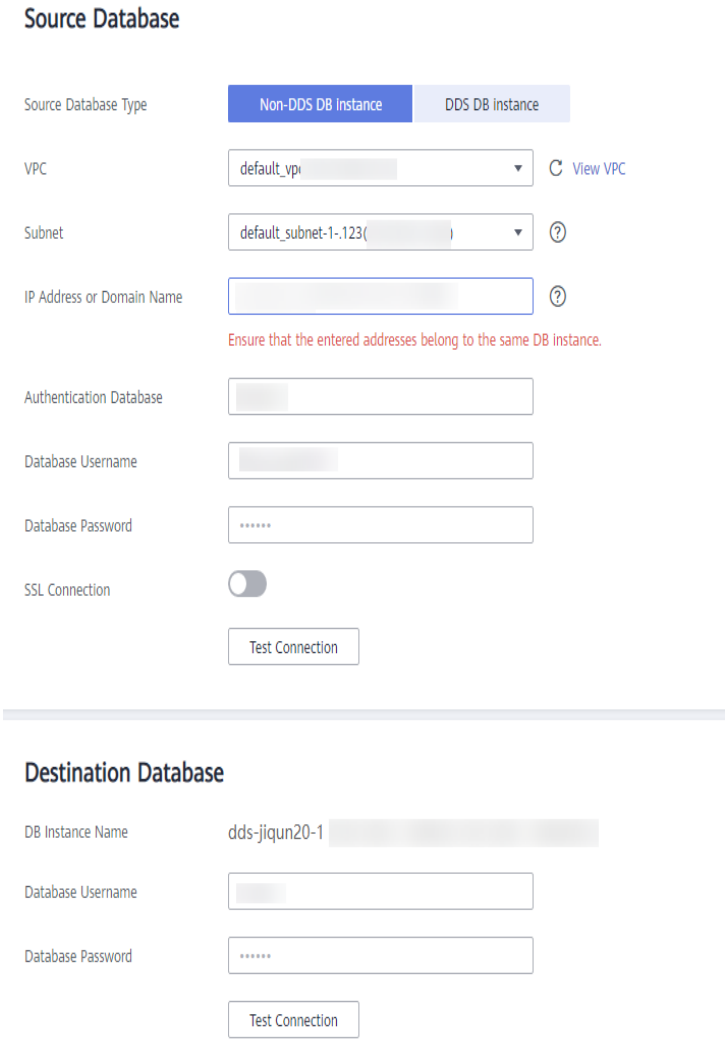


Table 6-6 Source database information

Parameter	Description
Source Database Type	Select **Self-built on ECS**.
VPC	A dedicated virtual network in which the source database is located. It isolates networks for different services. You can select an existing VPC or create a VPC. For details on how to create a VPC, see **Creating a VPC**.
Subnet	A subnet provides dedicated network resources that are logically isolated from other networks, improving network security. The subnet must be in the AZ where the source database resides. You need to enable DHCP for creating the source database subnet. For details on how to create a VPC, see the **Creating a VPC** section in the *Virtual Private Cloud User Guide*.

Parameter	Description
IP Address or Domain Name	The IP address or domain name of the source database.
Port	The port of the source database. Range: 1 - 65535
Database Username	A username for the source database.
Database Password	The password for the database username.
SSL Connection	To improve data security during the migration, you are advised to enable SSL to encrypt migration links and upload a CA certificate.

Table 6-7 Destination database information

Parameter	Description
DB Instance Name	The DDS DB instance you have selected during the migration task creation is displayed by default and cannot be changed.
Database Username	The username for accessing the destination DDS DB instance.
Database Password	The password for the database username.

5. On the **Set Task** page, select migration objects and click **Next**.

Figure 6-6 Migration object

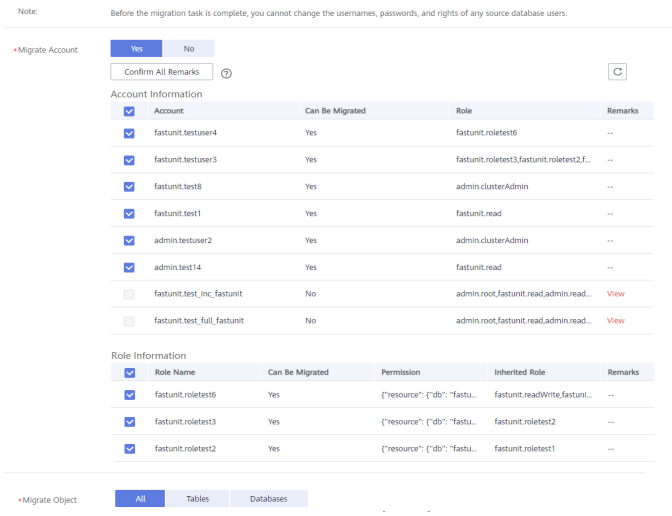


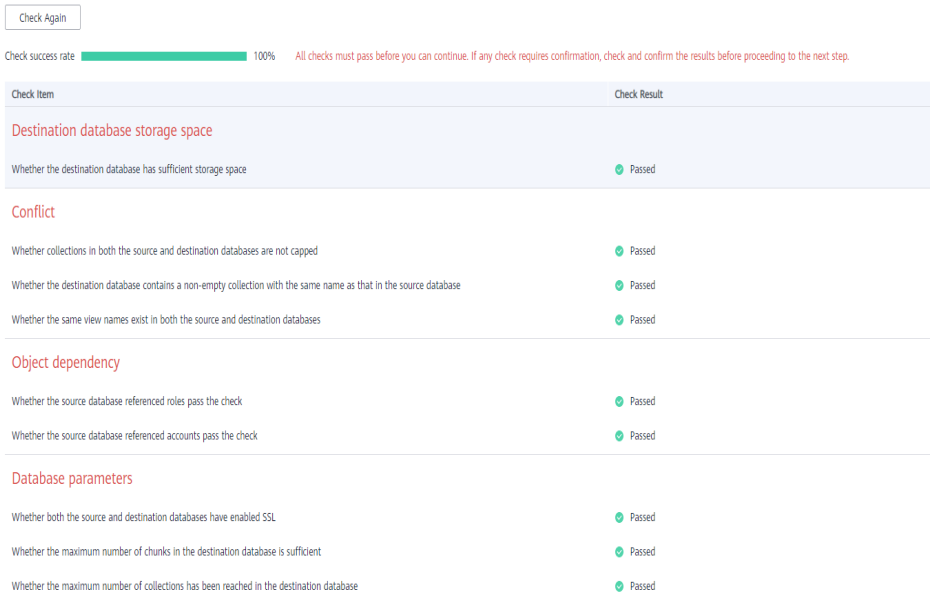
Table 6-8 Migration object

Parameter	Description
Migrate Account	There are accounts that can be migrated completely and accounts that cannot be migrated. You can choose whether to migrate the accounts. Accounts that cannot be migrated or accounts that are not selected will not exist in the destination database. Ensure that your services will not be affected by these accounts. – **Yes** If you choose to migrate accounts, see **Migrating Accounts** in *Data Replication Service User Guide* to migrate database users and roles. – **No** During the migration, accounts and roles are not migrated.
Migrate Object	You can choose to migrate all objects, tables, or databases based on your service requirements. – **All**: All objects in the source database are migrated to the destination database. After the migration, the object names will remain the same as those in the source database and cannot be modified. – **Tables**: The selected table-level objects will be migrated. – **Databases**: The selected database-level objects will be migrated. If the source database is changed, click in the upper right corner before selecting migration objects to ensure that the objects to be selected are from the changed source database. **NOTE** – If you choose not to migrate all of the databases, the migration may fail because the objects, such as stored procedures and views, in the database to be migrated may have dependencies on other objects that are not migrated. To ensure a successful migration, you are advised to migrate all of the databases. – When you select an object, the spaces before and after the object name are not displayed. If there are two or more consecutive spaces in the middle of the object name, only one space is displayed. – The search function can help you quickly select the required database objects.

6. On the **Check Task** page, check the migration task.
   – If any check fails, review the cause and rectify the fault. After the fault is rectified, click **Check Again**.

     For details about how to handle check failures, see **Checking Whether the Source Database Is Connected** in *Data Replication Service User Guide*.
   – If all check items are successful, click **Next**.

Figure 6-7 Task Check



NOTE

You can proceed to the next step only when all check items are successful. If any alarms are generated, view and confirm the alarm details first before proceeding to the next step.

7. On the displayed page, specify **Start Time**, **Send Notification**, **SMN Topic**, **Synchronization Delay Threshold**, and **Stop Abnormal Tasks After** and confirm that the configured information is correct and click **Submit** to submit the task.

Figure 6-8 Task startup settings

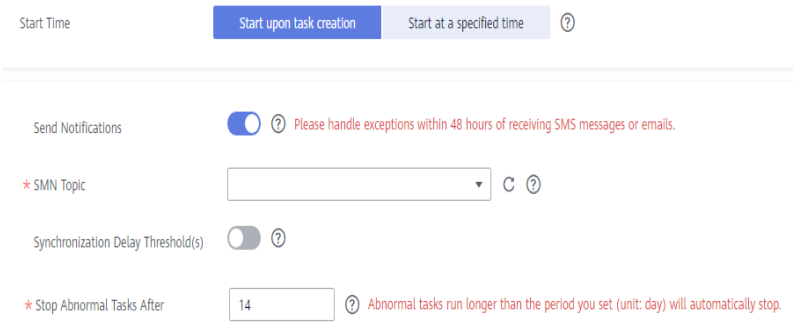


Table 6-9 Task startup settings

Parameter	Description
Start Time	Set **Start Time** to **Start upon task creation** or **Start at a specified time** based on site requirements. The **Start at a specified time** option is recommended. **NOTE** The migration task may affect the performance of the source and destination databases. You are advised to start the task in off-peak hours and reserve two to three days for data verification.
Send Notifications	SMN topic. This parameter is optional. If an exception occurs during migration, the system will send a notification to the specified recipients.
SMN Topic	This parameter is available only after you enable **Send Notification** and create a topic on the SMN console and add a subscriber. For details, see **Simple Message Notification User Guide**.
Synchronization Delay Threshold	During an incremental migration, a synchronization delay indicates a time difference (in seconds) of synchronization between the source and destination database. If the synchronization delay exceeds the threshold you specify, DRS will send alarms to the specified recipients. The value ranges from 0 to 3,600. To avoid repeated alarms caused by the fluctuation of delay, an alarm is sent only after the delay has exceeded the threshold for six minutes. **NOTE** – In the early stages of an incremental migration, there is more delay because more data is waiting to be synchronized. In this situation, no notifications will be sent. – Before setting the delay threshold, enable **Send Notification**. – If the delay threshold is set to 0, no notifications will be sent to the recipient.
Stop Abnormal Tasks After	Number of days after which an abnormal task is automatically stopped. The value must range from 14 to 100. The default value is **14**. **NOTE** Tasks in the abnormal state are still charged. If tasks remain in the abnormal state for a long time, they cannot be resumed. Abnormal tasks run longer than the period you set (unit: day) will automatically stop to avoid unnecessary fees.

8. After the task is submitted, go back to the **Online Migration Management** page to view the task status.

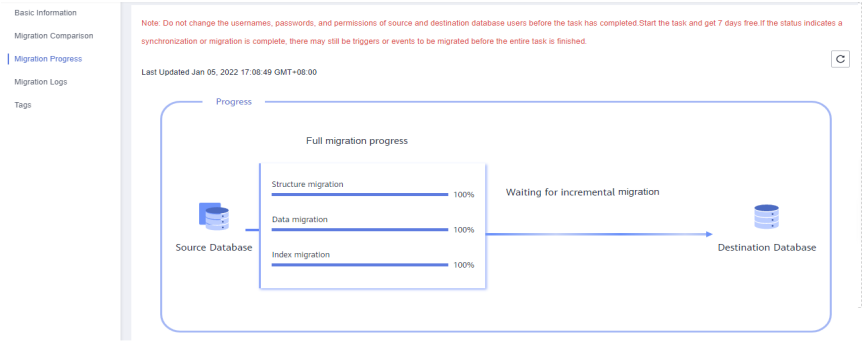
Step 2 Manage the migration task.

The migration task contains two phases: full migration and incremental migration. You can manage them in different phases.

- Full migration

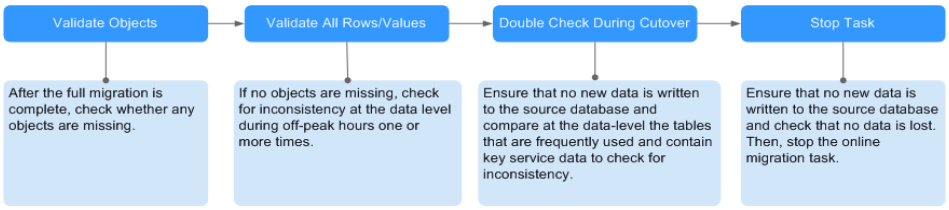
  - Viewing the migration progress: Click the target full migration task, and on the **Migration Progress** tab, you can see the migration progress of the structure, data, indexes, and migration objects. When the progress reaches 100%, the migration is complete.
  - Viewing migration details: In the migration details, you can view the migration progress of a specific object. If the number of objects is the same as that of migrated objects, the migration is complete. You can view the migration progress of each object in detail. Currently, this function is available only to whitelisted users. You can submit a service ticket to apply for this function.
- Incremental Migration Permission
  - Viewing the synchronization delay: After the full migration is complete, an incremental migration starts. On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Progress** to view the synchronization delay of the incremental migration. If the synchronization delay is 0s, the destination database is being synchronized with the source database in real time. You can also view the data consistency on the **Migration Comparison** tab.

    Figure 6-9 Viewing the synchronization delay



  - Viewing the migration results: On the **Online Migration Management** page, click the target migration task. On the displayed page, click **Migration Comparison** and perform a migration comparison in accordance with the comparison process, which should help you determine an appropriate time for migration to minimize service downtime.

    Figure 6-10 Database comparison process



    For details, see **Comparing Migration Items** in *Data Replication Service User Guide*.

Step 3 Cut over services.

You are advised to start the cutover process during off-peak hours. At least one complete data comparison is performed during off-peak hours. To obtain accurate

comparison results, start data comparison at a specified time point during off-peak hours. If it is needed, select **Start at a specified time** for **Comparison Time**. Due to slight time difference and continuous operations on data, inconsistent comparison results may be generated, reducing the reliability and validity of the results.

1. Interrupt services first. If the workload is not heavy, you may not need to interrupt the services.
2. Run the following statement on the source database and check whether any new sessions execute SQL statements within the next 1 to 5 minutes. If there are no new statements executed, the service has been stopped.
   ```
   db.currentOp()
   ```

   NOTE

   The process list queried by the preceding statement includes the connection of the DRS replication instance. If no additional session executes SQL statements, the service has been stopped.
3. On the **Migration Progress** page, view the synchronization delay. When the delay is displayed as 0s and remains stable for a period, then you can perform a data-level comparison between the source and destination databases. For details about the time required, refer to the results of the previous comparison.
   - If there is enough time, compare all objects.
   - If there is not enough time, use the data-level comparison to compare the tables that are frequently used and that contain key business data or inconsistent data.
4. Determine an appropriate time to cut the services over to the destination database. After services are restored and available, the migration is complete.

Step 4 Stop or delete the migration task.

1. Stopping the migration task. After databases and services are migrated to the destination database, to prevent operations on the source database from being synchronized to the destination database to overwrite data, you can stop the migration task. This operation only deletes the replication instance, and the migration task is still displayed in the task list. You can view or delete the task. DRS will not charge for this task after you stop it.
2. Delete the migration task. After the migration task is complete, you can delete it. After the migration task is deleted, it will no longer be displayed in the task list.

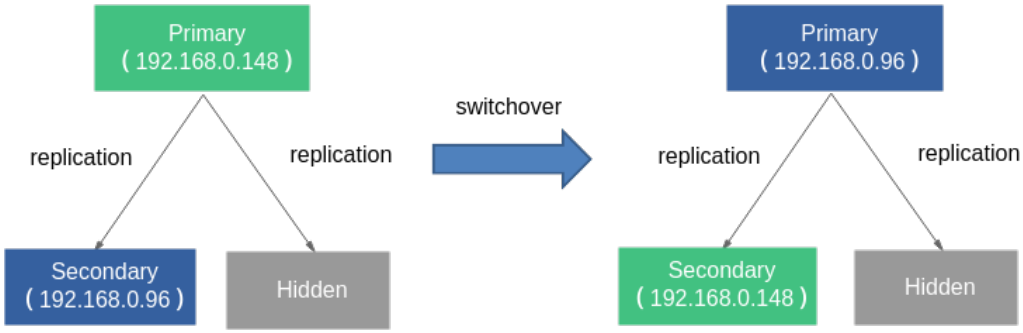
----End

7 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 7-1 Primary/Secondary switchover



Connecting to a Replica Set Instance (HA)

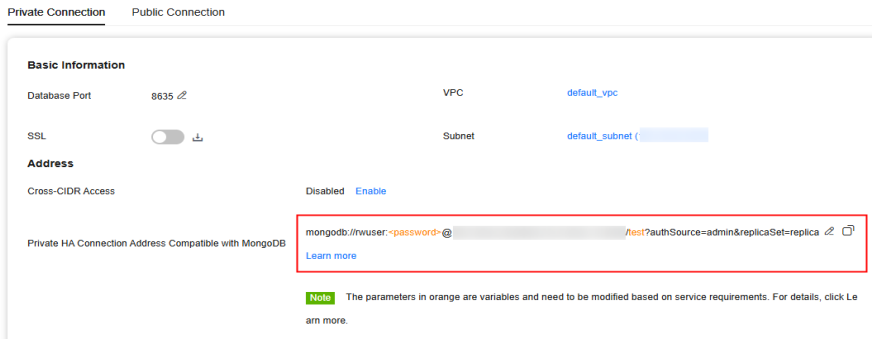
A DDS replica set consists of the primary, secondary, and hidden nodes. The hidden node is invisible to users. Read/Write splitting and HA can be realized only when you connect to the IP addresses and ports of the primary and secondary nodes of the replica set at the same time (in HA mode).

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.

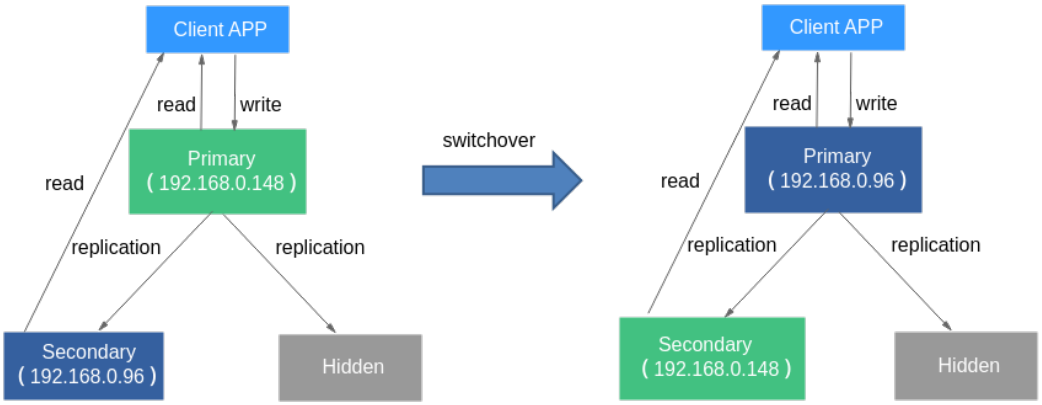
Figure 7-2 Obtaining the private HA connection address



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 7-3 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 7-1 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

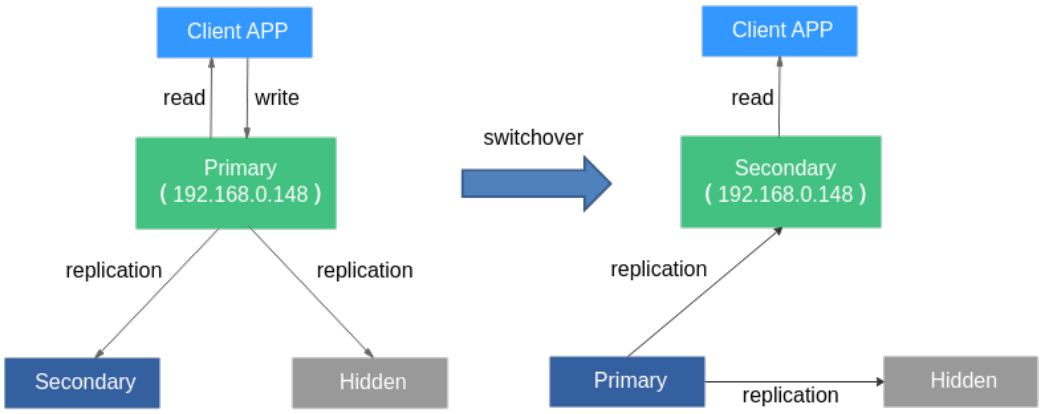
(Not Recommended) Connecting to a Replica Set Instance

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 7-4 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.

8 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.

CAUTION

If sharding is enabled for a non-empty collection, the collection is locked and services are blocked. The time required for sharding increases with the amount of data in the collection. To minimize the impact, before sharding, ensure that the target table's workload is running during off-peak hours.

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> }})
  ```

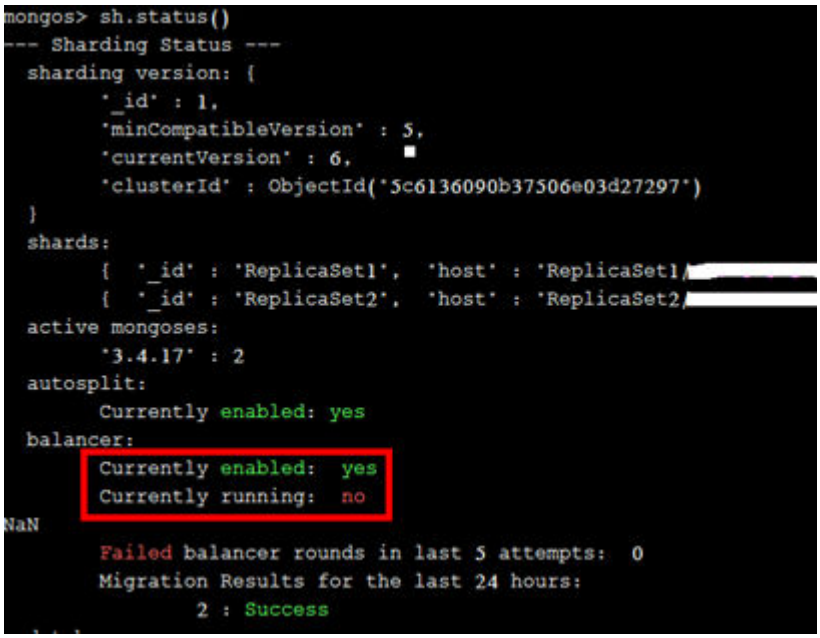
Table 8-1 Parameter description

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 8-2** to determine whether the selected shard keys meet your service requirements:

Table 8-2 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.

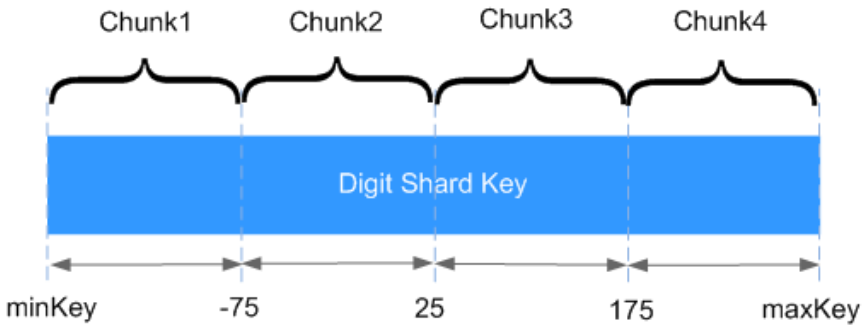
There are two data distribution policies: ranged sharding and hashed sharding. For details, see **Step 4**.

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 8-1 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.

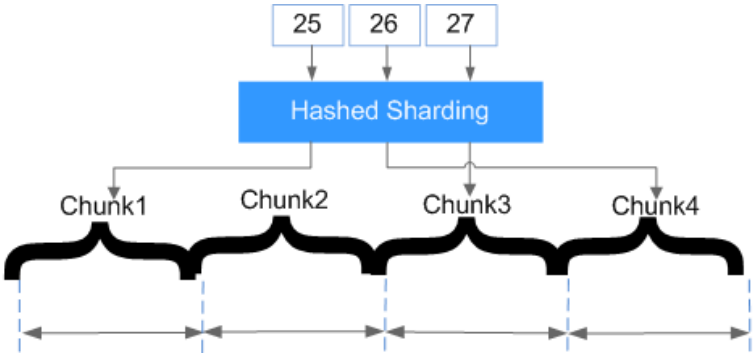
Ranged sharding can easily meet the requirements of query in a certain range. For example, if you need to query documents whose shard key is in range [-60,20], dds mongos only needs to forward the request to chunk 2.

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 8-2 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.

9 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

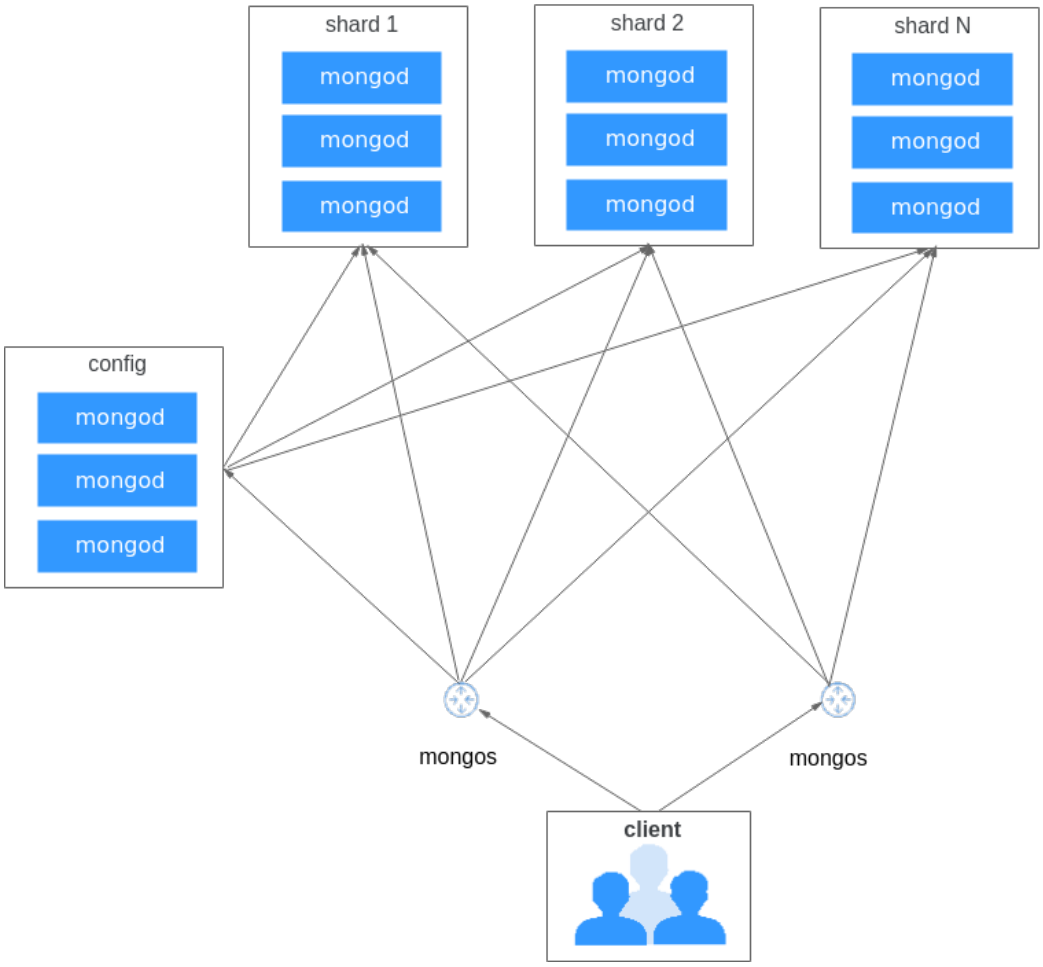
  During aggregation, $match must be placed before $group to reduce the number of documents to be processed by the $group operator.

10 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 10-1 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.

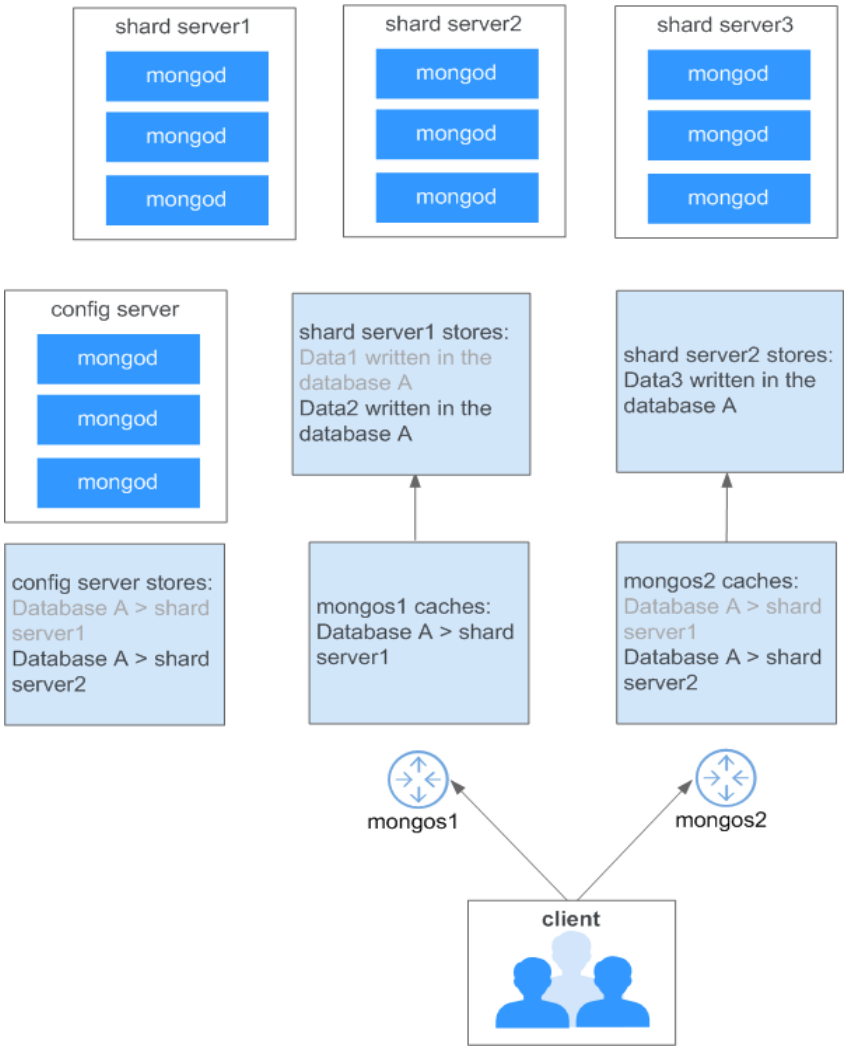
The data to be updated is in the unit of DB.

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 10-2 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.

11 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 collections
     system.profile
     test
     ```
   - 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 collections
     test
     ```
4. Check the slow query logs in the database.

   db.system.profile.find().pretty()
5. Analyze slow query logs to find the cause of the high CPU usage.

   The following is an example of a slow query log. The log shows 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.

12 How Do I Troubleshoot High Memory Usage of DDS DB Instances?

During DDS usage, if your memory usage is too high or close to 100%, service requests will be responded slowly and OOM risks will increase, affecting stable services.

This section describes how to troubleshoot the high memory usage of DDS DB instances by checking the number of database connections, analyzing slow requests, and checking cursors. After the analysis and optimization, query performance will be improved, indexes will be used more efficiently for all requests, and cursors will be used in a standard manner. If the memory usage increases due to business growth, you are advised to **upgrade the specifications** in a timely manner.

Checking the Number of Connections

1. Check the **percentage of connections**. The total number of connections cannot exceed 80% of the maximum number of connections supported by the current instance. If there are too many connections, the memory and multi-thread context overhead increases, affecting the delay in request processing.
2. Configure a connection pool. It is recommended that the maximum number of connections in a connection pool be 200. For details, see **How Do I Query and Limit the Number of Connections?**

Analyzing Slow Query Logs

In addition to reducing the number of connections, pay attention to the memory overhead of a single request to avoid full table scan and memory sorting in query statements.

1. Query **slow query logs** generated for the current DB instance.
2. Analyze slow query logs to locate the cause of memory usage increase. The following is an example of a slow request log. The log shows that a full table scan is performed for the request, 1,561,632 documents are scanned, and no index is used for query.

```
{
        "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 memory resources are occupied. If you find **COLLSCAN** in the slow query log, memory 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 memory usage.
- **IXSCAN** and **keysExamined** scan indexes.
- **SORT** and **hasSortStage** may involve sorting a large amount of data.

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

If there is **SORT**, you can use indexing to improve sorting speed.

NOTE

An excessive number of indexes can affect the write and update performance. You are advised to create indexes based on the ESR principle to improve query efficiency.

- "Equality" refers to an exact match on a single value. Place fields that require exact matches first in an index.
- "Sort" determines the order for results. A sort condition follows equality matches.
- "Range" filters scan fields. Place a range filter after a sort condition.

Checking Cursors

If cursors are used improperly, the memory usage may increase and the cursors may not be released for a long time. When a cursor is used on the client, release it in a timely manner (For details, see **official description**).

1. Check whether a cursor is set to **noTimeout**. By default, the database automatically releases a cursor 10 minutes later. The following is an example of cursor timeout code provided by the Java driver:
   ```
   MongoCursor<Document> cursor = collection.find(query)
                   .maxTime(10, TimeUnit.MINUTES)
                   .iterator();
   ```
2. Check whether the client actively releases a cursor after the cursor is used. The following is an example of releasing a cursor in the Java driver:
   ```
   cursor.close()
   ```
3. If cursors are set to **noTimeout** and they cannot be released on the client, **restart the instance node** with high memory usage to release these cursors. You are advised to optimize the service code to avoid setting **noTimeout** cursors and release them after using them.

13 What Can I Do If the Number of Connections of an Instance Reaches Its Maximum?

The number of connections indicates the number of applications that can be simultaneously connected to the database. The number of connections is irrelevant to the maximum number of users allowed by your applications or websites.

- For a cluster instance, the number of connections is the number of connections between the client and the mongos.
- For a replica set instance, the number of connections is the number of connections between the client and the primary and secondary nodes.

Symptom

When the number of connections to a DDS DB instance reaches the maximum, new connection requests cannot be responded. As a result, the connection to the DB instance fails.

- If the following information is displayed when you use Mongo Shell to connect to an instance, no more connections can be established.

```
[root@623-[redacted] ~]# mongo "mongodb://rwuser:[redacted]@192.168.0.180:8635,192.168.0.50:8635/test?authSource=admin&replicaSet=replica"
MongoDB shell version v3.4.17
connecting to: mongodb://rwuser:623_Huawei@192.168.0.180:8635,192.168.0.50:8635/test?authSource=admin&replicaSet=replica
2019-10-16T17:43:45.203+0800 I NETWORK  [thread1] Starting new replica set monitor for replica/192.168.0.180:8635,192.168.0.50:8635
2019-10-16T17:43:45.205+0800 W NETWORK  [ReplicaSetMonitor-TaskExecutor-0] No primary detected for set replica
2019-10-16T17:43:45.205+0800 I NETWORK  [ReplicaSetMonitor-TaskExecutor-0] All nodes for set replica are down. This has happened for 1 check
s in a row.
2019-10-16T17:43:45.708+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:45.708+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 2 checks in a row.
2019-10-16T17:43:46.210+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:46.210+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 3 checks in a row.
2019-10-16T17:43:46.712+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:46.712+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 4 checks in a row.
2019-10-16T17:43:47.215+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:47.215+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 5 checks in a row.
2019-10-16T17:43:47.717+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:47.717+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 6 checks in a row.
2019-10-16T17:43:48.218+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:48.218+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 7 checks in a row.
2019-10-16T17:43:48.721+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:48.721+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 8 checks in a row.
2019-10-16T17:43:49.222+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:49.222+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 9 checks in a row.
2019-10-16T17:43:49.724+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:49.724+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 10 checks in a row.
2019-10-16T17:43:50.226+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:50.226+0800 I NETWORK  [thread1] All nodes for set replica are down. This has happened for 11 checks in a row.
2019-10-16T17:43:50.727+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:51.230+0800 W NETWORK  [thread1] No primary detected for set replica
2019-10-16T17:43:51.731+0800 W NETWORK  [thread1] No primary detected for set replica
```

- If the following information is displayed when you use Python to connect to an instance, the number of connections reaches its maximum.

pymongo.errors.ServerSelectionTimeoutError: connection closed, connection closed

- You can also view the **instance monitoring metrics** to check whether the number of instance connections is used up.

Solution

Handle the problem based on its cause of burst traffic or long-term workloads.

- If the number of connections reaches the upper limit due to burst traffic, **restart the instance or node** to release the current connections. Also, check whether there is a large number of retry requests on the client. If the number of retry requests is used up, modify the client parameters to prevent the number of connections from being stacked. You need to modify the retry logic and increase the timeout retry duration.

  CAUTION

  Restarting an instance will restart the nodes of the instance in turn. Each node will be intermittently disconnected for about 30 seconds. If the number of collections exceeds 10,000, the intermittent disconnection duration will be prolonged. Before restarting an instance, arrange workloads and ensure that the application supports automatic reconnection.

- If the number of connections is used up due to long-term workloads, **increase the maximum number of connections** by changing the value of **net.maxIncomingConnections**. The change takes effect immediately. Ensure that a changed value is within 20% of the original value each time. After the value change, **observe the load changes**. If the load is too high after the number of connections is increased, the instance load has reached the bottleneck. In this case, **upgrade the instance specifications** in a timely manner.

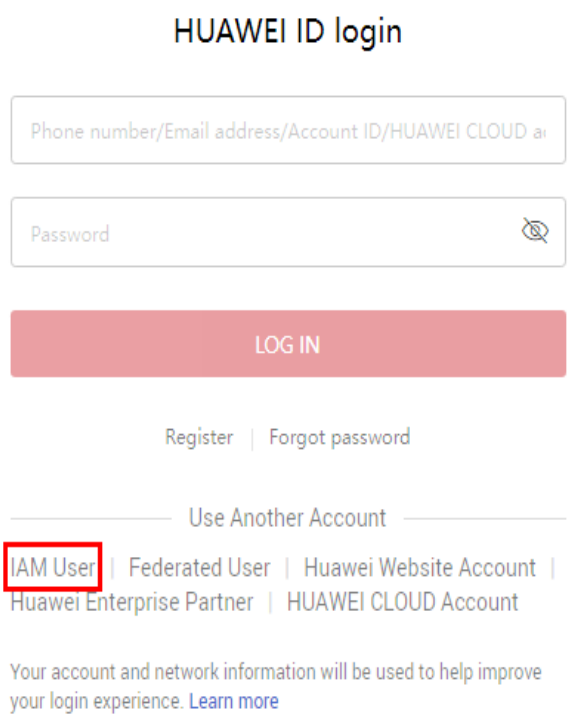
14 Creating a User and Granting the Read-Only Permission to the User

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:

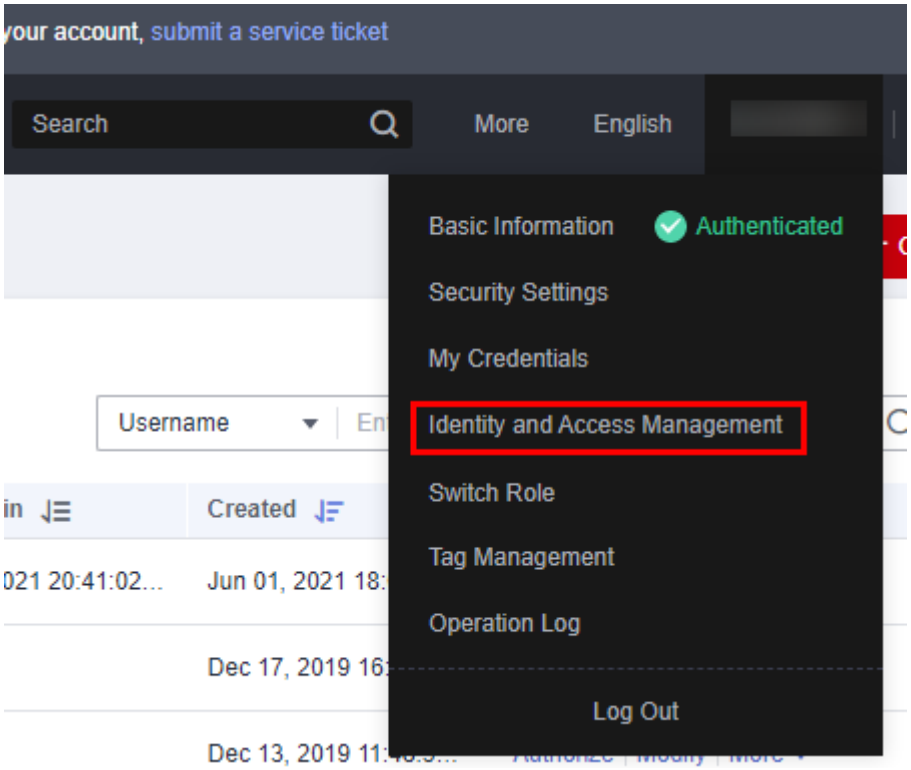
Step 1 Log in to Huawei Cloud using your Huawei account. Select **HUAWEI ID login**.

Figure 14-1 HUAWEI ID Login



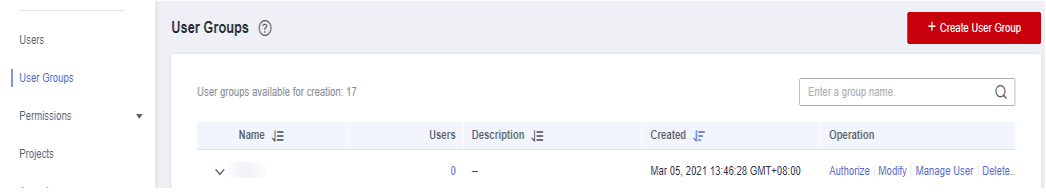
Step 2 On the management console, click the username in the upper right corner and then choose **Identity and Access Management**.

Figure 14-2 Choosing IAM



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

Figure 14-3 User group



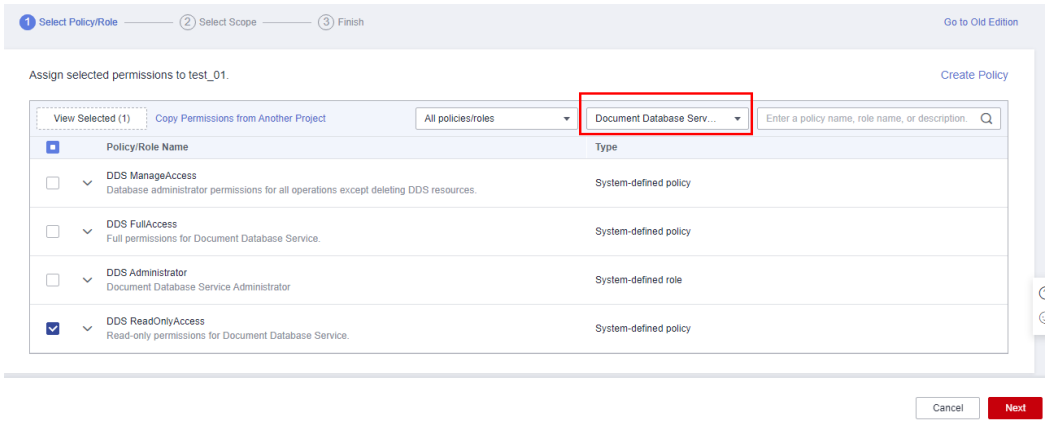
Step 4 Enter a user group name (for example, **test_01**), set the password, and click **OK**.

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**.

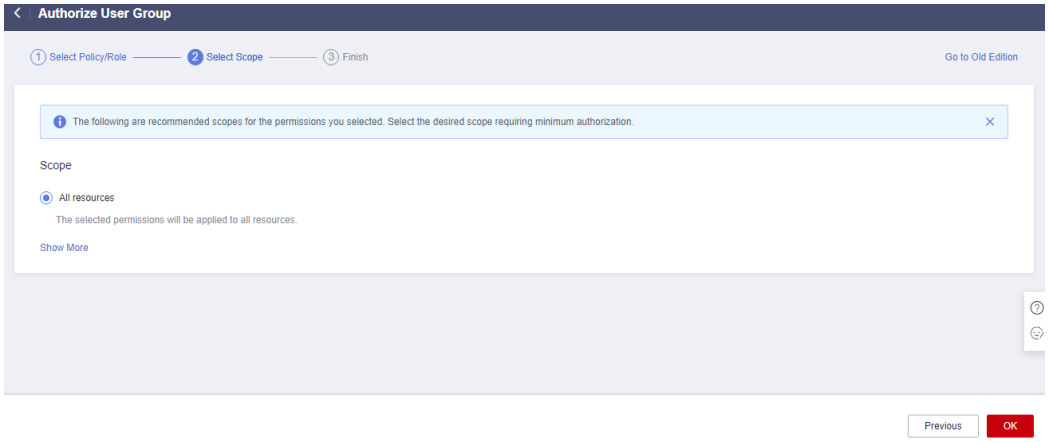
Figure 14-4 Authorization



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.

Figure 14-5 Specifying the scope



----End

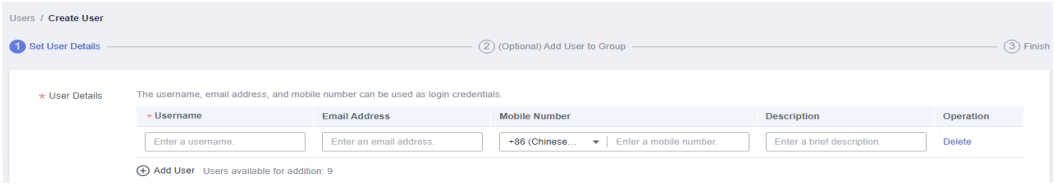
Step 2: Create an IAM User

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.

Figure 14-6 Creating a user



- **Username**: Used for logging in to Huawei Cloud. 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**.

Figure 14-7 Configuring user details

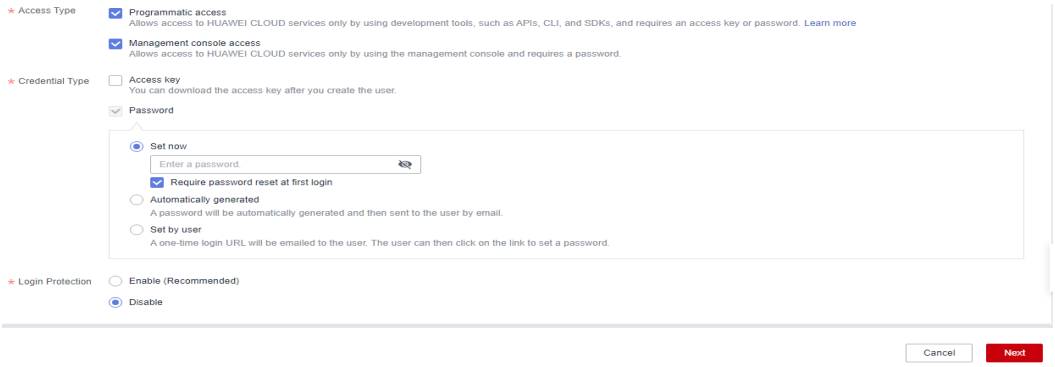


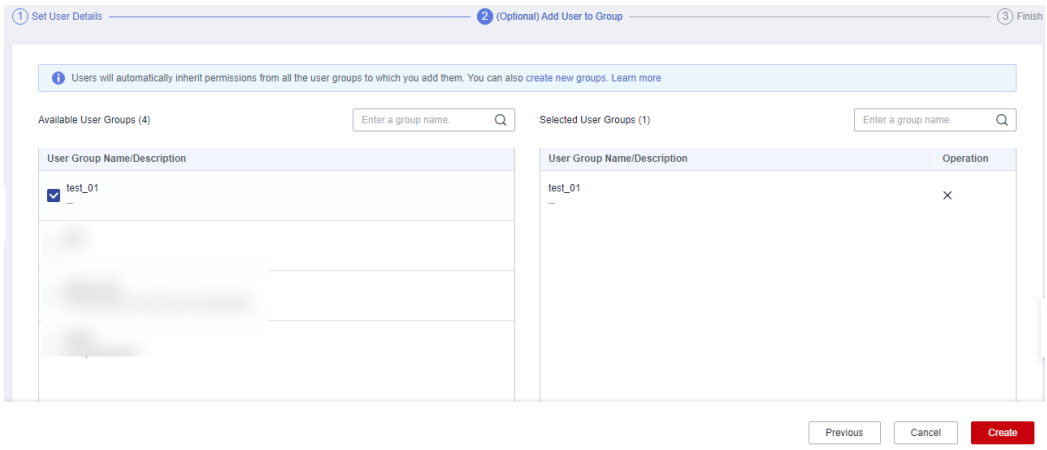
Table 14-1 Configuration items

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.
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.

Parameter	Description
Login Protection	To ensure account security, you are advised to select **Enable**. To enable or disable login protection for an IAM user after creation, see **Login Protection**.

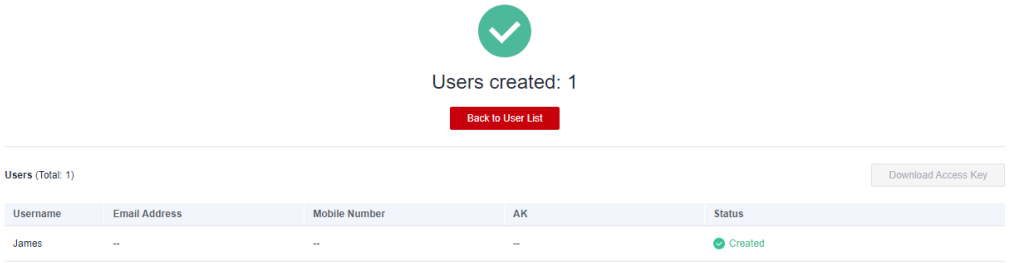
Step 4 Add the users to user group created in **Step 4** and click **Create User**.

Figure 14-8 Creating a 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.

Figure 14-9 Viewing the results



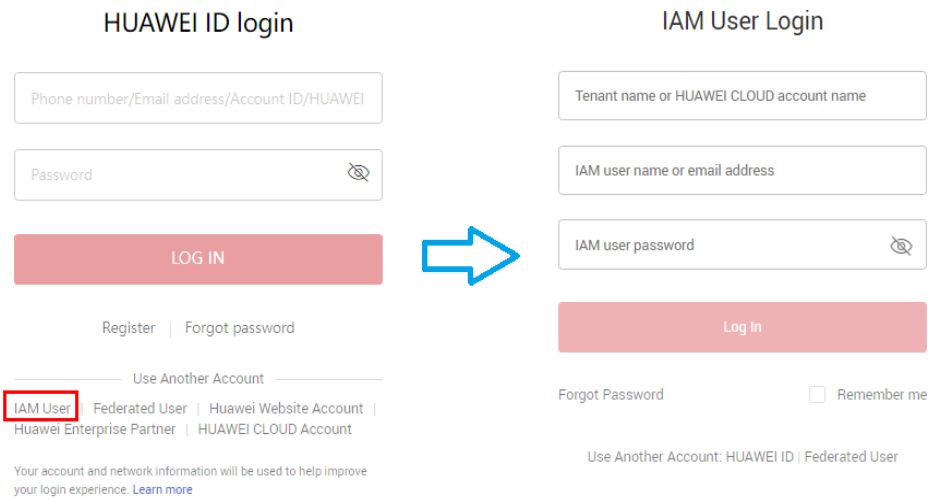
----End

Step 3: Log In and Verify Permissions

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

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

Figure 14-10 IAM user login



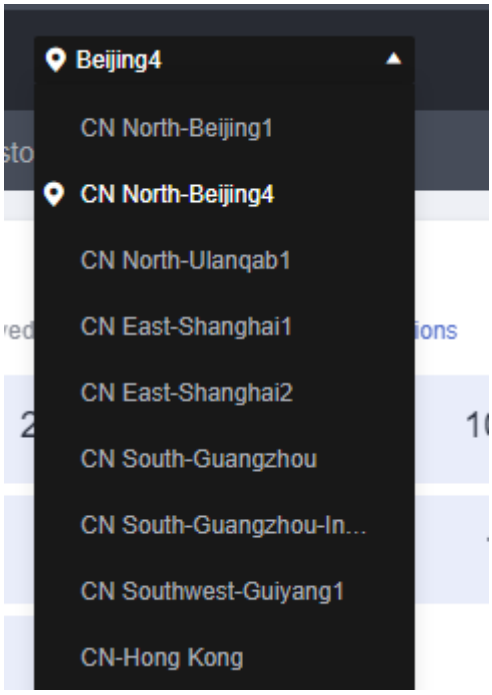
Step 2 Enter the account name, username, and password, and click **Log In**.

- The account name is the name of the Huawei 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.

Figure 14-11 Region



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

15 Proper Use of Data Definition Languages (DDL) Statements

A data definition language (DDL) is a SQL used to create, modify, and delete the structure of databases and collections.

There are the following common DDLs in DDS:

- **createCollection**: used to create a collection in a specified database.
- **drop**: used to delete a specified collection.
- **createIndex**: used to create one or more indexes in a collection to improve query efficiency.
- **dropIndex**: used to delete a specified index from a collection.
- **shardCollection**: Run the **sh.shardCollection** command to modify the sharding rule of a collection so that the collection is distributed in different shards.
- **collection.stats**: used to view metadata of a specified collection, such as the number of documents and storage size.
- **db.stats**: used to view metadata of a specified database, such as the number of collections and storage.

Precautions for Using DDLs

- Creating indexes, deleting indexes, and deleting collections consume a large amount of I/O or compute resources. Perform these operations during off-peak hours to prevent affecting services.
- Do not execute multiple DDLs at the same time. Otherwise, the execution may fail due to blocking. For example, do not create an index and delete a collection at the same time.
- Create necessary indexes only to prevent storage waste caused by redundant indexes. For example, do not create an index based on the prefix field of a composite index.
- When creating an index, you need to create a background index using **db.<collection_name>.createIndex({ <field_name>: <index_type> }, { background: true })**. Note that creating a background index does not block other services, but still consumes a large amount of I/O resources.

- Before deleting an index, perform a performance test to ensure that the index to be deleted does not affect the query performance.
- Do not create too many collections. If there are too many collections, a large amount of metadata is generated, extra resources are consumed, and it is too difficult to perform maintenance. For example, do not name a collection by date or create a new collection every day. Instead, store the date as a field in a given collection.
- Before deleting a collection, confirm the collection name with caution because data cannot be directly restored after the collection is deleted. You are advised to back up important data first.
- If you want to delete a collection, do not use the **remove** or **delete** command without filtering conditions. Instead, use **db.<collection_name>.drop()** to delete a collection. If a query condition is specified for the **remove** or **delete** command, the corresponding index must be created.
- In a cluster instance, if a collection has a large storage space, you need to shard the collection. For details, see **Sharding**.
- You can use **db.stats()** and **db.<collection_name>.stats()** to view the metadata (such as the number of documents and storage size) of databases and collections. The information is important for performance optimization and capacity planning.

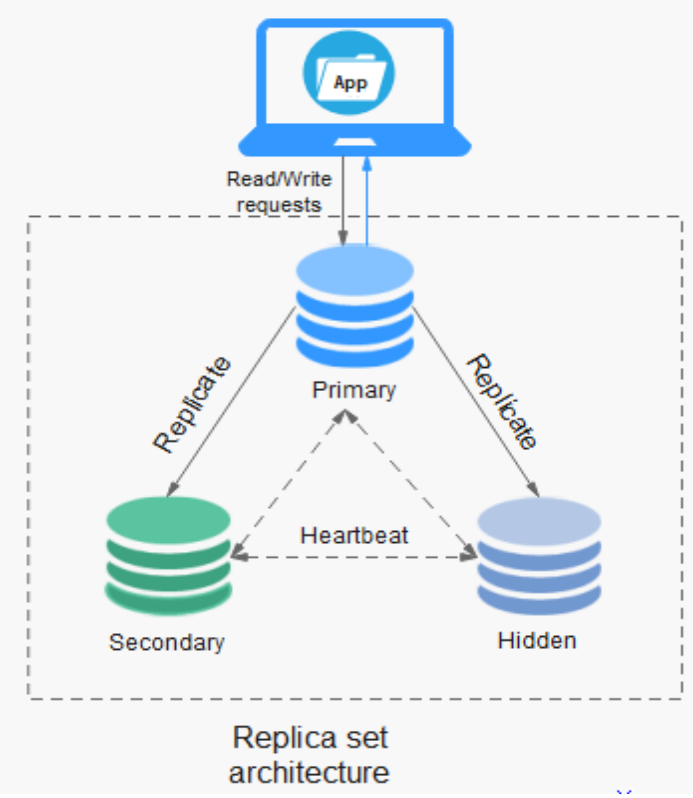
16 How Is a DDS Node Going to Be Disconnected and What Can I Do?

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 16-1 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 Does Replication Delay Occur?

If the write speed of the primary node exceeds the oplog retrieval and replay speed of the secondary nodes, the primary/secondary replication delay occurs.

When Does a Secondary Node Fall Out of Sync?

Because oplog storage capacity is limited, the earliest oplog entries are overwritten once the upper limit is reached. The secondary node reads the oplog and records the timestamp of the last retrieved oplog entry each time. If the replication delay increases to a certain point where the last replayed oplog position has already been overwritten on the primary, the secondary node can no longer read subsequent oplog entries. The secondary node has become out of sync.

How Do I Prevent Secondary Nodes from Falling Out of Sync?

- 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 **Modifying DDS DB Instance Parameters**.
- 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.

17 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));
    });
})();
```


18 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.

  For details, see **Sharding**.

- 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.

19 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. For details, see **Changing an Instance Class**.
  - [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.

20 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.

21 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**.

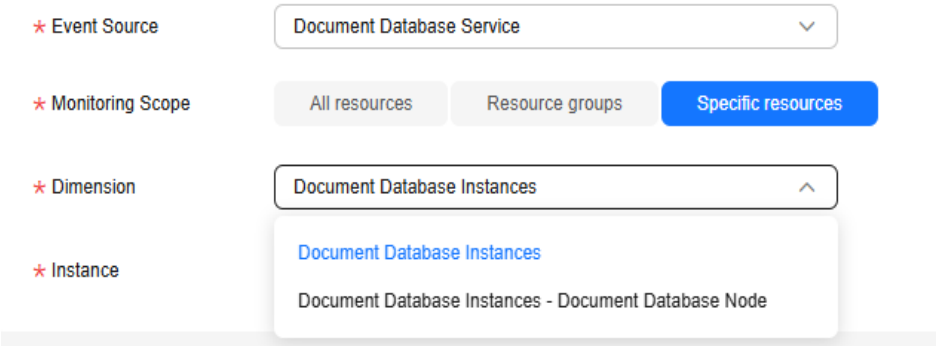
Step 4 On the displayed **Alarm Rules** page, click **Create Alarm Rule**.

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 21-1 Configuring a monitoring dimension



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

----End

Table 21-1 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.

22 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.

Regularly analyzing and optimizing index performance is essential. The database management system usually provides tools and commands to evaluate index usage and identify performance issues. It is 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 Index Creation

- Do not use aggregate functions 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 makes queries faster.
- Use high-selectivity filter criteria to narrow down the query scope and ensure that indexes exist on high-selectivity fields. You are advised 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, 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, 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.

Impact of Index Creation

After an index creation request is submitted, the system scans the original data documents 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 writes 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.

23 Oplog Size Planning and Optimization

Scenarios

This section provides guidance on properly configuring DDS oplog parameters to mitigate risks such as primary/secondary replication exceptions or limitations in point-in-time recovery. Correct oplog configuration helps ensure system stability and reliable data restoration.

Oplog Basics

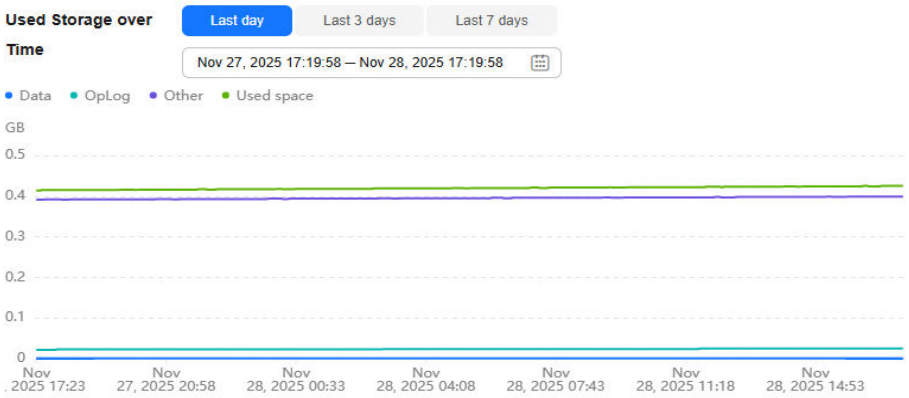
- The operations log (oplog) is the basis of replica set replication. In a cluster instance, shards and config are essentially replica sets and also rely on their own oplog to record and propagate data changes. The oplog is a special capped collection, functioning as a ring buffer, that continuously records all operations that modify data in your database. When the oplog's space is full, the earliest entries are overwritten by new ones. The primary node executes write operations and then records them in its own oplog. Secondary nodes asynchronously pull these oplog entries and replay them locally to maintain data consistency. If a write operation does not actually modify any data or fails, it typically does not generate an oplog entry.
- In a newly created instance, the oplog occupies 10% of the storage space by default. When you scale up the storage, if the original oplog size is less than 100 GB, it is resized (the new size is the smaller of either the storage space multiplied by **oplogSizePercent** or 100 GB). If the original oplog size is greater than or equal to 100 GB, the size remains unchanged.
- Oplog operations are stored in the **local.oplog.rs** collection and they are idempotent. That is, an oplog operation produces the same result regardless of whether it is applied once or multiple times.
- The oplog grows at a rate roughly proportional to the system's speed in processing write requests. If a single operation affects multiple documents, multiple corresponding oplog entries are generated.
- The oplog window refers to the time interval between the oldest and newest entries in the oplog. Primary/secondary replication depends on this time window. A secondary node can synchronize properly only if the oplog window of its source contains the required oplog entries.

Checking the Oplog Size

Use either of the following methods to check the oplog size.

1. Use **storage analysis** to view the storage space distribution and the oplog size.

   Figure 23-1 Viewing storage space distribution

2. Connect to the database to check the oplog size.
   - Connect to the database. For details, see **Connecting to a DDS Instance**.
   - Run the following command to check the oplog size and window:
     ```
     rs.printReplicationInfo()
     ```
     In the following command output, **configured oplog size** indicates the total size allocated for the oplog and **log length start to end** indicates the oplog window.

     Figure 23-2 Command output

     ```
     replica:PRIMARY> rs.printReplicationInfo()
     configured oplog size:   5120MB
     log length start to end: 4907719secs (1363.26hrs)
     oplog first event time:  Wed Feb 25 2026 07:15:39 GMT+0000 (UTC)
     oplog last event time:   Thu Apr 23 2026 02:30:58 GMT+0000 (UTC)
     now:                     Thu Apr 23 2026 02:31:00 GMT+0000 (UTC)
     ```

   - Run the following command to query the size of storage actually occupied by the oplog (default unit: bytes):
     ```
     db.getSiblingDB("local").oplog.rs.stats().storageSize
     ```

NOTE

For DDS cluster instances, the oplog size of shards cannot be queried through the dds mongos. You are advised to use **storage analysis** or directly connect to a shard to check its oplog size (following the instructions in **2**). For details about how to obtain the shard IP address, see **Enabling IP Addresses of Shard and Config Nodes**.

Oplog Consumption Analysis

Plan the oplog size for a replica set based on workload patterns and system monitoring data. The following table lists the workload patterns that may require a larger oplog size.

Table 23-1 Workload patterns that may require a larger oplog size

Workload Pattern	Description
Batch updates	A single batch operation is broken down into multiple document-level updates, causing oplog entries to grow rapidly. For example, updating 1,000 documents generates 1,000 oplog entries, which significantly accelerates oplog space consumption.
Frequent alternating insertions and deletions	Insertions and deletions are individually recorded in the oplog even if the data volume does not increase. For example, 1,000 "insert-delete" cycles per second generate 2,000 oplog entries, causing constant write pressure.
Frequent in-place updates	Frequent updates (such as status field changes) to the same document do not increase storage usage, but each update generates a separate oplog entry. For example, 100 updates per second to a single document will result in a 100-fold increase in the oplog write rate.
High-concurrency writes	When the workload involves frequent write operations (such as thousands of insertions, updates, or deletions per second), the oplog write rate increases significantly. Because the oplog functions as a ring buffer that records all write operations in chronological order, intense write activity will lead to rapid exhaustion of the oplog space.

Backup Policies

- Full backup

  Hidden nodes cannot synchronize data during backup. If the backup duration exceeds the oplog window, the hidden nodes will miss the latest operation records from the primary node, making them unable to catch up with the primary node through replication.

  To avoid this issue, take the following measures:

  - Modify the backup policy to schedule backup tasks during off-peak hours. For details, see **Configuring an Automated Backup Policy**.
  - Adjust the oplog size to ensure that the oplog window adequately covers the backup duration. For details, see **Size Planning Suggestions**.
  - Enable CBR snapshot backup to minimize the lock duration on hidden nodes. For details about how to enable it, see **Configuring an Automated Backup Policy**.

- Incremental backup

  If the oplog generation rate exceeds 250 GB/h (or 75 MB/s), the incremental backup process may lag behind, resulting in unavailable restoration time points. For example, if an instance has a 20 GB oplog with a 1-hour window, the estimated oplog generation rate is 20 GB/h.

  To prevent incremental backup issues caused by excessive write speeds, consider the following measures:

  - Throttling write concurrency: Properly manage the number of concurrent write threads to prevent a surge in data writes from overwhelming the system within a short period.
  - Adjusting write concern: Change the write concern from {w:1} to {w:"majority"}. This ensures that data is returned only after being acknowledged by a majority of nodes, thereby enhancing data reliability and consistency.
  - Optimizing data distribution: For replica set instances, consider migrating data to cluster instances. Distributing data across multiple shards can fully use their storage and compute power. This improves write efficiency and system scalability.

Size Planning Suggestions

Plan the oplog size based on two critical dimensions.

Dimension	Description
Oplog generation rate	If an instance has a 20 GB oplog with a 1-hour window, the estimated oplog generation rate is 20 GB/h.
Backup duration	The time required for creating a backup depends on the data volume of your instance. If the average backup speed is 60 MB/s, the backup duration does not exceed 60% of the oplog window.

If you need to increase the oplog size, use either of the following methods:

- If storage capacity is sufficient but the oplog allocation is inadequate, increase the oplog size by modifying **oplogSizePercent** in the parameter group. For details, see **Modifying DDS Instance Parameters**.

  CAUTION

  - Before modifying **oplogSizePercent**, ensure that you have enough storage.

- If storage capacity is insufficient and you cannot increase the oplog size by modifying **oplogSizePercent**, scale up the storage for your instance. For details, see **Scaling Up Storage Space**.

