ALIBABA CLOUD

Alibaba Cloud

ApsaraDB for MongoDB
Best Practices

Document Version: 20220706

Alibaba Cloud

Legal disclaimer

Alibaba Cloud reminds you to carefully read and fully understand the terms and conditions of this legal disclaimer before you read or use this document. If you have read or used this document, it shall be deemed as your total acceptance of this legal disclaimer.

1. You shall download and obtain this document from the Alibaba Cloud website or other Alibaba Cloud-authorized channels, and use this document for your own legal business activities only. The content of this document is considered confidential information of Alibaba Cloud. You shall strictly abide by the confidentiality obligations. No part of this document shall be disclosed or provided to any third party for use without the prior written consent of Alibaba Cloud.

2. No part of this document shall be excerpted, translated, reproduced, transmitted, or disseminated by any organization, company or individual in any form or by any means without the prior written consent of Alibaba Cloud.

3. The content of this document may be changed because of product version upgrade, adjustment, or other reasons. Alibaba Cloud reserves the right to modify the content of this document without notice and an updated version of this document will be released through Alibaba Cloud-authorized channels from time to time. You should pay attention to the version changes of this document as they occur and download and obtain the most up-to-date version of this document from Alibaba Cloud-authorized channels.

4. This document serves only as a reference guide for your use of Alibaba Cloud products and services. Alibaba Cloud provides this document based on the "status quo", "being defective", and "existing functions" of its products and services. Alibaba Cloud makes every effort to provide relevant operational guidance based on existing technologies. However, Alibaba Cloud hereby makes a clear statement that it in no way guarantees the accuracy, integrity, applicability, and reliability of the content of this document, either explicitly or implicitly. Alibaba Cloud shall not take legal responsibility for any errors or lost profits incurred by any organization, company, or individual arising from download, use, or trust in this document. Alibaba Cloud shall not, under any circumstances, take responsibility for any indirect, consequential, punitive, contingent, special, or punitive damages, including lost profits arising from the use or trust in this document (even if Alibaba Cloud has been notified of the possibility of such a loss).

5. By law, all the contents in Alibaba Cloud documents, including but not limited to pictures, architecture design, page layout, and text description, are intellectual property of Alibaba Cloud and/or its affiliates. This intellectual property includes, but is not limited to, trademark rights, patent rights, copyrights, and trade secrets. No part of this document shall be used, modified, reproduced, publicly transmitted, changed, disseminated, distributed, or published without the prior written consent of Alibaba Cloud and/or its affiliates. The names owned by Alibaba Cloud shall not be used, published, or reproduced for marketing, advertising, promotion, or other purposes without the prior written consent of Alibaba Cloud. The names owned by Alibaba Cloud include, but are not limited to, "Alibaba Cloud", "Aliyun", "HiChina", and other brands of Alibaba Cloud and/or its affiliates, which appear separately or in combination, as well as the auxiliary signs and patterns of the preceding brands, or anything similar to the company names, trade names, trademarks, product or service names, domain names, patterns, logos, marks, signs, or special descriptions that third parties identify as Alibaba Cloud and/or its affiliates.

6. Please directly contact Alibaba Cloud for any errors of this document.

Document conventions

Style	Description	Example
⚠ Danger	A danger notice indicates a situation that will cause major system changes, faults, physical injuries, and other adverse results.	⚠ **Danger:** Resetting will result in the loss of user configuration data.
🔔 Warning	A warning notice indicates a situation that may cause major system changes, faults, physical injuries, and other adverse results.	🔔 **Warning:** Restarting will cause business interruption. About 10 minutes are required to restart an instance.
📢 Notice	A caution notice indicates warning information, supplementary instructions, and other content that the user must understand.	📢 **Notice:** If the weight is set to 0, the server no longer receives new requests.
? Note	A note indicates supplemental instructions, best practices, tips, and other content.	? **Note:** You can use Ctrl + A to select all files.
>	Closing angle brackets are used to indicate a multi-level menu cascade.	Click **Settings**> **Network**> **Set network type**.
Bold	Bold formatting is used for buttons , menus, page names, and other UI elements.	Click **OK**.
`Courier font`	Courier font is used for commands	Run the `cd /d C:/window` command to enter the Windows system folder.
Italic	Italic formatting is used for parameters and variables.	`bae log list --instanceid` *Instance_ID*
[] or [a|b]	This format is used for an optional value, where only one item can be selected.	`ipconfig [-all|-t]`
{} or {a|b}	This format is used for a required value, where only one item can be selected.	`switch {active|stand}`

Table of Contents

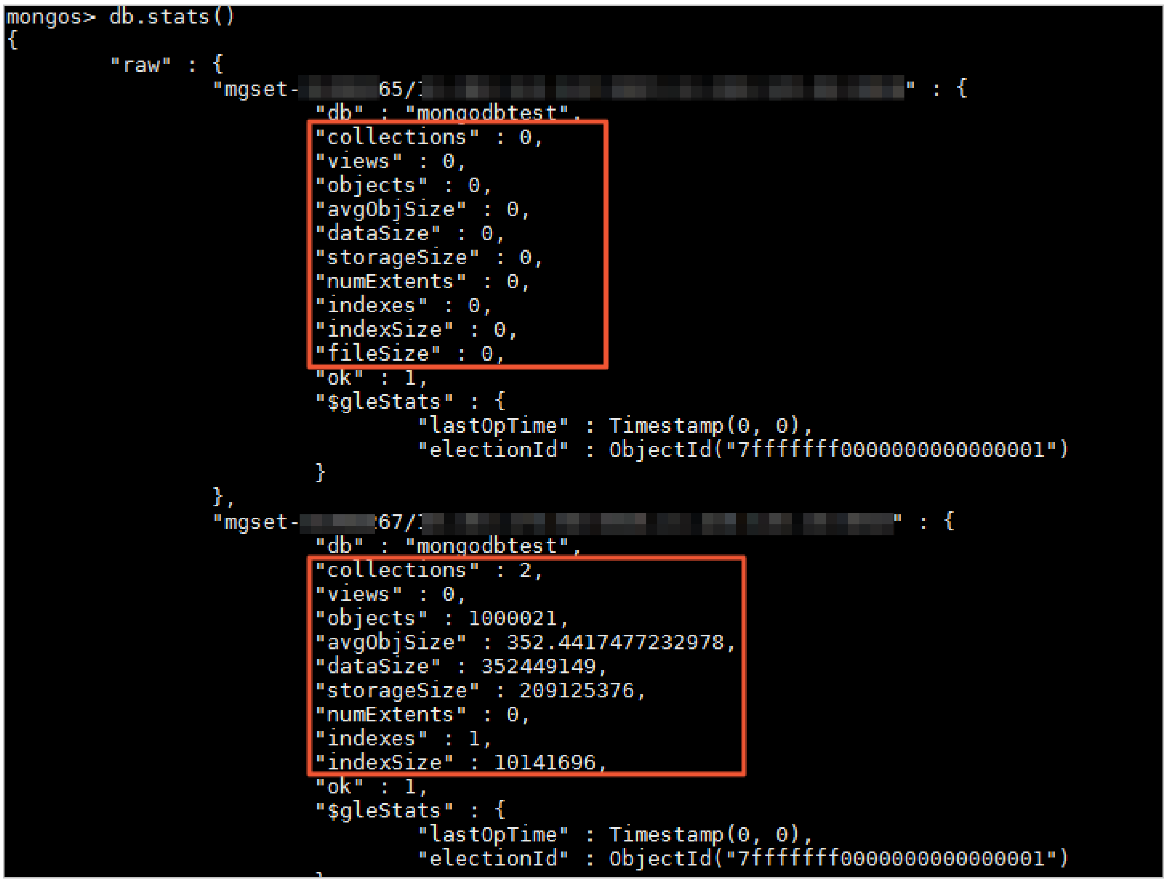
1.Performance

1.1. Configure sharding to maximize the performance of shards

You can configure sharding for each collection in a sharded cluster instance to make full use of the storage space and maximize the computing performance of shards in the sharded cluster.

Background

If a collection is not sharded, all its data is stored on the same shard. In this case, you cannot make full use of the storage space and maximize the computing performance of other shards in the sharded cluster.



```
mongos> db.stats()
{
        "raw" : {
                "mgset-[redacted]65/[redacted]" : {
                        "db" : "mongodbtest",
                        "collections" : 0,
                        "views" : 0,
                        "objects" : 0,
                        "avgObjSize" : 0,
                        "dataSize" : 0,
                        "storageSize" : 0,
                        "numExtents" : 0,
                        "indexes" : 0,
                        "indexSize" : 0,
                        "fileSize" : 0,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
                        }
                },
                "mgset-[redacted]67/[redacted]" : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 1000021,
                        "avgObjSize" : 352.4417477232978,
                        "dataSize" : 352449149,
                        "storageSize" : 209125376,
                        "numExtents" : 0,
                        "indexes" : 1,
                        "indexSize" : 10141696,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
```

Prerequisites

A sharded cluster instance is used.

Precautions

- You cannot change or delete the configured shard key after sharding.
- After you configure sharding, the balancer shards existing data that meets the specified criteria, which consumes the resources of an instance. We recommend that you perform this operation during off-peak hours.

> Note Before configuring sharding, you can set an active time window to limit the effective period of the balancer to off-peak hours. For more information, see Set an active time window for the balancer.

- The choice of a shard key affects the performance of a sharded cluster instance. For more information about how to choose a shard key, see Shard key selection.

Sharding strategies

Sharding strategy	Description	Scenario
Ranged sharding	MongoDB divides data into contiguous ranges determined by the shard key values. Each chunk represents a contiguous range of data. • Advantage: The mongos can quickly locate the data being requested and forward the requests to the target shard. • Disadvantage: Data may be distributed unevenly among shards, causing hot shards for reads and writes and an uneven spread of writes.	The shard key value is not monotonically increasing or decreasing. The shard key has large cardinality and low frequency. Range-based queries are required.
Hashed sharding	MongoDB computes the hash value of a single field as the index value and divides data into chunks based on the range of hash values. • Advantage: Data can be distributed evenly among shards, guaranteeing an even spread of writes. • Disadvantage: This strategy is inapplicable to range-based queries because read requests must be distributed across all shards during the queries.	The shard key value is monotonically increasing or decreasing. The shard key has large cardinality and low frequency. Data writes are randomly distributed to shards. Data is read with high randomness.

In addition to the preceding two sharding strategies, you can also configure a compound shard key. For example, configure both a key with low cardinality and a monotonically increasing key. For more information, see .

Procedure

The following procedure uses the database named mongodbtest and the collection named customer as an example.

1. Connect to a sharded cluster instance by using the mongo shell.
2. Enable sharding for the database where the collection to be sharded resides.

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

> Note <database>: the name of the database.

Example:

```
sh.enableSharding("mongodbtest")
```

> **Note** You can run the `sh.status()` command to check whether sharding is enabled.

3. Create an index on the shard key field.

```
db.<collection>.createIndex(<keyPatterns>,<options>)
```

> **Note**
>
> - <collection>: the name of the collection.
> - <keyPatterns>: the field used for indexing and the index type.
>
>   Common index types are as follows:
>
>   - 1: an ascending index
>   - -1: a descending index
>   - "hashed": a hashed index
> - <options>: the optional parameters. For more information, see db.collection.createIndex(). This field is not used in this example.

Sample command for creating an ascending index:

```
db.customer.createIndex({name:1})
```

Sample command for creating a hashed index:

```
db.customer.createIndex({name:"hashed"})
```

4. Configure sharding for the collection.

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

> **Note**
>
> - <database>: the name of the database.
> - <collection>: the name of the collection.
> - <key>: the shard key that MongoDB uses to shard data.
> - <value>
>   - 1: ranged sharding. This strategy supports efficient range-based queries based on the shard key.
>   - "hashed": hashed sharding. This strategy distributes data evenly among shards.

Sample command for configuring ranged sharding:

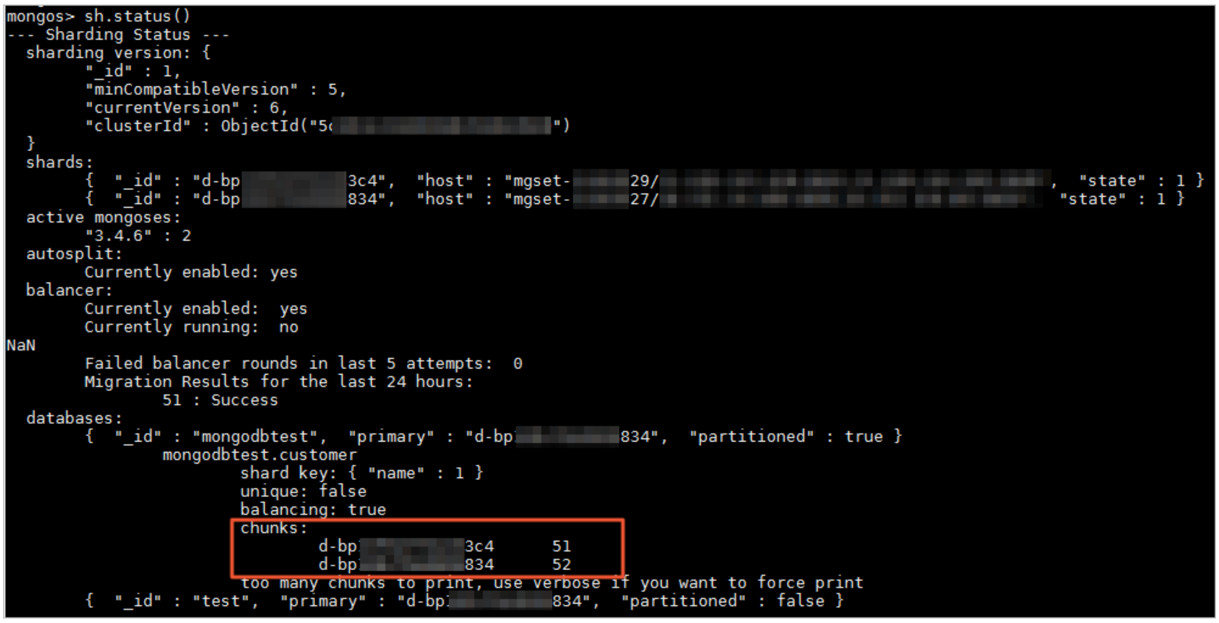
```
sh.shardCollection("mongodbtest.customer",{"name":1})
```

Sample command for configuring hashed sharding:

```
sh.shardCollection("mongodbtest.customer",{"name":"hashed"})
```

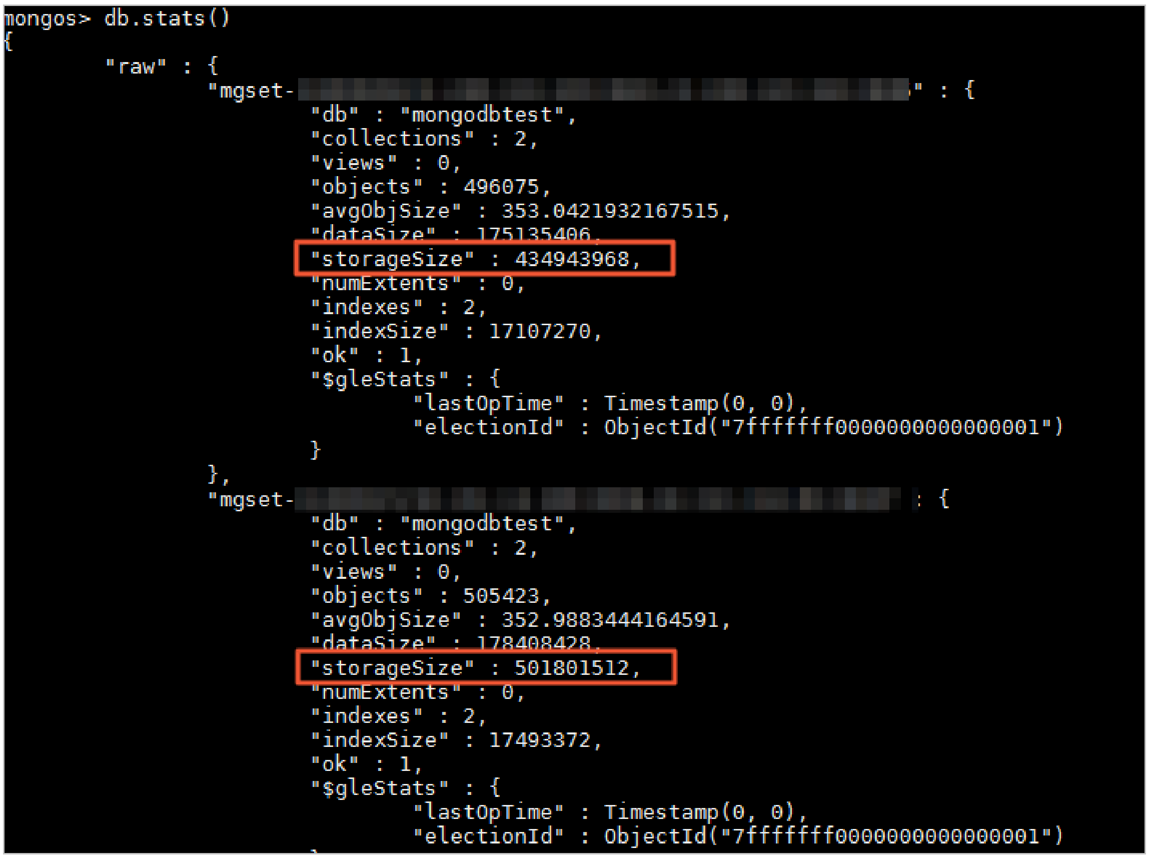
What to do next

After the database has been running and data has been written for a while, you can run the `sh.status()` command in the mongo shell to check the chunk information on shards.



```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5[redacted]")
  }
  shards:
        {  "_id" : "d-bp[redacted]3c4",  "host" : "mgset-[redacted]29/[redacted],  "state" : 1 }
        {  "_id" : "d-bp[redacted]834",  "host" : "mgset-[redacted]27/[redacted]  "state" : 1 }
  active mongoses:
        "3.4.6" : 2
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
NaN
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                51 : Success
  databases:
        {  "_id" : "mongodbtest",  "primary" : "d-bp[redacted]834",  "partitioned" : true }
                mongodbtest.customer
                        shard key: { "name" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                d-bp[redacted]3c4      51
                                d-bp[redacted]834      52
                        too many chunks to print, use verbose if you want to force print
        {  "_id" : "test",  "primary" : "d-bp[redacted]834",  "partitioned" : false }
```

You can also run the `db.stats()` command to check the size of data stored on each shard.



```
mongos> db.stats()
{
        "raw" : {
                "mgset-[redacted]" : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 496075,
                        "avgObjSize" : 353.0421932167515,
                        "dataSize" : 175135406,
                        "storageSize" : 434943968,
                        "numExtents" : 0,
                        "indexes" : 2,
                        "indexSize" : 17107270,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
                        }
                },
                "mgset-[redacted] : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 505423,
                        "avgObjSize" : 352.9883444164591,
                        "dataSize" : 178408428,
                        "storageSize" : 501801512,
                        "numExtents" : 0,
                        "indexes" : 2,
                        "indexSize" : 17493372,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
```

1.2. Defragment the disk space to improve disk usage

Fragmentation may occur in the disk space when you frequently write and delete large amounts of data in ApsaraDB for MongoDB databases. The fragments occupy disk space and reduce disk usage. You can rewrite and defragment all the data and indexes in a collection to release idle space. This improves disk usage and query performance.

Prerequisites

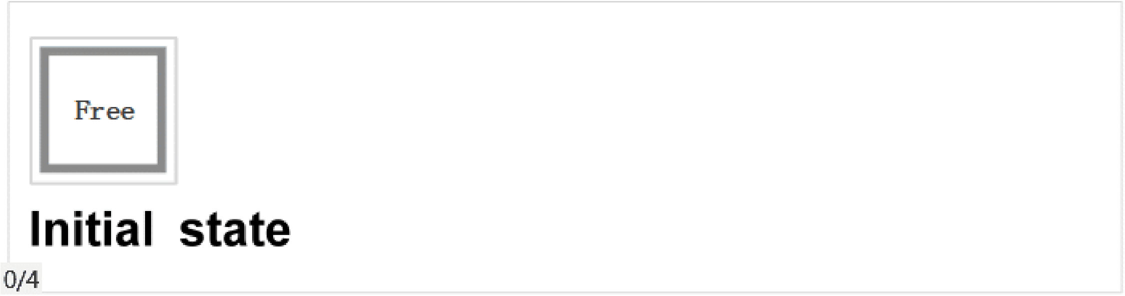
The ApsaraDB for MongoDB instance uses WiredTiger as the storage engine.

Important notes

- We recommend that you back up data in ApsaraDB for MongoDB databases before defragmentation. For more information, see Manually back up an ApsaraDB for MongoDB instance.
- During defragmentation, the database where the collection is stored is locked and read and write operations are not allowed in the database. We recommend that you defragment the disk space during off-peak hours.

> **Note** The time required for defragmenting the disk space through the compact command depends on multiple factors, such as the data volume of the collection and the system load.

Background



Generally, if you run the `db.collection.remove({}, {multi: true})` command to delete a document from the B tree, the disk space occupied by the document is not reclaimed. If you run the remove command to delete a large number of documents, but write little data to the disk later, disk usage is reduced. In this case, you can run the compact command to reclaim the idle disk space.

> **Note**
>
> - The newly written data occupies the disk space that is not reclaimed. Therefore, you do not need to frequently run the compact command for defragmentation in scenarios where data is continuously written.
> - If you run the `db.collection.drop()` command to delete a collection, the files in the collection are deleted and the disk space occupied by the files is reclaimed.

Estimate the disk space to be reclaimed

1. Connect to the ApsaraDB for MongoDB instance by using the mongo shell. For more information, see the following topics:
    - Connect to a standalone ApsaraDB for MongoDB instance through DMS
    - Connect to a replica set instance by using the mongo shell
    - Connect to a sharded cluster instance by using the mongo shell
2. Run the following command to switch to the database where the collection is stored:

    ```
    use <database_name>
    ```

    ⓘ **Note** <database_name>: the name of the database.

3. Run the following command to query the disk space that can be reclaimed from the collection:

    ```
    db.<collection_name>.stats().wiredTiger["block-manager"]["file bytes available for reus
    e"]
    ```

    ⓘ **Note** <collection_name>: the name of the collection.

    Sample command:

    ```
    db.customer.stats().wiredTiger["block-manager"]["file bytes available for reuse"]
    ```

    Sample result:

    ```
    207806464
    ```

Defragment a standalone instance or a replica set instance

1. Connect to the primary node of an ApsaraDB for MongoDB instance by using the mongo shell. For more information, see Connect to a replica set instance by using the mongo shell.
2. Run the following command to switch to the database where the collection is stored:

    ```
    use <database_name>
    ```

    ⓘ **Note** <database_name>: the name of the database.

3. Run the `db.stats()` command to view the disk space occupied by the database before defragmentation.
4. Run the following command to defragment a collection:

    ```
    db.runCommand({compact:"<collection_name>",force:true})
    ```

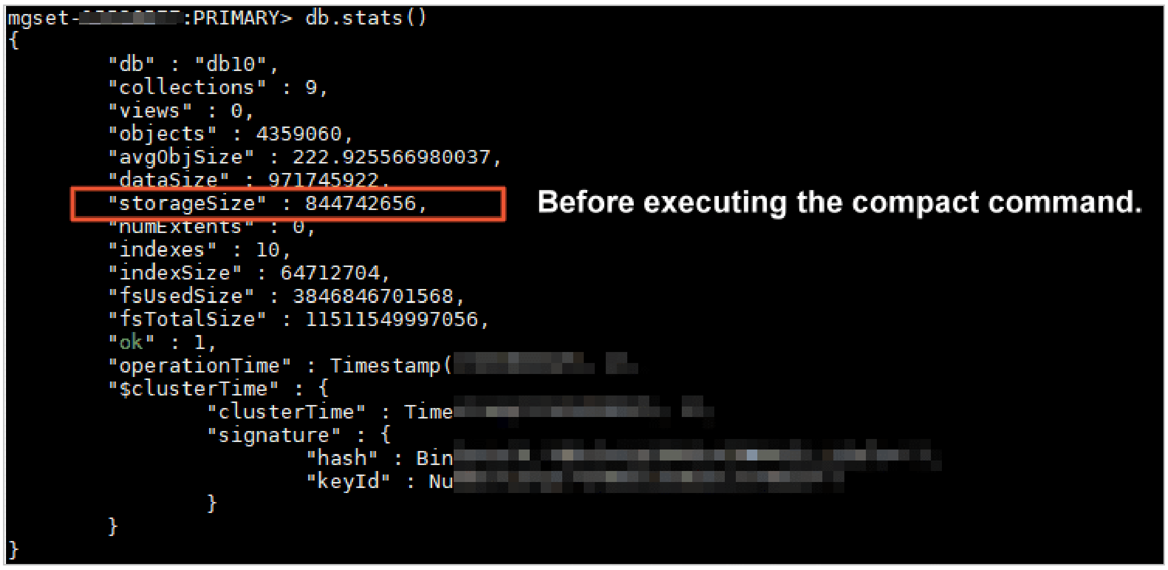
    ⓘ **Note**

    - <collection_name>: the name of the collection.
    - The `force` parameter is optional. To run the compact command on the primary node of a replica set instance, you must set the `force` parameter to *true*.

5. Wait until `{"ok":1}` is returned, indicating that the command is executed.

> Note The compact command executed on the primary node does not affect a secondary node. For a replica set instance, repeat the preceding steps to connect to a secondary node through the mongo shell and run the compact command.

After defragmentation is complete, you can run the `db.stats()` command to view the disk space occupied by the database. The following figure shows the storage size before and after defragmentation.



```
mgset-[redacted]:PRIMARY> db.stats()
{
        "db" : "db10",
        "collections" : 9,
        "views" : 0,
        "objects" : 4359060,
        "avgObjSize" : 222.925566980037,
        "dataSize" : 971745922,
        "storageSize" : 844742656,
        "numExtents" : 0,
        "indexes" : 10,
        "indexSize" : 64712704,
        "fsUsedSize" : 3846846701568,
        "fsTotalSize" : 11511549997056,
        "ok" : 1,
        "operationTime" : Timestamp([redacted]
        "$clusterTime" : {
                "clusterTime" : Time[redacted]
                "signature" : {
                        "hash" : Bin[redacted]
                        "keyId" : Nu[redacted]
                }
        }
}
```

Before executing the compact command.

Defragment a sharded cluster instance

1. Connect to any mongos node in the sharded cluster instance by using the mongo shell. For more information, see Connect to a sharded cluster instance by using the mongo shell.
2. Run the `db.stats()` command to view the disk space occupied by the database before defragmentation.
3. Run the following command to defragment a collection on the primary node of a shard:

```
db.runCommand({runCommandOnShard:"<Shard ID>","command":{compact:"<collection_name>",force:true}})
```

> Note
> - <Shard ID>: the ID of the shard.
> - <collection_name>: the name of the collection.

4. Run the following command to defragment a collection on a secondary node of a shard:

```
db.runCommand({runCommandOnShard:"<Shard ID>","command":{compact:"<collection_name>"},$queryOptions: {$readPreference: {mode: 'secondary'}}})
```

> ? Note
> - <Shard ID>: the ID of the shard.
> - <collection_name>: the name of the collection.

After defragmentation is complete, you can run the `db.runCommand({dbstats:1})` command to view the disk space occupied by the database.

1.3. Troubleshoot the high CPU utilization of ApsaraDB for MongoDB

When you use ApsaraDB for MongoDB, the CPU utilization may become excessively high or even approach 100%. A high CPU utilization slows down read and write operations and affects normal business operations. This topic describes how to troubleshoot the high CPU utilization of ApsaraDB for MongoDB for your applications.

Analyze running requests in ApsaraDB for MongoDB databases

1. Connect to an ApsaraDB for MongoDB instance by using the mongo shell.

   Connection methods vary with instance architecture. You can refer to the following topics for details:

   - Connect to a standalone ApsaraDB for MongoDB instance by using the mongo shell
   - Connect to a replica set instance by using the mongo shell
   - Connect to a sharded cluster ApsaraDB for MongoDB instance by using the mongo shell

2. Run the `db.currentOp()` command to check running operations in ApsaraDB for MongoDB databases.

   The following code provides an example of the command output:

```
{
        "desc" : "conn632530",
        "threadId" : "140298196924160",
        "connectionId" : 632530,
        "client" : "11.192.159.236:57052",
        "active" : true,
        "opid" : 1008837885,
        "secs_running" : 0,
        "microsecs_running" : NumberLong(70),
        "op" : "update",
        "ns" : "mygame.players",
        "query" : {
            "uid" : NumberLong(31577677)
        },
        "numYields" : 0,
        "locks" : {
            "Global" : "w",
            "Database" : "w",
            "Collection" : "w"
        },
        ....
    }
```

The following table describes the fields to which you need to pay close attention.

Parameter	Description
`client`	The client that sent the request.
`opid`	The unique ID of the operation. If necessary, you can run the `db.killOp(opid)` command to terminate the operation.
`secs_running`	The duration that the operation has been running. Unit: seconds. If a greater value is returned for this field, check whether the request is appropriate.
`microsecs_running`	The duration that the operation has been running. Unit: microseconds. If a greater value is returned for this field, check whether the request is appropriate.
`ns`	The collection on which the operation performs a scan.
`op`	The operation type. In most cases, this value is query, insert, update, or delete.
`locks`	The lock-related information. For more information, see FAQ: Concurrency.

Note For more information about the `db.currentOp()` command, see db.currentOp().

You can run the `db.currentOp()` command to check running operations and analyze whether ApsaraDB for MongoDB is processing time-consuming requests. For example, the CPU utilization is not high for your routine business. However, when an O&M personnel logs on to ApsaraDB for MongoDB databases to perform specific operations that require a collection scan, the CPU utilization significantly increases and ApsaraDB for MongoDB becomes sluggish. In this case, you must check for time-consuming operations.

> **Note** If you find an abnormal request, you can obtain the operation ID (`opid`) of this request and run the `db.killOp(opid)` command to terminate this request.
>
> For more information about the `db.killOp()` command, see db.killOp().

Analyze slow requests in ApsaraDB for MongoDB databases

If the CPU utilization of your ApsaraDB for MongoDB instance immediately increases and remains high after your application starts running and you cannot find abnormal requests in the output of the `db.currentOp()` command, you can analyze slow requests in the instance databases.

1. View the slow query logs of the instance in the ApsaraDB for MongoDB console. For more information, see View slow query logs.
2. Analyze the slow query logs to troubleshoot high CPU utilization for the instance.

   The following example shows a slow query log. For this request, ApsaraDB for MongoDB runs a collection scan on 11,000,000 documents, instead of querying data based on an index.

```
{
        "op" : "query",
        "ns" : "123.testCollection",
        "command" : {
                "find" : "testCollection",
                "filter" : {
                        "name" : "zhangsan"
                },
                "$db" : "123"
        },
        "keysExamined" : 0,
        "docsExamined" : 11000000,
        "cursorExhausted" : true,
        "numYield" : 85977,
        "nreturned" : 0,
        "locks" : {
                "Global" : {
                        "acquireCount" : {
                                "r" : NumberLong(85978)
                        }
                },
                "Database" : {
                        "acquireCount" : {
                                "r" : NumberLong(85978)
                        }
                },
                "Collection" : {
                        "acquireCount" : {
                                "r" : NumberLong(85978)
                        }
                }
        },
        "responseLength" : 232,
        "protocol" : "op_command",
        "millis" : 19428,
        "planSummary" : "COLLSCAN",
        "execStats" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "name" : {
                                "$eq" : "zhangsan"
                        }
                },
                "nReturned" : 0,
                "executionTimeMillisEstimate" : 18233,
                "works" : 11000002,
                "advanced" : 0,
                "needTime" : 11000001,
                "needYield" : 0,
                "saveState" : 85977,
                "restoreState" : 85977,
                "isEOF" : 1,
                "invalidates" : 0,
                "direction" : "forward",
....in"
                }
        ],
        "user" : "root@admin"
}
```

For slow query logs, you must pay close attention to the following items:

- Collection scan (keywords: `COLLSCAN` and `docsExamined`)
    - `COLLSCAN` indicates a collection scan.

      A collection scan for a request (such as a query, update, or delete operation) may cause a high CPU utilization. If you find a `COLLSCAN` keyword in slow query logs, your CPU resources may have been occupied by these requests.

      > **Note** If such requests are frequently submitted, we recommend that you create an index on queried fields to optimize query performance.

    - The `docsExamined` field indicates the number of documents that ApsaraDB for MongoDB has scanned for a request. The greater the field value is, the more CPU resources this request occupies.
- Inappropriate indexes (keywords: `IXSCAN` and `keysExamined`)

Note

- Excessive indexes affect the write and update performance.
- If your application involves a large number of write operations and you use indexes, the application performance may be affected.

The `keysExamined` field indicates the number of index keys that ApsaraDB for MongoDB has scanned for a request that uses an index. The greater the field value is, the more CPU resources this request occupies.

If you create an index that is inappropriate or matches a large amount of data, the index cannot reduce CPU overheads or accelerate the execution of a request.

For example, for the data in a collection, the x field can be set only to 1 or 2, whereas the y field has a wider value range.

```
{ x: 1, y: 1 }
{ x: 1, y: 2 }
{ x: 1, y: 3 }
......
{ x: 1, y: 100000}
{ x: 2, y: 1 }
{ x: 2, y: 2 }
{ x: 2, y: 3 }
......
{ x: 1, y: 100000}
```

To query data {x: 1, y: 2}, you can create an index.

```
db.createIndex( {x: 1} )         //This index is inappropriate because a large amount of
data has the same value as the x field.
db.createIndex( {x: 1, y: 1} )   //This index is inappropriate because a large amount of
data has the same value as the x field.
db.createIndex( {y: 1 } )        //This index is appropriate because a small amount of da
ta has the same value as the y field.
db.createIndex( {y: 1, x: 1 } )  //This index is appropriate because a small amount of da
ta has the same value as the y field.
```

Note For the difference between indexes {y: 1} and {y: 1, x: 1}, see Design Principles of MongoDB Indexes and Compound Indexes.

- Sorting of a large amount of data (keywords: `SORT` and `hasSortStage`)

The value of the `hasSortStage` field is `true` when a query contains a sort order. If the query cannot use an index to return the requested sorted results in order, ApsaraDB for MongoDB must sort the query results. A sort operation may cause a high CPU utilization. In response to this issue, you can create an index on frequently sorted fields to optimize sorting performance.

Note If you find the `SORT` keyword in slow query logs, you can use an index to optimize sorting performance.

Other operations such as index creation and `aggregation` (a combination of traverse, query, update, and sort) may also cause a high CPU utilization. You can also use the preceding troubleshooting methods.

Assess service capabilities

After you analyze and optimize running requests and slow requests in ApsaraDB for MongoDB databases, all requests are appropriate and efficiently use indexes. If CPU resources are still fully occupied, your instances may have reached the maximum capabilities. In this case, we recommend that you use the following method to address the issue:

1. View the monitoring information to analyze the resource usage of instances. For more information, see View monitoring data and Basic monitoring.
2. Check whether current instances meet the performance and capability requirements of your business scenarios.

For information about how to upgrade instances, see Overview or Change the configurations of a replica set instance.

1.4. Connect to a replica set instance and implement read /write splitting and high availability

An ApsaraDB for MongoDB replica set instance provides multiple copies of data to ensure the high reliability of data. It also provides an automatic failover mechanism to guarantee the high availability of ApsaraDB for MongoDB. You must use a correct method to connect to a replica set instance to implement high availability. You can also configure the connection for read /write splitting.

Notes

- The primary node of a replica set instance may change. A failover between the primary and secondary nodes may be triggered when nodes of the replica set instance are upgraded in turn, the primary node is faulty, or the network is partitioned. In these scenarios, the replica set can elect a new primary node and downgrade the original primary node to a secondary node.
- If the primary node of a replica set instance is directly connected through the connection string of the primary node, the primary node will bear heavy load to process all read and write operations. If a failover is triggered in the replica set instance and the connected primary node is downgraded to a secondary node, you can no longer perform write operations and your business is seriously affected.

Connection string URIs

To correctly connect to a replica set instance, you must understand the format of connection string URIs of MongoDB. All official MongoDB drivers allow you to use a connection string URI to connect to MongoDB.

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database]
[?options]]
```

Parameter description:

- `mongodb://` : the prefix, indicating a connection string URI.

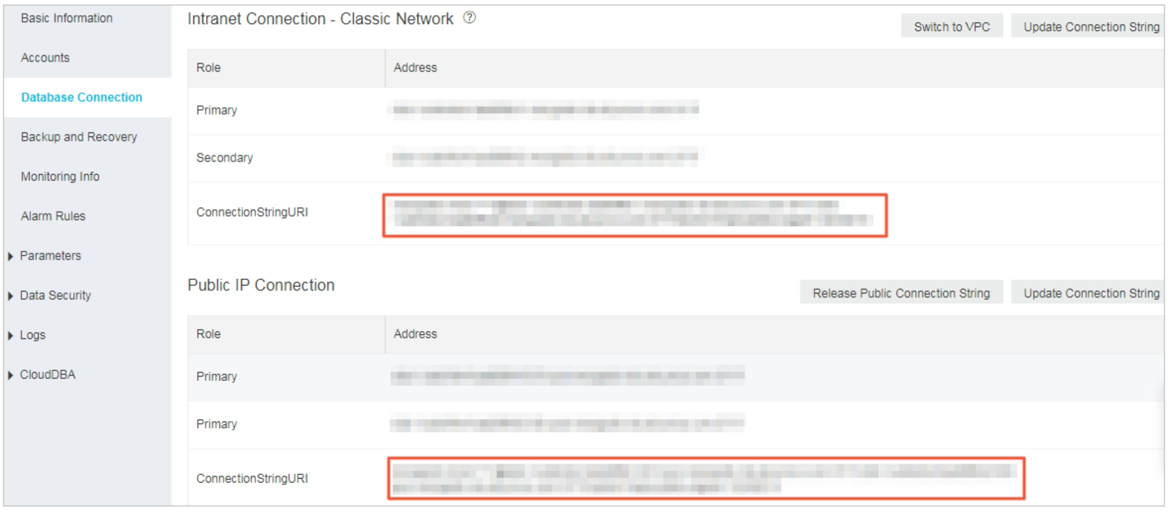
- `username:password@` : the username and password used to log on to the database. If authentication is enabled, a password is required.
- `hostX:portX` : the list of connection strings used to connect to nodes in the replica set instance. Each connection string consists of an IP address and a port number. Separate multiple connection strings with commas (,).
- `/database` : the database corresponding to the username and password if authentication is enabled.
- `?options` : additional connection options.

> **Note** For more information about connection string URIs, see connection String URI Format.

Use a connection string URI to connect to a replica set instance

You can use a connection string URI to connect to a replica set instance.

1. Obtain the connection string URI of a replica set instance. For more information, see Overview of replica set instance connections.



Intranet Connection - Classic Network

Role	Address
Primary	[redacted]
Secondary	[redacted]
ConnectionStringURI	[redacted]

Public IP Connection

Role	Address
Primary	[redacted]
Primary	[redacted]
ConnectionStringURI	[redacted]

2. Use the obtained connection string URI to connect your applications to the instance. For more information, see Connection sample code for MongoDB drivers.

> **Note**
>
> For read /write splitting, you must add `readPreference=secondaryPreferred` in the options parameter to set read preference to secondary nodes.
>
> For more information about read preference options, see Read Preference.

Example:

```
mongodb://root:xxxxxxxx@dds-xxxxxxxxxxxx:3717,xxxxxxxxxxxx:3717/admin? replicaSet=mgset
-xxxxxx&readPreference=secondaryPreferred
```

After you use the preceding method to connect to a replica set instance, a client can preferentially send read requests to secondary nodes to implement read/write splitting. The client also automatically detects the relationship between the primary and secondary nodes. If the primary node changes, the client automatically switches over write operations to the new primary node to ensure the high availability of ApsaraDB for MongoDB.

1.5. Import and export MongoDB data through Data Integration

Data Integration is a stable, efficient, and scalable data synchronization platform provided by DataWorks. It supports batch transmission of data for Alibaba Cloud services such as MaxCompute, AnalyticDB, and OSS. This topic describes how to import and export MongoDB data through Data Integration.

For more information about how to import and export MongoDB data through Data Integration, see Configure a MongoDB reader and Configure a MongoDB writer.

2.Best practices for data security of ApsaraDB for MongoDB

ApsaraDB for MongoDB provides comprehensive security protection to eliminate your data security concerns. You can secure the data in your ApsaraDB for MongoDB instance by using zone-disaster recovery, Resource Access Management (RAM) authorization, audit logs, network isolation, IP address whitelists, and password authentication.

Zone-disaster recovery

ApsaraDB for MongoDB provides a zone-disaster recovery solution to achieve high reliability and high data security. This solution allows you to deploy the nodes of a replica set instance or sharded cluster instance in three different zones of the same region. If one of the three zones is disconnected due to force majeure factors such as blackouts and network faults, ApsaraDB for MongoDB automatically triggers a failover to ensure service availability and data security.

You can select multiple zones when you create an ApsaraDB for MongoDB instance. For more information, see Create a multi-zone replica set instance or Create a multi-zone sharded cluster instance. You can also migrate an existing replica set instance or sharded cluster instance to multiple zones. For more information, see Migrate an ApsaraDB for MongoDB instance to different zones in the same region.

> ? **Note** You can migrate an ApsaraDB for MongoDB instance across zones only when the instance is a replica set instance or sharded cluster instance that runs MongoDB 4.2 or earlier and transparent data encryption (TDE) is not enabled for the instance.

Access control

- Authorize RAM users to manage specific ApsaraDB for MongoDB instances.

  You can use RAM to create and manage RAM users. You can also use RAM to control the permissions of the created RAM users on the resources that are available within your Alibaba Cloud account. If multiple users in your enterprise need to simultaneously use the same resources, you can use RAM to assign the least permissions to the users. This prevents the users from sharing the same key and reduces the information security risks for your enterprise.

  For more information, see How to configure RAM user permissions on ApsaraDB for MongoDB.

- Create accounts on an ApsaraDB for MongoDB instance and grant permissions to the accounts.

  In a production environment, do not connect to an ApsaraDB for MongoDB instance by using the credentials of the root account. You can create accounts on the instance and grant permissions to the created accounts.

  For more information, see Manage user permissions on MongoDB databases.

Network isolation

- Deploy ApsaraDB for MongoDB instances in virtual private clouds (VPCs).

  ApsaraDB for MongoDB supports various networks. We recommend that you deploy ApsaraDB for MongoDB instances in VPCs.

A VPC is an isolated virtual network that provides higher security and higher performance than the classic network. Before you deploy ApsaraDB for MongoDB instances in VPCs, you must create VPCs. For more information, see Default VPC and default vSwitch.

If an ApsaraDB for MongoDB instance is deployed in the classic network, you can migrate the instance to a VPC. For more information, see Switch the network type of an ApsaraDB for MongoDB instance from classic network to VPC. If an ApsaraDB for MongoDB instance is deployed in a VPC, no further action is required.

> ⓘ **Note** ApsaraDB for MongoDB supports password-free access over VPCs. VPCs provide a convenient, secure method to connect to ApsaraDB for MongoDB instances. For more information, see Enable or disable password-free access over a VPC for an ApsaraDB for MongoDB instance.

- Configure IP address whitelists.

  After an ApsaraDB for MongoDB instance is created, a default IP address whitelist is created. The default IP address whitelist contains only the `127.0.0.1` IP address. Before you can connect to the ApsaraDB for MongoDB instance, you must manually configure the IP address whitelist.

  For more information, see Configure a whitelist or an ECS security group for an ApsaraDB for MongoDB instance.

> ⓘ **Note**
>
> - Do not add the `0.0.0.0/0` entry to an IP address whitelist. The 0.0.0.0/0 entry indicates that the ApsaraDB for MongoDB instance can be accessed from all IP addresses.
> - We recommend that you configure IP address whitelists based on your business requirements and update the configured IP address whitelists on a regular basis. After you confirm that an IP address no longer requires access to the ApsaraDB for MongoDB instance, we recommend that you immediately delete the IP address.

Audit logs

The audit logs of an ApsaraDB for MongoDB instance record all operations that are performed on the instance. The audit logs help you obtain information about the operations that are performed on the data in the instance. You can analyze the audit logs to troubleshoot issues, identify abnormal behavior, and audit the security of the instance.

For more information, see View audit logs.

Data encryption

- SSL encryption

  If you connect to an ApsaraDB for MongoDB instance over the Internet, you can enable SSL encryption for the instance. SSL encryption helps protect the data in transit. ApsaraDB for MongoDB encrypts network connections at the transport layer in compliance with SSL to improve data security and ensure data integrity. For more information, see Use the mongo shell to connect to an ApsaraDB for MongoDB database in SSL encryption mode.

- TDE

TDE is used to encrypt data before the data is written from data files into a disk and decrypts data before the data is read from a disk and written into the memory. TDE does not increase the size of data files. You can use TDE without the need to modify the configuration data of your application. For more information, see Configure TDE for an ApsaraDB for MongoDB instance.

ⓘ **Note** TDE supports only collection-level encryption. For more information about field-level encryption, see Explicit (Manual) Client-Side Field Level Encryption. Field-level encryption is supported only by ApsaraDB for MongoDB instances that run MongoDB 4.2.

3.Set common alert rules for ApsaraDB for MongoDB

ApsaraDB for MongoDB provides the instance monitoring and alerting feature. This topic describes how to configure common metrics such as disk usage, input/output operations per second (IOPS), connections, and CPU utilization.

Background information

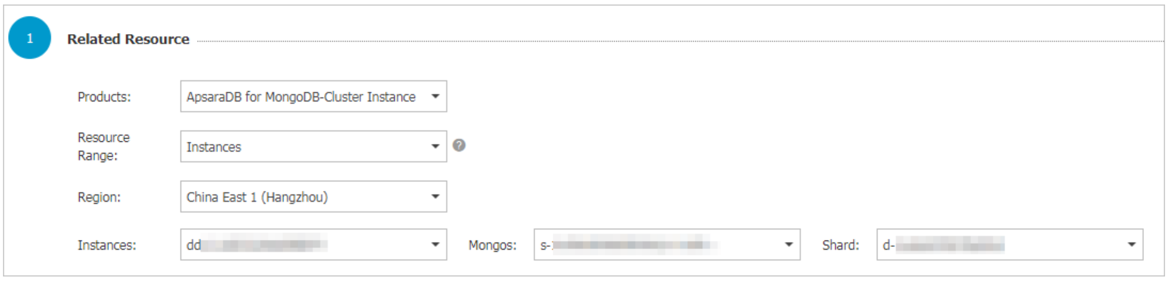
- With the growth of business data, more performance resources of ApsaraDB for MongoDB instances are consumed. Sometimes performance resources may even be used up.
- For example, when a great number of slow queries occur, writing a large amount of data causes performance resources of ApsaraDB for MongoDB instances exceptionally consumed.

> **Note** Instances with insufficient disk space may be locked. If an instance is locked, you can submit a ticket. After that, you can increase disk space with the configuration change feature.

You can set alert rules for key performance metrics of instances to help you detect abnormal data and troubleshoot errors.

Procedure

1.
2.
3.
4.
5. In the left-side navigation pane, click **Alert Rules**.
6. Click **Set Alert Rule**. You are redirected to the **Cloud Monitor console**.
7. On the Threshold Value Alert tab of the **Cloud Monitor console**, click **Create Alert Rule** in the upper-right corner.
8. On the **Create Alert Rule** page, configure related resource parameters.



Products: ApsaraDB for MongoDB-Cluster Instance

Resource Range: Instances

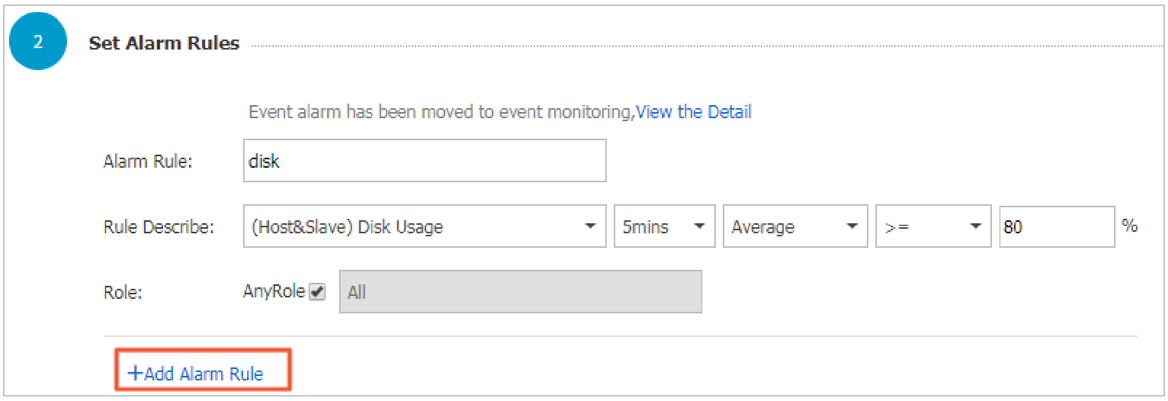
Region: China East 1 (Hangzhou)

Instances: dd[redacted] Mongos: s-[redacted] Shard: d-[redacted]

Parameter	Description

Parameter	Description
Products	The architecture type of the instance. Valid values: ○ ApsaraDB for MongoDB-Instance Copy ○ ApsaraDB for MongoDB-Cluster Instance ○ ApsaraDB for MongoDB-Single Node Instance Note If you select **ApsaraDB for MongoDB-Cluster Instance**, you must specify the **mongos** and **shard** nodes to be monitored.
Resource Range	○ **All Resources**: The alert rule is applicable to all ApsaraDB for MongoDB instances. ○ **Instances**: The alert rule is applicable to the specified ApsaraDB for MongoDB instances.
Region	The region where the instance is deployed.
Instances	The ID of the instance. You can select multiple instance IDs.

9. Set alert rules. You can set the disk usage first, and then click **Add Alert Rule**.



2 Set Alarm Rules

Event alarm has been moved to event monitoring,View the Detail

Alarm Rule: disk

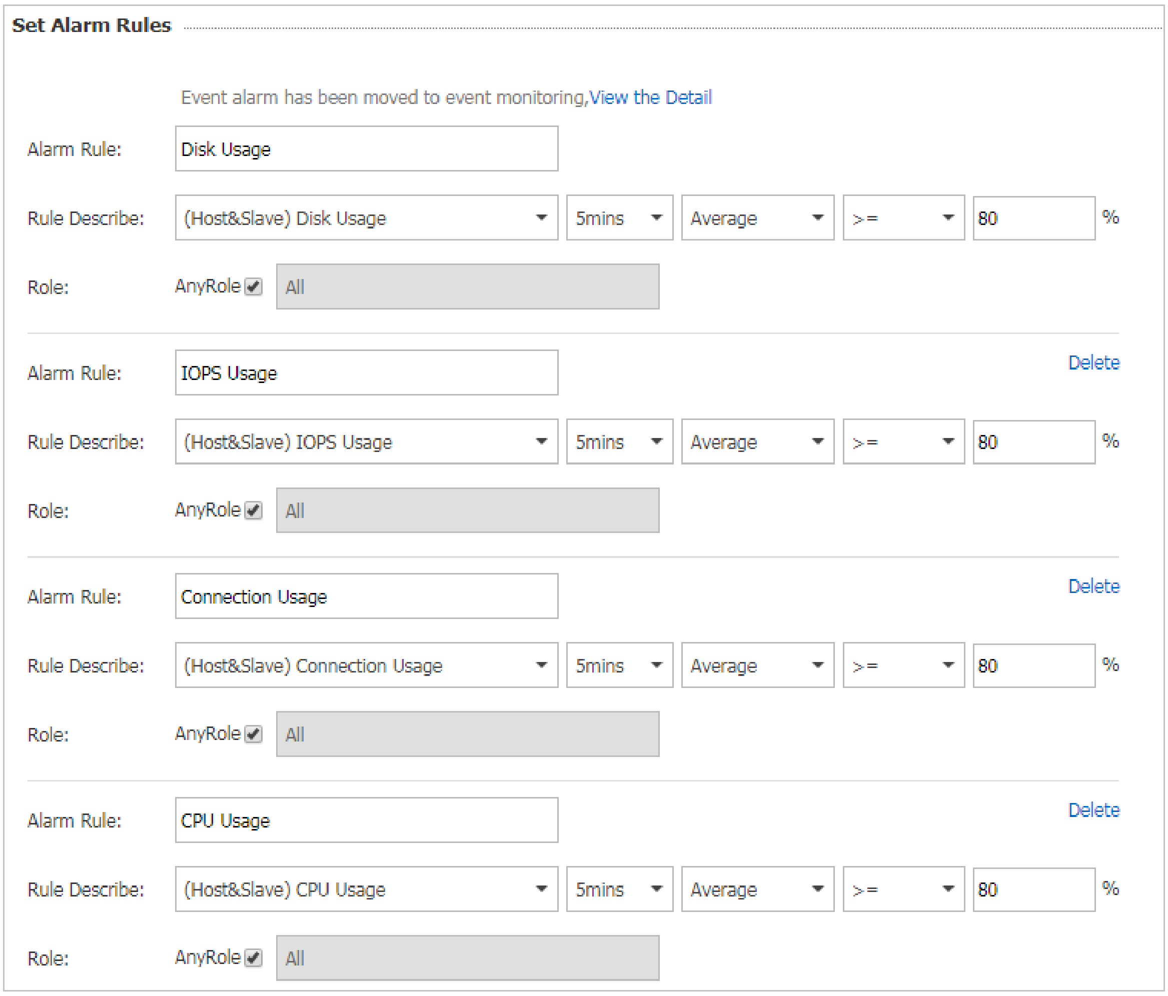
Rule Describe: (Host&Slave) Disk Usage 5mins Average >= 80 %

Role: AnyRole ☑ All

Note

- If you set Rule Description to Disk Usage 5mins Average >= 80%, the alerting module checks the disk usage every 5 minutes to detect whether the average disk usage in the last 5 minutes is greater than or equal to 80%. You can adjust each alert threshold based on your business scenarios.
- If you select AnyRole for the Role parameter, the primary and secondary nodes for each instance are monitored.

10. Set alert rules for IOPS, connections, and CPU utilization in the similar way to disk usage.



Set Alarm Rules

Event alarm has been moved to event monitoring,View the Detail

Alarm Rule:	Disk Usage					
Rule Describe:	(Host&Slave) Disk Usage	5mins	Average	>=	80	%
Role:	AnyRole ☑	All				
Alarm Rule:	IOPS Usage					Delete
Rule Describe:	(Host&Slave) IOPS Usage	5mins	Average	>=	80	%
Role:	AnyRole ☑	All				
Alarm Rule:	Connection Usage					Delete
Rule Describe:	(Host&Slave) Connection Usage	5mins	Average	>=	80	%
Role:	AnyRole ☑	All				
Alarm Rule:	CPU Usage					Delete
Rule Describe:	(Host&Slave) CPU Usage	5mins	Average	>=	80	%
Role:	AnyRole ☑	All				

11. Set other parameters for alert rules.

Parameter	Description
Mute for	Specifies the mute period. If the alert is not cleared within the mute period, a new alert notification is sent when the mute period ends.
Effective Period	The period when the alert rule takes effect.

12. Set the notification method.

Parameter	Description
Notification Contact	The contacts or contact group to which an alert notification is sent. For more information, see Manage an alert contact or alert group.
Notification Methods	The notification methods corresponding to an alert severity, which can be Critical, Warning, or Info. ○ Critical: phone calls, SMS messages, emails, and DingTalk ChatBot. ○ Warning: SMS messages, emails, and DingTalk ChatBot. ○ Info: emails and DingTalk ChatBot.

Parameter	Description
Email Subject	The subject of the alert notification email. The default email subject is in the format of service name + metric name + instance ID.
Email Remark	The custom additional information in the alert notification email. Remarks will be included in the alert notification email if you specify this parameter.
HTTP Callback	For more information, see Use the alert callback feature.

13. Click **Confirm**. Alert rules automatically take effect.

4.Migrate Azure Cosmos DB's API for MongoDB to ApsaraDB for MongoDB

MongoDB provides native backup utilities that you can use to migrate Azure Cosmos DB's API for MongoDB to ApsaraDB for MongoDB.

Precautions

- This is a full migration. To ensure data consistency we recommend that you stop all write operations to the database before migration.
- If you have used mongodump commands to back up the database, move the files in the dump folder to other directories. Make sure that the default dump folder is empty before data migration. Otherwise, existing backup files in this folder will be overwritten.
- Run mongodump and mongorestore commands on servers on which MongoDB is installed. Do not run these commands in the mongo shell.

Required database account permissions

Instance	Account permission
Azure Cosmos DB	Read
Destination MongoDB instance	Read and write

Environment configuration

1. Create an ApsaraDB for MongoDB instance. For more information, see Create an instance.

   > Note
   > - The instance storage capacity must be larger than Azure Cosmos DB.
   > - Select MongoDB version 3.4.

2. Set a password for the ApsaraDB for MongoDB instance. For more information, see Set a password.
3. Install MongoDB on a server. For more information, see Install MongoDB.

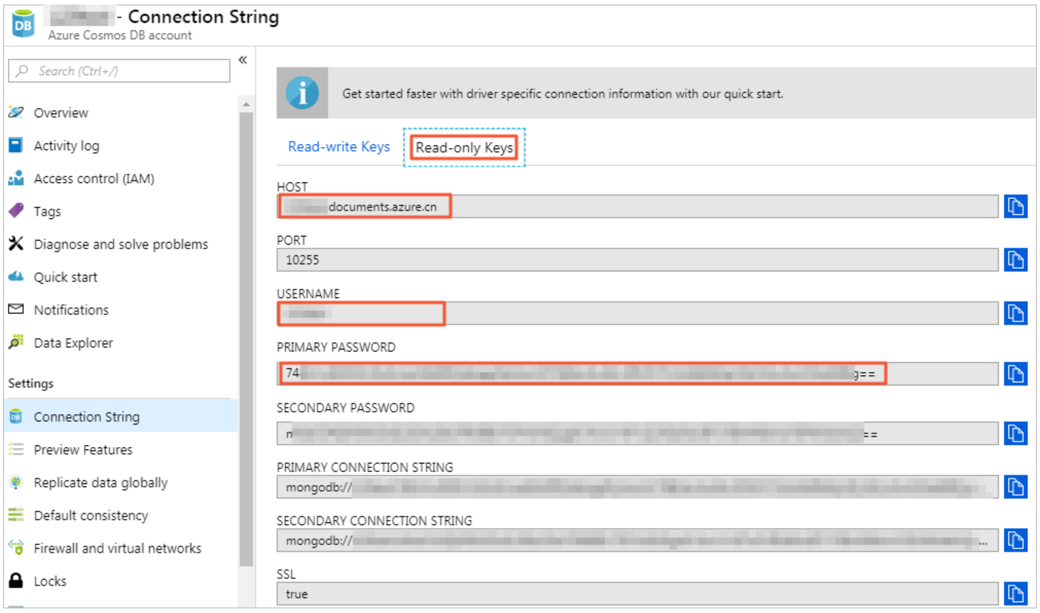
   > Note
   > - Install a MongoDB version later than 3.0.
   > - This server is used to temporarily store data during backup and recovery, and is not needed after the migration is complete.
   > - The capacity of the disk where the backup is stored must be larger than Azure Cosmos DB.

   This example installs MongoDB on a Linux server. You can also use other operating systems, such as Windows.

Procedure

1. Log on to the Azure portal.
2. In the left-side navigation pane, click **Azure Cosmos DB**.
3. On the **Azure Cosmos DB** page, click the account name of the Azure Cosmos DB that you want to migrate.
4. On the account details page, click **Connection String**.
5. Click the **Read-only Keys** tab to view the database connection information.

   Azure connection information



Get started faster with driver specific connection information with our quick start.

HOST: [redacted]documents.azure.cn

PORT: 10255

USERNAME: [redacted]

PRIMARY PASSWORD: 74[redacted]g==

SECONDARY PASSWORD: n[redacted]==

PRIMARY CONNECTION STRING: mongodb://[redacted]

SECONDARY CONNECTION STRING: mongodb://[redacted]...

SSL: true

> ⓘ **Note** To migrate data, you only need a database account that has read-only permissions.

6. Run the following command on the MongoDB server to back up the Azure Cosmos DB to this server.

```
mongodump --host <HOST>:10255 --authenticationDatabase admin -u <USERNAME> -p <PRIMARY
PASSWORD> --ssl --sslAllowInvalidCertificates
```

   Note: Replace <HOST>, <USERNAME>, and <SECONDARY PASSWORD> with the actual values shown in the Azure connection information figure.

   After the backup is complete, backups of the Azure Cosmos DB are stored in the dump folder.

7. Obtain the endpoint of the primary node of the ApsaraDB for MongoDB instance. For more information, see Overview of replica set instance connections.
8. Run the following command on the MongoDB server to export the backups to the ApsaraDB for MongoDB instance.

```
 mongorestore --host <mongodb_host>:3717 --authenticationDatabase admin -u <username> -
p <password> dump
```

   Parameter description:

   - <mongodb_host>: the endpoint of the primary node of the MongoDB instance.

- <username>: the account used to log on to the ApsaraDB for MongoDB instance.
- <password>: the password used to log on to the ApsaraDB for MongoDB instance.

After the recovery is complete, backups of the Azure Cosmos DB are migrated to the ApsaraDB for MongoDB instance.

5.Manage the ApsaraDB for MongoDB balancer

ApsaraDB for MongoDB allows you to manage the balancer. You can enable or disable the balancer and set an active time window for the balancer to meet specific business needs.

Precautions

- The balancer is a feature of the sharded cluster architecture and applies only to sharded cluster instances.
- Operations related to the balancer may occupy the resources of an instance. Therefore, we recommend that you perform these operations during off-peak hours.

Set an active time window for the balancer

The balancer consumes resources of nodes in an instance to migrate chunks, which may cause imbalanced resource usage on the nodes and affect business operations. To avoid negative impact on your business while the balancer is migrating chunks, you can set an active time window to allow the balancer to migrate chunks only within the specified period of time.

> **Note** Before performing this operation, make sure that the balancer is enabled. For more information about how to enable the balancer, see Enable the balancer.

1. Connect to a sharded cluster instance by using the mongo shell.
2. After connecting to a mongos node, run the following command in the mongo shell to switch to the config database:

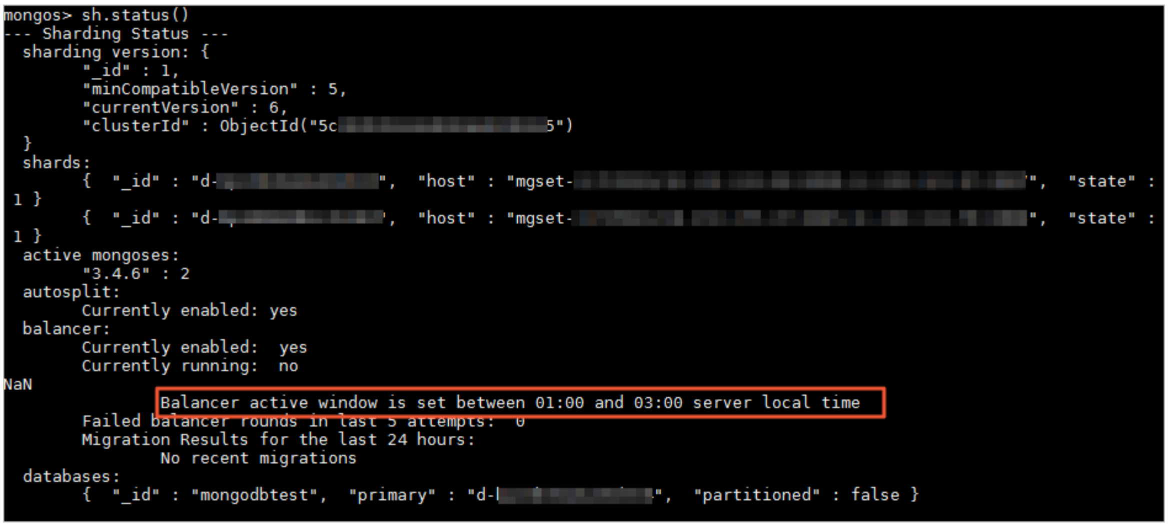
   ```
   use config
   ```

3. Run the following command to set an active time window for the balancer:

   ```
   db.settings.update(
      { _id: "balancer" },
      { $set: { activeWindow : { start : "<start-time>", stop : "<stop-time>" } } },
      { upsert: true }
   )
   ```

   > **Note**
   >
   > - <start-time>: the beginning of the time range. Specify the time in the HH:MM format (the local time of the specified region). The valid values for HH: 00 to 23. The valid values for MM: 00 to 59.
   > - <stop-time>: the end time of the time range. Specify the time in the HH:MM format (the local time of the specified region). The valid values for HH: 00 to 23. The valid values for MM: 00 to 59.

You can run the `sh.status()` command to check the active time window of the balancer. For example, the following command output shows that the active time window of the balancer is 01:00 to 03:00.



```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5c[redacted]5")
  }
  shards:
        {  "_id" : "d-[redacted]",  "host" : "mgset-[redacted]",  "state" :
 1 }
        {  "_id" : "d-[redacted]",  "host" : "mgset-[redacted]",  "state" :
 1 }
  active mongoses:
        "3.4.6" : 2
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
NaN
                Balancer active window is set between 01:00 and 03:00 server local time
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "mongodbtest",  "primary" : "d-[redacted]",  "partitioned" : false }
```

You can also perform other related operations. For example, to ensure that the balancer is always running, you can run the following command to clear the active time window settings:

```
db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

Enable the balancer

If you have configured sharding, the balancer can immediately start a balancing procedure among shards after being enabled. Balancing occupies the resources of an instance. Therefore, we recommend that you perform this operation during off-peak hours.

1. Connect to a sharded cluster instance by using the mongo shell.
2. After connecting to a mongos node, run the following command in the mongo shell to switch to the config database:

   ```
   use config
   ```

3. Run the following command to enable the balancer:

   ```
   sh.setBalancerState(true)
   ```

Disable the balancer

ApsaraDB for MongoDB enables the balancer by default. To disable the balancer in special business scenarios, follow these steps:

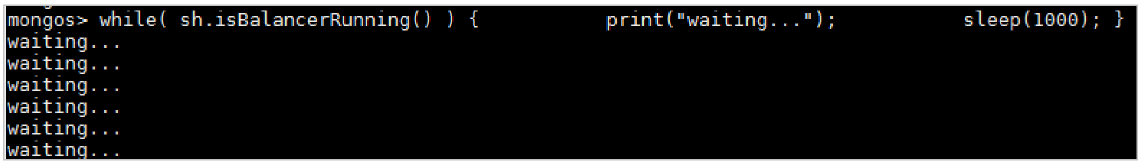
1. Connect to a sharded cluster instance by using the mongo shell.
2. After connecting to a mongos node, run the following command in the mongo shell to switch to the config database:

   ```
   use config
   ```

3. Run the following command to check the running status of the balancer:

```
while( sh.isBalancerRunning() ) {
          print("waiting...");
          sleep(1000);
}
```

- If the command does not return any values, the balancer is not running any tasks. You can proceed to disable the balancer.
- If the command returns `waiting` , the balancer is migrating chunks. In this case, you cannot disable the balancer. Otherwise, data may become inconsistent.



```
mongos> while( sh.isBalancerRunning() ) {          print("waiting...");          sleep(1000); }
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
```

4. After confirming that the balancer is not running any tasks, run the following command to disable the balancer:

```
sh.stopBalancer()
```

6.Redirect read requests to data images while data is being updated

ApsaraDB for MongoDB allows you to create read-only data images in a replica set instance or a sharded cluster instance. A data image in a replica set instance can store up to 3 TB of data and that in a sharded cluster instance can store up to 96 TB of data.

How data images work

After you create data images for an instance in which a large amount of data is being updated, all read requests from applications are redirected to read data from the data images. The applications do not read data from the instance until data is completely updated in the instance. The applications read data from the data images as if they were reading data from the instance, without any compromise to the read performance.

> **Note** After data is updated in the instance, you can synchronize the updated data to the data images so that the applications can read the latest data from the data images later. You can use the commands provided by Alibaba Cloud to enable automatic data synchronization within seconds. Data synchronization does not affect normal read operations.

Create data images in a replica set instance

1. Use the mongo shell to connect to the primary node or a secondary node where you want to create a data image. For more information, see Connect to a replica set instance by using the mongo shell.
2. Run the following command to create a data image:

   ```
   db.runCommand({checkpoint:"create"})
   ```

3. Run the following command to delete the data image when you no longer need it:

   ```
   db.runCommand({checkpoint:"drop"})
   ```

Create data images in a sharded cluster instance

1. Use the mongo shell to connect to any mongos node in the sharded cluster instance where you want to create data images. For more information, see Connect to a sharded cluster instance by using the mongo shell.
2. Create data images.
   - To create a data image on the primary node of each shard, run the following command:

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"create"}})
     ```

   - To create a data image on a secondary node of each shard, run the following command:

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"create"}, $queryOptions: {$readPreference: {mode: 'secondary'}}})
     ```

3. Delete the data images when you no longer need them.
   - To delete the data image on the primary node of each shard, run the following command:

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"drop"}})
     ```

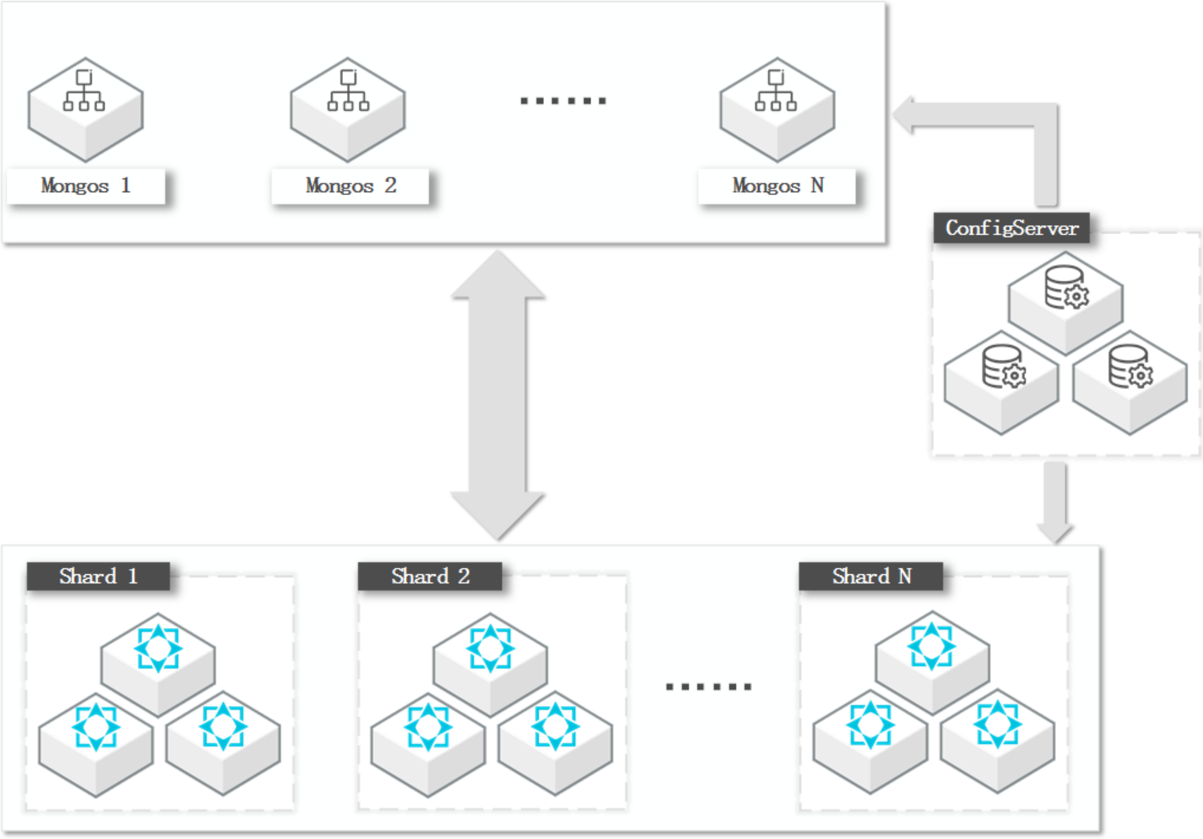
   - To delete the data image on a secondary node of each shard, run the following command:

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"drop"}, $queryOptions: {$readPreference: {mode: 'secondary'}}})
     ```

7.Use a connection string URI to connect to a sharded cluster instance

An ApsaraDB for MongoDB sharded cluster instance provides the connection string for each mongos node. You can access the databases of the sharded cluster instance after connecting to a mongos node. However, you must use a correct method to connect to a sharded cluster instance to implement load balancing and high availability.

Background information



A sharded cluster instance distributes and stores data on multiple shards to facilitate high scalability. When creating a sharded cluster instance, ApsaraDB for MongoDB uses a Configserver to store the metadata of the cluster and uses one or more mongos nodes to provide the entrance to the cluster for applications. The mongos nodes read routing information from the Configserver to route requests to corresponding shards at the backend.

- When you connect to a mongos node, the mongos node can function as a mongod process.
- All mongos nodes are equal. You can connect to one or more mongos nodes to access a sharded cluster instance.
- Mongos nodes are stateless and can be scaled out as required. The service capability of a sharded cluster instance is subject to the smaller one between the total service capability of shards and that of mongos nodes.
- When you access a sharded cluster instance, we recommend that you share the load of applications

evenly among multiple mongos nodes.

Connection string URIs

To correctly connect to a sharded cluster instance, you must understand the format of connection string URIs of MongoDB. All official MongoDB drivers allow you to use a connection string URI to connect to MongoDB.

Example:

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

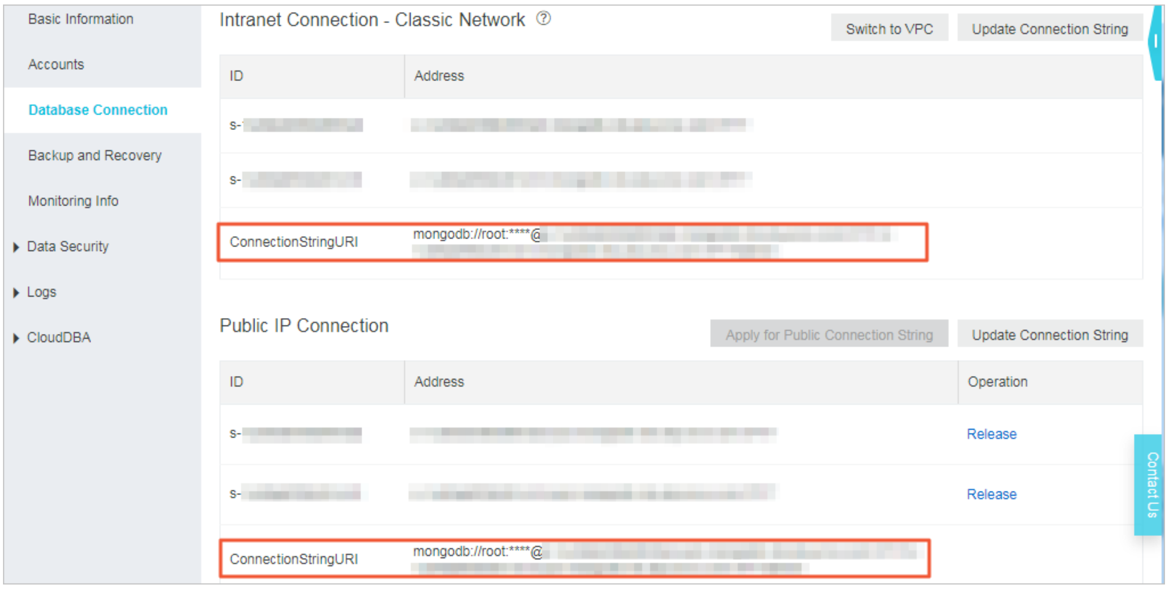
Note

- `mongodb://` : the prefix, indicating a connection string URI.
- `username:password@` : the username and password used to log on to the database.
- `hostX:portX` : the list of connection string used to connect to mongos nodes.
- `/database` : the database corresponding to the username and password if authentication is enabled.
- `?options` : additional connection options.

Use a connection string URI to connect to a sharded cluster instance

You can use a connection string URI to connect to a sharded cluster instance to implement load balancing and high availability.

1. Obtain the connection string URI of a sharded cluster instance. For more information, see Connect to a sharded cluster instance.



ConnectionStringURI mongodb://root:****@

ConnectionStringURI mongodb://root:****@

2. Use the obtained connection string URI to connect your applications to the instance. For more information, see Connect to an ApsaraDB for MongoDB instance by using program code.

   Example of the Java code:

```
MongoClientURI connectionString = new MongoClientURI("mongodb://:****@s-xxxxxxxx.mongodb.rds.aliyuncs.com:3717,s-xxxxxxxx.mongodb.rds.aliyuncs.com:3717/admin"); //Replace **** with the password of the root user.
MongoClient client = new MongoClient(connectionString);
MongoDatabase database = client.getDatabase("mydb");
MongoCollection<Document> collection = database.getCollection("mycoll");
```

> **Note**
>
> After you use the preceding method to connect to a sharded cluster instance, a client can automatically distribute requests to multiple mongos nodes to balance the load. If you have used a connection string URI to connect to two or more mongos nodes and a mongos node is faulty, the client can automatically skip this faulty node and distribute requests to the other normal mongos nodes.

If a large number of mongos nodes are used, you can group them by application. For example, you have application A, application B, and four mongos nodes. You can specify only the connection strings of mongos 1 and mongos 2 in the URI for application A, and specify only the connection strings of mongos 3 and mongos 4 in the URI for application B. In this way, you can isolate mongos nodes to provide separate entrance for different applications.

> **Note** Although applications are connected to mutually isolated mongos nodes, they share shards at the backend.

Common connection options

- Implement read/write splitting

  Add `readPreference=secondaryPreferred` in the options parameter to set read preference to secondary nodes.

  Example:

  ```
  mongodb://root:xxxxxxxx@dds-xxxxxxxxxxxx:3717,xxxxxxxxxxxx:3717/admin? replicaSet=mgset-xxxxxx&readPreference=secondaryPreferred
  ```

- Limit connections

  Add `maxPoolSize=xx` in the options parameter to limit the maximum number of connections in the connection pool of a client to xx.

- Send an acknowledgment after data has been written to the majority of nodes

  Add `w= majority` in the options parameter to ensure that ApsaraDB for MongoDB sends an acknowledgment to a client after writing data to the majority of nodes for a write request.

8.Use MongoDB to store logs

An online service generates a large number of operational logs and access logs, which contain information about errors that occurred, triggered alerts, and user behaviors. Generally, such logs are stored in text files, which are readable and can be used to quickly locate issues in routine O&M. However, after a service generates a large number of logs, it is necessary to store and analyze the logs in a more advanced way to explore the value of the log data.

This topic takes the access logs of a web service as an example to describe how to use MongoDB to store and analyze logs to make full use of the log data. The methods and operations described in this topic also apply to other types of log storage services.

Access logs of a web server

The following example shows an access log of a web server. Typically, an access log contains information about the IP address used for the access, the user who accessed the target resource, the operating system and browser that the user used for the access, the endpoint of the target resource, and the result of the access.

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"
```

You can use MongoDB to store each access log in a single document in the following format:

```
{
            _id: ObjectId('4f442120eb03305789000000'),
            line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
            }
```

The preceding method is easy to configure. However, it may bring inconvenience to the analysis of log data. As MongoDB is not a service targeted at text analysis, we recommend that you convert the format of a log to extract each field and its value from the log before you store it in MongoDB as a document. As shown in the following document, the preceding log is converted to separate fields and values:

```
{
            _id: ObjectId('4f442120eb03305789000000'),
            host: "127.0.0.1",
            logname: null,
            user: 'frank',
            time: ISODate("2000-10-10T20:55:36Z"),
            path: "/apache_pb.gif",
            request: "GET /apache_pb.gif HTTP/1.0",
            status: 200,
            response_size: 2326,
            referrer: "[http://www.example.com/start.html](http://www.example.com/start.htm
l)",
            user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
            }
```

When you convert the format of a log, you can also remove the fields that you regard as useless to the data analysis to save storage space. Several irrelevant fields in the preceding document need to be removed, including the user, request, and status fields. You can also remove the time field because the _id field contains the information about the time when the access was performed. However, you can retain the time field for later analysis because it demonstrates the information in a more clear way and a query statement that uses the time field is more user-friendly. In addition, compared with the _id field, the data type of the time field requires less storage space. Based on the preceding reasons, the following updated content may be eventually stored in the document:

```
{
            _id: ObjectId('4f442120eb03305789000000'),
            host: "127.0.0.1",
            time: ISODate("2000-10-10T20:55:36Z"),
            path: "/apache_pb.gif",
            referer: "[http://www.example.com/start.html](http://www.example.com/start.html
)",
            user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
            }
```

Write logs to MongoDB

A log storage service is required to collect a large number of logs at a time. To meet such a requirement, you can specify a write concern for MongoDB to manage the write operation. For example, you can specify the following write concern:

```
db.events.insert({
                host: "127.0.0.1",
                time: ISODate("2000-10-10T20:55:36Z"),
                path: "/apache_pb.gif",
                referer: "[http://www.example.com/start.html](http://www.example.com/start.
html)",
                user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
                }
                )
```

Note

- If you require the highest write throughput, you can set the w option of the write concern to {w: 0}.
- If the target log data is of great importance, for example, the log data is used as the credentials of service billing, you can set the w option to {w: 1} or {w: "majority"}, which is more secure compared with {w: 0}.

To improve the efficiency of the write operation, you can write multiple logs to MongoDB with a single request. The request is in the following format:

```
db.events.insert([doc1, doc2, ...])
```

Query logs in MongoDB

After logs are stored in MongoDB by using the preceding method, you can query logs in MongoDB based on different query requirements.

- Query the logs of all requests to access /apache_pb.gif.

  ```
  q_events = db.events.find({'path': '/apache_pb.gif'})
  ```

  Note If you need to frequently query such access logs, you can create an index on the path field to improve query efficiency, for example, `db.events.createIndex({path: 1})` .

- Query the logs of all requests within a day.

  ```
  q_events = db.events.find({'time': { '$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': IS
  ODate("2016-12-20T00:00:00.00Z")}})
  ```

  Note You can create an index on the time field to improve query efficiency, for example, `db.events.createIndex({time: 1})` .

- Query the logs of all requests that are sent to a server over a period of time.

  ```
  q_events = db.events.find({
                  'host': '127.0.0.1',
                  'time': {'$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': ISODate("2016-
  12-20T00:00:00.00Z" }
                  })
  ```

  Similarly, you can use the aggregation pipeline or perform map-reduce operations provided by MongoDB to initiate more complex queries for data analysis. We recommend that you create indexes on fields properly to improve query efficiency.

Data sharding

When the number of service nodes that generate logs increases, the requirements for the write and storage capabilities of a log storage service also increase. In this case, you can use the sharding feature provided by MongoDB to distribute the log data across multiple shards. When you use the sharding feature, you need to focus on choosing shard keys.

- Use a field that indicates the timestamp as the shard key: For example, use the _id field that contains

ObjectId values or the time field as the shard key. However, the following issues may occur in this type of sharding:

  - As the timestamp grows in sequence, newly collected log data will be distributed to the same shard. Hence, the write capability of MongoDB is not enhanced.
  - Many log queries target at the latest log data, which is distributed to only a few shards. Hence, only statistics related to these shards are returned for these queries.

- Use the hashed sharding method: The default shard key of hashed sharding is set to the _id field. This sharding method evenly distributes log data to each shard. Therefore, the write capability of MongoDB grows with shards in a linear manner. However, hashed sharding distributes data randomly. This leads to the issue where MongoDB cannot efficiently process the requests of given ranged queries, which are often used in data analysis. To process such a request, MongoDB needs to traverse all the shards and merge the queried data to return the final result.
- Use the ranged sharding method: Assume that values of the path field in the preceding example are evenly distributed and many queries are based on the path field. Then, you can specify the path field as the shard key to divide data into contiguous ranges. This method has the following benefits:

  - Write requests are evenly distributed to each shard.
  - Query requests based on the path field are densely distributed to one or more shards, which improves the query efficiency.

  In addition, the following issues may occur:

  - If a value of the path field is frequently accessed, logs with the same shard key value are likely to be in the same chunk or shard. The value is at high frequency and the size of the chunk may be large.
  - If the path field has few values, access logs cannot be properly distributed to each shard.

  To fix these issues, you can pass an additional field to the shard key. Assume that the original shard key value is {path: 1}. Then, you can add the ssk field to the shard key, for example, `{path: 1, ssk: 1}` .

  You can assign a random value to the `ssk` field, such as the hash value of the _id field. You can also assign a timestamp to the ssk field so that the shard key values with the same path value are sorted by time.

  In this way, the shard key has multiple values with even frequency. No shard key value is at an extremely high frequency. Each of the preceding sharding methods has its own advantages and disadvantages. You can select a method based on your business requirements.

Solutions to data growth

MongoDB provides the sharding feature for you to store massive data. However, the storage costs increase with the growth of data volume. Typically, the value of log data decreases over time. Data generated one year ago or even three months ago, which is valueless to the analysis, needs to be cleared to reduce storage costs. MongoDB allows you to use the following solutions to meet such a requirement.

- Use TTL indexes: Time to live (TTL) indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. In the preceding example, the time field indicates the time when the request was sent. You can run the following command to create a TTL index on the time field and specify that MongoDB removes the document after 30 hours: `db.events.createIndex( { time: 1 }, { expireAfterSeconds: 108000 } )` .

> **Note** With a TTL index created, the background task that removes expired documents written in single-threading mode runs every 60 seconds by default. If a large amount of log data is written to MongoDB, many documents in MongoDB are about to expire over time. The expired documents that are not removed occupy large storage space.

- Use capped collections: If you do not have strict limits on the storage period, whereas you want to limit the storage space, you can use capped collections to store log data. Capped collections work in the following way: After you specify a maximum storage space or a maximum number of stored documents for a capped collection, once one of the limits is reached, MongoDB automatically removes the oldest documents in the collection. For example, you can run the following command to create and configure a capped collection: `db.createCollection("event", {capped: true, size: 104857600000}` .
- Archive documents by collection or database periodically: At the end of a month, you can rename the collection that stores the documents of that month and create another collection to store documents of the next month. We recommend that you append the information about the year and month to the collection name. The following example describes how the 12 collections that store the documents written in each month of 2016 are named:

  ```
  events-201601
                  events-201602
                  events-201603
                  events-201604
                  ....
                  events-201612
  ```

  If you need to clear the documents of a specific month, you can run the following command to directly delete the corresponding collection:

  ```
   db["events-201601"].drop()
                  db["events-201602"].drop()
  ```

  The query statements may be complex if you need to query the documents of multiple months as MongoDB needs to merge the queried data of multiple collections to return the final result.

9.Introduction to ApsaraDB for MongoDB sharded cluster instances

This topic introduces ApsaraDB for MongoDB sharded cluster instances. You can use sharded cluster instances to store large amounts of data.

Scenarios

If you encounter one of the following issues, you can use sharded cluster instances:

- The storage capacity of a single physical host is limited.
- The read and write capabilities of a single physical host are limited due to insufficient CPU, memory, or network interface controller (NIC) resources.

Number of shard nodes and number of mongos nodes

You can specify the number of shard nodes and number of mongos nodes in a sharded cluster instance based on your business requirements.

- The sharded cluster instance is used only to store a large amount of data, which is not frequently accessed. For example, if the size of the data that a single shard node can store is M and the total size of the data that needs to be stored is N, calculate the number of shard nodes and the number of mongos nodes by using the following formulas:
  - Number of shard nodes = N/M/0.75. The water mark for storage usage is 75%.
  - Number of mongos nodes = 2 or more. When the data is not frequently accessed, at least two mongos nodes must be deployed to ensure high availability.
- The sharded cluster instance is used to store a small amount of data, which is accessed by highly concurrent read or write operations. The shard nodes and mongos nodes of the sharded cluster instance must deliver the required read and write performance. For example, the maximum queries per second (QPS) of a single shard node is M, the maximum QPS of a single mongos node is Ms, and the QPS that is required for your business is Q. In this case, calculate the number of shard nodes and the number of mongos nodes by using the following formulas:
  - Number of shard nodes = Q/M/0.75 (The water mark for storage usage is 75%.)
  - Number of mongos nodes = Q/Ms/0.75

> ? **Note** The QPS of a single mongos or mongod node in MongoDB varies based on the features that are enabled. To obtain the actual QPS, you must perform tests.

Note

- If you want a sharded cluster instance to meet the requirements in both the preceding scenarios, you must estimate the number of shard nodes and number of mongos nodes in the instance based on higher specification requirements.
- The preceding formulas are used to estimate the number of shard nodes and number of mongos nodes in a sharded cluster instance based on the assumption that the data and requests in the instance are evenly distributed. In actual scenarios, data and requests in the instance may be unevenly distributed. To balance the loads in the instance, you must select an appropriate shard key for each shard node.

Shard key selection

- ApsaraDB for MongoDB supports the following sharding strategies:
  - Ranged sharding, which is used to divide data into contiguous ranges that are determined by shard key values
  - Hashed sharding, which is used to distribute data among the configured shard nodes
  - Tag aware sharding, which is used to customize the rules based on which chunks are distributed

  Note

  - How sharding strategies work

    a. ApsaraDB for MongoDB invokes the `sh.addShardTag()` method to add Tag A to a shard node.

    b. ApsaraDB for MongoDB invokes the `sh.addTagRange()` method to add Tag A to a specific chunk range of a collection. This way, ApsaraDB for MongoDB can distribute the data within the chunk range labeled with Tag A to the shard node labeled with Tag A. ApsaraDB for MongoDB can also distribute the data within the supersets of the chunk range labeled with Tag A to the shard node labeled with Tag A.

  - Scenarios

    - Add tags to shard nodes based on the data centers in which these shard nodes are deployed. This way, ApsaraDB for MongoDB can distribute the data within a chunk range to the data center that is has the same tag as the chunk range.
    - Add tags to shard nodes based on the QPS values of these shard nodes. This way, ApsaraDB for MongoDB can distribute more chunks to the shard nodes that deliver high QPS.

  - Usage notes

    ApsaraDB for MongoDB cannot directly distribute chunks to shard nodes that are labeled with the specified tags. Chunks are gradually distributed as chunk splitting and migration are frequently triggered by insert and update operations. Make sure that the balancer is enabled during the distribution process. After tags are added to chunk ranges, written data may not be distributed to the shard node that has the same tags as the chunk ranges.

  For more information, see Tag aware sharding.

- Ranged sharding and hashed sharding cannot resolve the following issues:
    - The value range of the shard key is small. For example, if data centers are selected as the shard key, sharding is inefficient because the number of data centers is limited.
    - A specific value of the shard key is contained in a large number of documents. In this case, a single chunk may store a large number of documents. If the size of documents in a chunk exceeds the size per chunk, the chunk is labeled as a jumbo chunk and cannot be migrated by the balancer.
    - If you query and update data based on criteria rather than the shard key, all the operations become scatter-gather queries, which are slow.
- Sharding is efficient when the shard key has the following characteristics:
    - Sufficient cardinality
    - Evenly distributed write operations
    - Targeted read operations

Example:

An IoT application uses a sharded cluster instance to store the logs of millions of devices. Each device sends logs to the sharded cluster instance once every 10 seconds. The logs contain information such as device IDs and timestamps. The logs that are generated for a specific device over a specific time range are frequently queried.

The following sharding methods are provided:

- Method 1: Recommended. Use the combination of the device ID and the timestamp as the shard key and perform ranged sharding.
    - Written data is evenly distributed among multiple shard nodes.
    - The data that carries the same device ID can be further split and distributed among multiple chunks based on the timestamps of the data.
    - When you query the logs that are generated for a specific device over a specific time range, ApsaraDB for MongoDB can combine the device ID and the timestamp to create a composite index based on which it completes the query.
- Method 2: Use the timestamp as the shard key and perform ranged sharding.
    - Written data that carries continuous timestamps is distributed to the same shard node. This causes uneven distribution of written data.
    - Device ID-based queries are distributed to all shard nodes. This lowers query efficiency.
- Method 3: Use the timestamp as the shard key and perform hashed sharding.
    - Written data is evenly distributed among multiple shard nodes.
    - Device ID-based queries are distributed to all shard nodes. This lowers query efficiency.
- Method 4: Use the device ID as the shard key and perform hashed sharding.

    > ⓘ **Note** If no rules are imposed on the device ID, you can perform ranged sharding.

    - Written data is evenly distributed among multiple shard nodes.
    - The data that carries the same device ID can be distributed only to the same chunk. This causes jumbo chunks. Each device ID-based query can be distributed only to a single shard node, and the shard node needs to scan and sort all tables based on the timestamp range that is specified in the query.

Jumbo chunk and size per chunk

The default size per chunk in a sharded cluster instance is 64 MB. If the size of a chunk exceeds 64 MB and the chunk cannot be split, the chunk is labeled as a jumbo chunk. For example, if all documents have the same shard key value, these documents are stored in the same chunk, which cannot be split. The balancer cannot migrate jumbo chunks, which may cause load imbalance. We recommend that you prevent jumbo chunks.

If you do not require load balancing, you do not need to prevent jumbo chunks because jumbo chunks do not affect read or write operations. You can use one of the following methods to handle jumbo chunks:

- Split jumbo chunks. After a jumbo chunk is split, the configured mongos nodes automatically remove the jumbo label from the chunk.
- If a chunk cannot be split, verify that the chunk is not a jumbo chunk and manually remove the jumbo label from the chunk.

  > **Note** Before you remove the jumbo label from a chunk, you must back up the database named config to prevent data corruption that is caused by unintended operations.

- Increase the size per chunk. When the size of a jumbo chunk no longer exceeds the size per chunk, the jumbo label is automatically removed from the chunk. However, some chunks may still become jumbo chunks as data is written. The best way to prevent jumbo chunks is to select an appropriate shard key.

You need to adjust the size per chunk in the following scenarios. Make sure that the size per chunk ranges from 1 MB to 1,024 MB.

- If the I/O load is high during chunk migration, you can reduce the size per chunk.
- When you perform tests to verify sharding efficiency, you can reduce the size per chunk.
- If the initial size per chunk is large and consequently the loads are imbalanced due to a large number of jumbo chunks, you can increase the size per chunk.
- If you want to shard a collection that stores terabytes of data, you must increase the size per chunk. For more information, see Sharding Existing Collection Data Size.

Balancer

Automatic load balancing in a sharded cluster instance is implemented by a background thread that runs on the mongos nodes of the instance. Only one migration task can be run for each collection at a specific point in time. Load balancing is triggered based on the number of chunks per collection on each shard node. If the difference between the number of chunks of a collection on a shard node and the total number of chunks of the collection reaches the specified threshold, ApsaraDB for MongoDB starts to migrate the chunks of the collection from the shard node to other shard nodes. You must specify a threshold based on the total number of chunks.

By default, the balancer is enabled. To prevent interruptions to your online workloads due to the migration of chunks in a sharded cluster instance, you can configure the instance to perform the migration during off-peak hours, such as 02:00 to 06:00.

```
use config
                db.settings.update(
                { _id: "balancer" },
                { $set: { activeWindow : { start : "02:00", stop : "06:00" } } },
                { upsert: true }
                )
```

Notice When you back up a sharded cluster instance by using the configured mongos nodes or when you separately back up the Configserver and each configured shard node, you must run the following command to stop the balancer. This allows you to prevent data inconsistencies in the backup data.

```
sh.stopBalancer()
```

Archiving setting in the moveChunk command

If a sharded cluster instance runs MongoDB 3.0 or earlier, the disk space usage in the data catalog may continue to increase even after data writes are stopped.

The preceding issue is caused by the value of the `sharding.archiveMovedChunks` parameter in the moveChunk command. In MongoDB 3.0 or earlier, the default value of this parameter is `true`. The value true indicates that chunks are archived on Shard A after they are migrated from Shard A to Shard B. If the chunks on Shard B are damaged, you can restore the chunks from Shard A. The chunks occupy storage space on both Shard A and Shard B. That is why the disk space usage in the data catalog continues to increase even after data writes are stopped.

Note In MongoDB 3.2, the default value of this parameter is `false`. The value false indicates that chunks are not archived on Shard A after they are migrated from Shard A to Shard B.

recoverShardingState parameter

When you use a sharded cluster instance, you may encounter the following issues:

The configured shard nodes do not run as expected after they are started. When the db.isMaster() method is invoked on the primary node of a shard node, the value true is returned. In addition, the other commands cannot be run. The following code snippet shows an example:

```
mongo-9003:PRIMARY> db.isMaster()
                {
                "hosts" : [
                "host1:9003",
                "host2:9003",
                "host3:9003"
                ],
                "setName" : "mongo-9003",
                "setVersion" : 9,
                "ismaster" : false,  // Indicates that the node is not a primary node.
                "secondary" : true,
                "primary" : "host1:9003",
                "me" : "host1:9003",
                "electionId" : ObjectId("57c7e62d218e9216c70a****"),
                "maxBsonObjectSize" : 16777216,
                "maxMessageSizeBytes" : 48000000,
                "maxWriteBatchSize" : 1000,
                "localTime" : ISODate("2016-09-01T12:29:27.113Z"),
                "maxWireVersion" : 4,
                "minWireVersion" : 0,
                "ok" : 1
                }
```

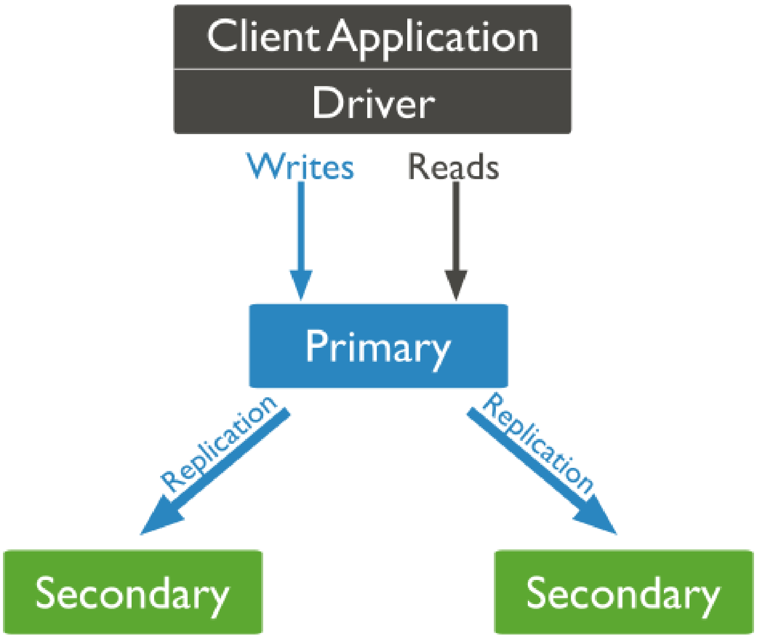
The error logs indicate that the shard node cannot connect to the Configserver because the `sharding.recoverShardingState` parameter is set to the default value true. When the shard node is started, it attempts to connect to the Configserver and initialize the sharding settings. If the shard node cannot connect to the Configserver, the sharding settings cannot be initialized. As a result, the status of the shard node is abnormal.

When you migrate all the shard nodes of a sharded cluster instance to a new physical host, the information about the Configserver changes, but the shard node still attempts to connect to the Configserver by using the original information about the Configserver. In This case, the shard node cannot connect to the Configserver. To resolve this problem, add `setParameter recoverShardingState=false` in the command that is used to start the shard node.

10.How a replica set in MongoDB works

A replica set in ApsaraDB for MongoDB is a group of mongod processes and contains a primary node and multiple secondary nodes. MongoDB drivers write data to the primary node only. Then, data is synchronized from the primary node to secondary nodes. This ensures data consistency across all nodes in the replica set. Therefore, replica sets provide high availability.

The following figure is extracted from the official documentation of MongoDB. It shows a typical MongoDB replica set that contains one primary node and two secondary nodes.



Primary node election (1)

A replica set is initialized by running the `replSetInitiate` command or running the `rs.initiate()` command in the mongo shell. After the replica set is initialized, the members send heartbeat messages to each other and initiate the primary node election. The node that receives votes from a majority of the members becomes the primary node and the other nodes become secondary nodes.

Initialize the replica set

```
config = {
    _id : "my_replica_set",
    members : [
         {_id : 0, host : "rs1.example.net:27017"},
         {_id : 1, host : "rs2.example.net:27017"},
         {_id : 2, host : "rs3.example.net:27017"},
   ]
}
rs.initiate(config)
```

Definition of majority

A group of voting members is considered a majority only if it contains members of more than the average of the total number of members. If the number of members in a replica set is equal to or less than the average number of all voting members, the election cannot be implemented. In this case, you cannot write data to the replica set.

Number of voting members	Majority	Maximum number of failed nodes
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

We recommend that you set the number of members in a replica set to an odd number. The preceding table shows that both a replica set with three nodes and a replica set with four nodes tolerate the failure of one node. The two replica sets provide the same service availability. However, the replica set with four nodes provides more reliable data storage.

Special secondary nodes

Secondary nodes of a replica set participate in the primary node election. A secondary node may also be elected as the primary node. The latest data written to the primary node is synchronized to secondary nodes to ensure data consistency across all nodes.

You can read data from secondary nodes. Therefore, you can add secondary nodes to a replica set to improve the read performance and service availability of the replica set. ApsaraDB for MongoDB allows you to configure secondary nodes of a replica set to meet the requirements of different scenarios.

- Arbiter

  An arbiter node participates in the election as a voter only. It cannot be elected as the primary node or synchronize data from the primary node.

  Assume that you deploy a replica set that contains one primary node and one secondary node. If one node fails, the primary node election cannot be implemented. As a result, the replica set becomes unavailable. In this case, you can add an arbiter node to the replica set to enable primary node election.

  An arbiter node is a lightweight node that does not store data. If the number of members in a replica set is an even number, we recommend that you add an arbiter node to increase the availability of the replica set.

- Priority0

  A node that has priority 0 in the primary node election cannot be elected as the primary node.

  Assume that you deploy a replica set that contains nodes in both Data center A and Data center B. To ensure that the elected primary node is deployed in Data center A, set the priorities of the replica set members in Data center B to 0.

> **Note** If you set the priorities of the members in Data center B to 0, we recommend that you deploy a majority of replica set nodes in Data center A. Otherwise, the primary node election may fail during network partitioning.

- Vote0

  In MongoDB 3.0, a replica set contains a maximum of 50 members, and up to seven members can vote in a primary node election. You must set the members[n].votes attribute to 0 for members that are not expected to vote.

- Hidden

  A hidden member in a replica set cannot be elected as the primary node because its priority is 0. Hidden nodes are invisible to MongoDB drivers.

  You can use hidden nodes to back up data or perform offline computation tasks. This does not affect the services of the replica set because hidden nodes do not process requests from MongoDB drivers.

- Delayed

  A delayed node must be a hidden node. Data on a delayed node reflects an earlier state of the data on the primary node. If you configure a one-hour delay, data on the delayed node is the same as the data on the primary node an hour ago.

  Therefore, if you write incorrect or invalid data to the primary node, you can use the data on the delayed node to restore the data on the primary node to an earlier state.

Primary node election (2)

A primary node election is triggered not only after replica set initialization but also in the following scenarios:

- Replica set reconfiguration

  A primary node election is triggered if the primary node fails or voluntarily steps down and becomes a secondary node. A primary node election is affected by various factors, such as heartbeat messages among nodes, priorities of nodes, and the time when the last oplog entry was generated.

  - Node priorities

    All nodes tend to vote for the node that has the highest priority. A priority 0 node cannot trigger primary node elections. If a secondary node has a higher priority than the primary node and the time difference between the latest log entry of the secondary node and that of the primary node is within 10 seconds, the primary node steps down. In this case, this secondary node becomes a candidate for the primary node.

  - Optime

    Only secondary nodes that have the latest oplog entry are eligible to be elected as the primary node.

- Network partitioning

Only nodes that are connected to a majority of voting nodes can be elected as the primary node. If the primary node is disconnected from a majority of the other nodes in the replica set, the primary node voluntarily steps down and becomes a secondary node. A replica set may have multiple primary nodes for a short period of time during network partitioning. When MongoDB drivers write data, we recommend that you set a policy that allows data synchronization from only the primary node that is connected to a majority of nodes.

Data synchronization

Data is synchronized from the primary node to secondary nodes based on the oplog. After a write operation on the primary node is complete, an oplog entry is written to the special local.oplog.rs collection. Secondary nodes constantly import new oplog entries from the primary node and apply the operations.

To prevent unlimited growth in the size of the oplog, local.oplog.rs is configured as a capped collection. When the amount of oplog data reaches the specified threshold, the earliest entries are deleted. All operations in the oplog must be idempotent. This ensures that an operation produces the same results regardless of whether it is repeatedly applied to secondary nodes.

The following code block is a sample oplog entry, which contains fields such as ts, h, op, ns, and o:

```
    {
      "ts" : Timestamp(1446011584, 2),
      "h" : NumberLong("1687359108795812092"),
      "v" : 2,
      "op" : "i",
      "ns" : "test.nosql",
      "o" : { "_id" : ObjectId("563062c0b085733f34ab4129"), "name" : "mongodb", "score" : "
100" }
    }
```

- ts: the time when the operation was performed. The value contains two numbers. The first number is a UNIX timestamp. The second number is a counter that indicates the serial number of each operation that occurs within a second. The counter is reset every second.
- h: the unique identifier of the operation
- v: the version of the oplog
- op: the type of the operation
- i: insert
- u: update
- d: delete
- c: run commands such as createDatabase and dropDatabase
- n: null. This value is used for special purposes.
- ns: the collection on which the operation is performed
- o: the operation details. This field is valid for update operations only.
- o2: the conditions of an update operation. This field is valid for update operations only.

During the initial synchronization, a secondary node runs the `init sync` command to synchronize all data from the primary node or another secondary node that stores the latest data. Then, the secondary node continuously uses the `tailable cursor` feature to query the latest oplog entries in the local.oplog.rs collection of the primary node and applies the operations in these oplog entries.

The initial synchronization process includes the following steps:

1. Before the time of T1, the data synchronization tool runs the `listDatabases`, `listCollections`, and `cloneCollection` commands. At the time of T1, all data in the cloud databases (except the local.oplog.rs database) starts to be synchronized from the primary node to the secondary node. Assume that the synchronization is completed at the time of T2.
2. All the operations in oplog entries generated from T1 to T2 are applied to the secondary node. Operations in oplog entries are idempotent. Therefore, operations that have been applied in Step 1 can be reapplied.
3. Based on the index for each collection on the primary node, indexes for the corresponding collections are created on the secondary node. The index for each collection on the primary node was created in Step 1.

   ? **Note** You must configure the oplog size based on the database size and the volume of data to be written by the application. If the oplog is oversized, the storage space may be wasted. If the oplog size is too small, the secondary node may fail to complete initial synchronization. For example, in Step 1, if the database stores a large amount of data and the oplog is not large enough, the oplog may fail to store all the oplog entries generated from T1 to T2. As a result, the secondary node cannot synchronize all the data sets from the primary node.

Modify a replica set

You can modify a replica set by running the `replSetReconfig` command or running the `rs.reconfig()` command in the mongo shell. For example, you can add or delete members, and change the priority, vote, hidden, and delayed attributes of a member.

For example, you can run the following command to set the priority of the second member in the replica set to 2:

```
cfg = rs.conf();
cfg.members[1].priority = 2;
rs.reconfig(cfg);
```

Roll back operations on the primary node

Assume that the primary node of a replica set fails. If write operations have been performed on the new primary node when the former primary node rejoins the replica set, the former primary node must roll back operations that have not been synchronized to other nodes. This ensures data consistency between the former primary node and the new primary node.

The former primary node writes the rollback data to a dedicated directory. This allows the database administrator to run the `mongorestore` command to restore operations as needed.

Replica set read/write settings

- Read Preference

  By default, all the read requests for a replica set are routed to the primary node. However, you can modify the read preference modes on the drivers to route read requests to other nodes.

  - primary: This is the default mode. All read requests are routed to the primary node.
  - primaryPreferred: Read requests are routed to the primary node preferentially. If the primary node is unavailable, read requests are routed to secondary nodes.

  - secondary: All read requests are routed to secondary nodes.
  - secondaryPreferred: Read requests are routed to secondary nodes preferentially. If all secondary nodes are unavailable, read requests are routed to the primary node.
  - nearest: Read requests are routed to the nearest reachable node, which can be detected by running the `ping` command.

- Write Concern

  By default, the primary node returns a message that indicates a successful write operation after data is written to the primary node. You can set the write concern on drivers to specify the rule for a successful write operation. For more information, see Write concern.

  The following write concern indicates that a write operation is successful only after the data is written to a majority of nodes before the request times out. The timeout period is five seconds.

  ```
  db.products.insert(
    { item: "envelopes", qty : 100, type: "Clasp" },
    { writeConcern: { w: majority, wtimeout: 5000 } }
  )
  ```

  The preceding settings apply to individual requests. You can also modify the default write concern of a replica set. The write concern of a replica set applies to all the requests for the replica set.

  ```
  cfg = rs.conf()
  cfg.settings = {}
  cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
  rs.reconfig(cfg)
  ```