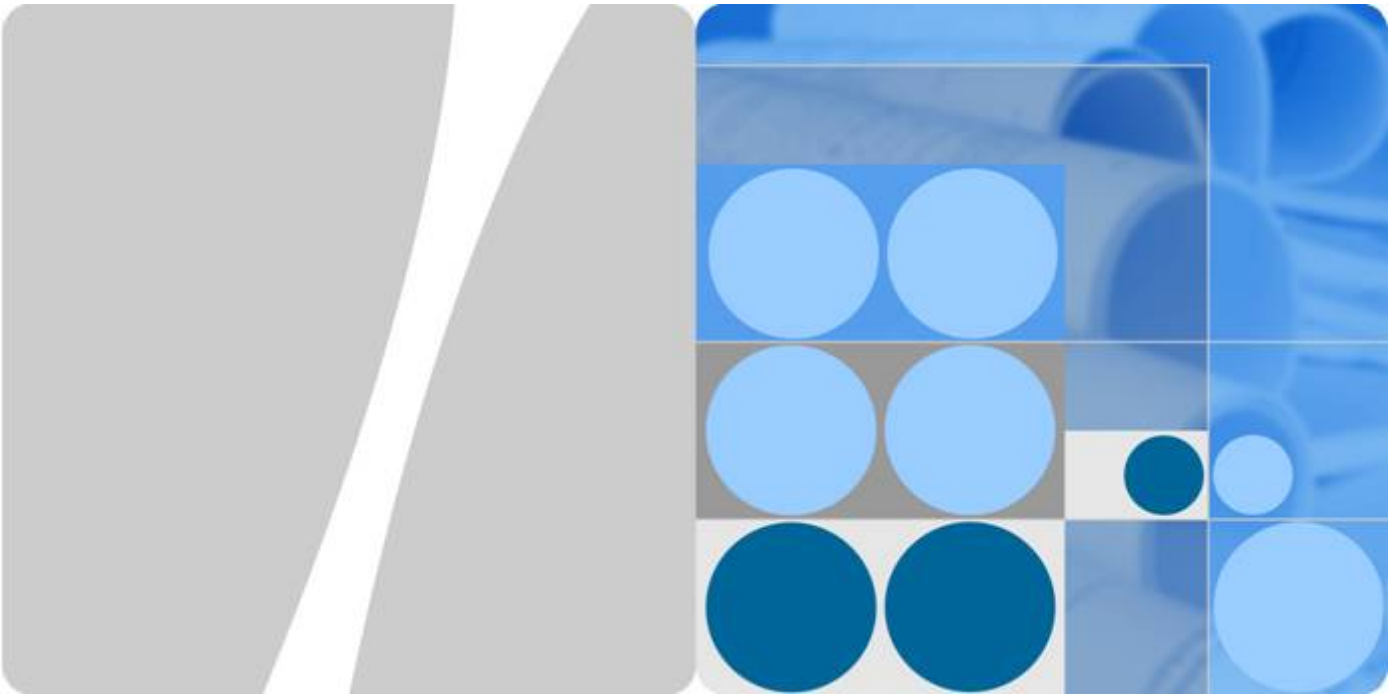


Distributed Database Middleware

Best Practices

Issue 01

Date 2020-03-18

HUAWEI TECHNOLOGIES CO., LTD.

HUAWEI

Trademarks and Permissions


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

Notice

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

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

Huawei Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: https://e.huawei.com

Contents

1 Selecting Proper Specifications

DDM supports multiple types of instances with different specifications. The more cores DDM has, the stronger its computing capacity becomes. The larger memory DDM has, the larger volumes of complex SQL statements can be queried and processed.

It is a basic requirement for databases to support concurrent service requests during peak hours. When using Query per Second (QPS) to measure service requirements, you can select proper specifications for DDM instances on the basis of testing data and your own service planning. This allows you to trim costs while meeting service demands, on the premise that DDM instances have no performance bottlenecks.

NOTE

- Actually, QPS depends on many factors, including the network, storage, table design, SQL execution plan, and size of a single data record. QPS metric values are obtained using the **sysbench** tool.
- This document is based on HUAWEI CLOUD DDM practices and will guide you through the functions provided by DDM.

OLTP Performance Test

Test Environment

- **AZ**: AZ1 of CN-North Beijing1
- **Subnet**: subnet-ff59 (192.168.0.0/24)
- **Intranet security group**: Sys-default
- **VPC**: vpc-blue
- **RDS DB instance class**: 16 vCPUs | 32 GB, common instance (c3)
- **Press (four sysbench tools)**: 16 vCPUs | 32 GB

Test Parameters

- An example SQL statement used for table creation is as follows:

```
CREATE TABLE sbtest1(
 id INTEGER UNSIGNED NOT NULL auto_increment PRIMARY KEY,
 k INTEGER UNSIGNED DEFAULT '0' NOT NULL,
 c CHAR(120) DEFAULT '' NOT NULL,
 pad CHAR(60) DEFAULT '' NOT NULL
 );
```


- Press test parameters are described as follows:

```
--test='/usr/local/share/sysbench/oltp.lua' Test model is oltp.lua
--oltp-table-size=160000000  Prepare 160 million data records.
--oltp_auto_inc=off Disable the auto increment primary key.
--oltp_skip_trx=off Skip transactions.
--oltp_secondary Set the ID to a non-primary key to prevent primary key conflicts.
--oltp_range_size=5 Continuously assign five values so that five shards must be gone through.
--rand-init=on Each test table is a sysbench tool filled with random data.
--num-threads=256/512 Number of concurrent threads
```

Test Results

Table 1-1 describes the test result.

Table 1-1 OLTP performance test

DDM Specifications	Concurrency	Total QPS of Nodes in Cluster Mode
8 vCPUs | 16 GB	256	49,474
16 vCPUs | 32 GB	256	87,302
32 vCPUs | 64 GB	256	152,671
64 vCPUs | 128 GB	512	317,431

Simple Query Performance Test

Test Environment

- **AZ**: AZ1 of CN-North Beijing1
- **RDS DB instance class**: 16 vCPUs | 32 GB, common instance (c3)
- **Number of RDS DB instances**: 8/16
- **RDS AZ**: AZ1 of CN-North Beijing1
- **Press test ECS specifications**: 16 vCPUs | 32 GB
- **Press test ECS region**: AZ1 of CN-North Beijing1

Test Parameters

- An example SQL statement for table creation is as follows:

```
CREATE TABLE sbtest1(
 id INTEGER UNSIGNED NOT NULL auto_increment PRIMARY KEY,
 k INTEGER UNSIGNED DEFAULT '0' NOT NULL,
 c CHAR(120) DEFAULT '' NOT NULL,
 pad CHAR(60) DEFAULT '' NOT NULL
 );
```

- Press test parameters are described as follows:

```
--test='/usr/local/share/sysbench/select.lua' The test model is select.lua, that is, equality query on sharding keys.
--oltp_tables_count=1 There is one table.
--oltp-table-size=160000000  Prepare 160 million data records.
--oltp_auto_inc=off Disable the auto increment primary key.
--oltp_secondary Set the ID to a non-primary key to prevent primary key conflicts.
--rand-init=on Each test table is a sysbench filled with random data.
--report-interval=1
--max-time=600 Perform the press test for 10 minutes.
--num-threads=256/512 Number of concurrent threads
```


Test Result

Table 1-2 describes the test result.

Table 1-2 Simple query performance test

DDM Specifications	Concurrency	QPS
8 vCPUs | 16 GB	256	108,003
16 vCPUs | 32 GB	256	205,228
32 vCPUs | 64 GB	256	425,585
64 vCPUs | 128 GB	512	796,614

2 Determining the Number of RDS DB Instances

RDS DB instances are data storage engines. Therefore, database storage and performance design must be properly planned.

1. Evaluate the database performance metrics required, for example, storage capacity, database throughput, and maximum number of database connections.
2. Select appropriate RDS DB instance specifications. The following example assumes that the RDS DB instance specifications are 8 vCPUs and 32 GB.
3. Calculate the minimum number of RDS DB instances required by each metric, which is the minimum value of **Number of RDS DB Instances for a Single Metric (Example)** in **Table 2-1**.

Table 2-1 Calculation of metrics

Single Metric	General Baseline (Example)	Baseline for a Single RDS DB Instance (Example)	Number of RDS DB Instances for a Single Metric (Example)	Reference Calculation Method
{Total storage capacity}	2000 GB	500 GB	4	{Total number of records} x {Number of bytes occupied by a single record}

Single Metric	General Baseline (Example)	Baseline for a Single RDS DB Instance (Example)	Number of RDS DB Instances for a Single Metric (Example)	Reference Calculation Method
{Throughput or QPS}	40,000	23,000	2	{Maximum number of supported requests per second} x {Number of database attempts of a single request}
...	...	...	...	...

4. Calculate the number of RDS DB instances.

   Calculate the maximum number of RDS DB instances required by each metric, which is the maximum value of **Number of RDS DB Instances for a Single Metric (Example)** in **Table 2-1**. The upper limit is 100.

   NOTE

   - If the data volume in an unsharded table is greater than 10 million entries, you are advised to split the data. Otherwise, query efficiency may be negatively affected.
   - When planning the RDS DB instance storage capacity, you are advised to base your decisions on the expected data volume for the next one or two years.
   - If the query concurrency pressure is high, you can add read replicas to RDS DB instances.

     Select an RDS DB instance on **Cloud Eye** and view its CPU and memory usage to determine the concurrency pressure.
   - If the overall service throughput or the RDS DB instance resource usage is high, you can add RDS DB instances to improve the parallel computing capabilities of databases.

3 Determining the Number of Shards in a Schema

Create schemas in a DDM instance to shard the tables that you will use. When creating a schema, import an appropriate RDS DB instance and select the number of shards for the RDS DB instance. Determine the number of shards by estimating your total data volumes on each sharded table to prevent tables in each shard from carrying excessive amount of data.

- In unsharded mode, a schema corresponds only to one RDS DB instance, and only one shard is created for the RDS DB instance.
- In sharded mode, a schema corresponds to multiple RDS DB instances. By default, eight shards are created for each RDS DB instance. You can also create 16, 32, 64, or 128 shards for an RDS DB instance. The number of shards cannot be changed once it is configured.

Determining the Number of Shards per RDS DB Instance

The number of shards per RDS DB instance must be properly planned. If specifications of an RDS DB instance exceed the upper limit, system performance is affected. If the total number of shards is fixed, evaluate specifications of an RDS DB instance, and determine the relationship between RDS DB instances and shards per RDS DB instance.

The following are the shard calculation formulas:

- Total number of shards = {Total number of records}/{Number of records in an unsharded table} = {Number of RDS DB instances} x {Number of shards per RDS DB instance}
- {Number of records for a single RDS DB instance} = {Number of records in an unsharded table} x {Number of shards per RDS DB instance}
- Storage capacity of a single RDS DB instance = {Number of records in a single RDS DB instance} x {Number of bytes in a single record}
- Specifications of a single RDS DB instance include the storage capacity, throughput, response delay, connections, and physical resources.

For tables that need to be sharded horizontally, evaluate the expected service scale for the next one to two years. Determine the number of shards per RDS DB instance based on **Table 3-1**.

Table 3-1 References for evaluating service scales

Item	Calculation Method Reference	User Value	Example
Total number of records	Evaluated based on the service scale	Enter the actual value.	1 billion = 1,000,000,000
Number of records in an unsharded table	Evaluated based on the service scale	Enter the actual value.	10,000,000
Total number of shards	{Total number of records}/{Number of records in an unsharded table} ({Number of RDS DB instances} x {Number of shards per RDS DB instance})	Enter the actual value.	100
Number of RDS DB instances	For details, see **Determining the Number of RDS DB Instances**.	Enter the actual value.	4
Number of shards per RDS DB instance	{Total number of shards}/{Number of RDS DB instances} Enumeration value: [8, 16, 32, 64, 128] **NOTE** • The number of shards per RDS DB instance is based on service requirements. If the number exceeds the upper limit, system performance is affected. • If instances with low specifications have many shards, adjust the specifications or add more instances.	Enter the actual value.	32

4 Selecting a Sharding Key and a Sharding Algorithm

This section describes when to use and how to select sharding keys and algorithms.

For details, see **Table 4-1**.

Table 4-1 Sharding keys and algorithms

Sharding Algorithm	Hash		Range	
Sharding Key	Table field	Table field+date function	Table field	Table field+date function
Description	Evenly distributes data to shards by the specified table field.	Evenly distributes data to shards by the specified table field and date function. The table field must be of the date type such as **date**, **datetime**, or **timestamp**.	Distributes data to specific shards according to the rules defined in the algorithm metadata.	Distributes data to shards by the specified table field and date function according to the rules defined in the algorithm metadata. The table field must be of the date type such as **date**, **datetime**, or **timestamp**).

Application Scenario	Applicable to scenarios requiring even data distribution. For example, for banking applications, the service logical entities are customers. In this case, use the table field corresponding to customers (for example, customer numbers) as the sharding key.	Applicable to scenarios requiring data splitting by time (year, month, day, week, and their combinations). For example, for game applications, use the table field corresponding to players (for example, player registration time) as the sharding key. Sharding by day, month, or year enables you to easily collect and query operation statistics of players for a specified day or month. This helps game vendors to conduct big data analysis.	Applicable to scenarios with a large number of range operations. For example, for e-commerce applications, if service scenarios focus on promotional activities and the service logical entities are activity dates, use the table field corresponding to activity dates (for example, activity name and date range) as the sharding key. This helps you collect statistics about the sales volume for a specified.	For example, when performing log analysis, you can select the time field as the sharding key because the log system may contain various types of complicated information. You can then shard data using the date function. To facilitate log clearing and dumping, select the range algorithm and convert the time field value into year using the date function so that logs are stored in shards by year. For details, see examples in the following passages.

Selecting a Sharding Algorithm

A sharding algorithm shards data in logical tables to multiple shards. DDM supports hash and range algorithms.

- Hash

  This algorithm evenly distributes data across shards.

  It is applicable to SQL query conditions that frequently use the operators **=** and **IN**.

- Range

  This algorithm stores records in tables based on the value range specified in the algorithm metadata.

  It is applicable to SQL query conditions that frequently use the operators greater (>), less (<), and BETWEEN ... AND ...

Select an appropriate algorithm based on your service requirements to improve efficiency.

NOTE

During a seamless schema scale-out:

- Data of logical tables sharded by the hash algorithm is migrated and redistributed evenly across all shards.
- By default, data of logical tables sharded by the range algorithm is not migrated. If you modify the splitting rule and redefine the range, connect to the RDS shards to complete data migration.

Selecting a Sharding Key

A sharding key is a data table field used for generating route results during horizontal sharding of logical tables. After specifying a table field, you can further select a date function. Alternatively, enter "date function (field name)". The data table field must be of the date type, such as **date**, **datetime**, or **timestamp**. A date function is applied to data table sharding by time (year, month, day, week, or their combinations).

DDM calculates the route results based on the sharding key and sharding algorithm, executes horizontal sharding for the data, and distributes the data to shards.

Note the following rules for selecting a sharding key and a sharding algorithm:

- Ensure that data is evenly distributed to each shard.
- The sharding key should be the most frequently used or the most important query condition.
- You are advised to select the primary key as the sharding key. The query speed is the fastest when the primary key is used as the query condition.

Service Scenarios with a Clear Entity

Generally, the data volume of a sharded table reaches tens of million entries. Therefore, selecting an appropriate sharding key and a sharding algorithm is important. If the service entity can be identified and most database operations are performed on data of the entity, the table field corresponding to the entity can be selected as the sharding key for horizontal sharding.

The service logical entity depends on actual application scenarios. In the following scenarios, a clear service logical entity exists.

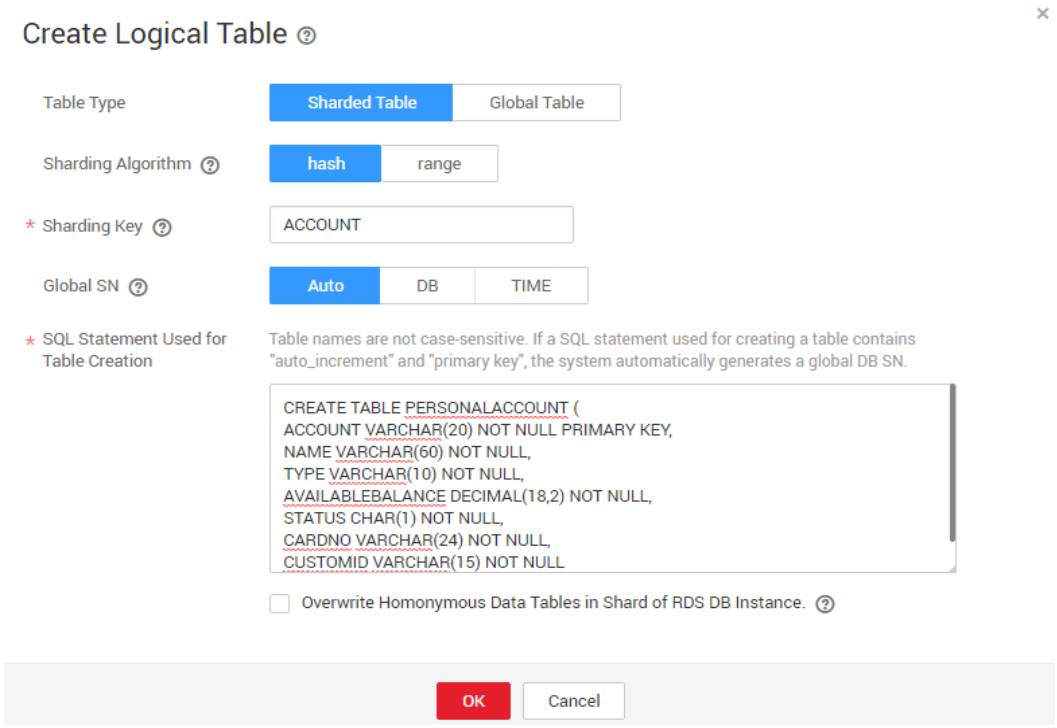
1. For banking applications, the service logical entities are customers. In this case, use the table field corresponding to customers (for example, customer numbers) as the sharding key. Service scenarios of some systems are based on bank cards or accounts. In such cases, select the bank card or account as the sharding key.
2. For e-commerce applications, if service scenarios are based on products, the service logical entities are products. In this case, use the table field corresponding to products (for example, product code) as the sharding key.
3. Game applications mainly focus on player data, and the service logical entities are players. In this case, use the table field corresponding to players (for example, player ID) as the sharding key.

For example, for a bank service, a SQL statement for table creation is as follows:

```
CREATE TABLE PERSONALACCOUNT (
ACCOUNT VARCHAR(20) NOT NULL PRIMARY KEY,
NAME VARCHAR(60) NOT NULL,
TYPE VARCHAR(10) NOT NULL,
AVAILABLEBALANCE DECIMAL(18,2) NOT NULL,
STATUS CHAR(1) NOT NULL,
CARDNO VARCHAR(24) NOT NULL,
CUSTOMID VARCHAR(15) NOT NULL
) ENGINE=INNODB DEFAULT CHARSET=UTF8;
```

Figure 4-1 shows the example SQL statement for table creation.

Figure 4-1 Table creation statement for a bank service



Service Scenarios Without a Clear Entity

If no suitable entity is found in a service scenario, select the table field corresponding to attributes with even data distribution as the sharding key.

For example, in a log analysis scenario, the log system may contain complex information. In this case, select the time field as the sharding key.

When the time field is selected as the sharding key, you can further specify a date function for the sharding key for data sharding.

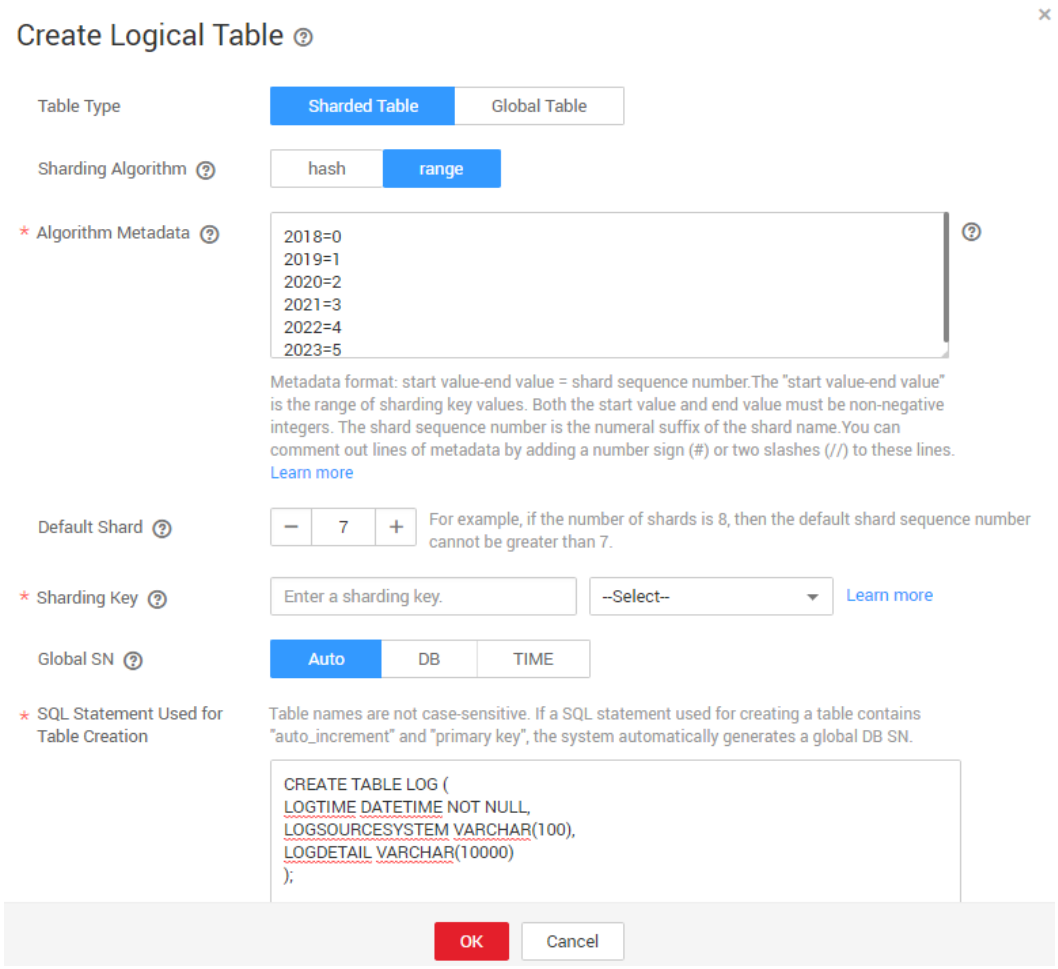
To facilitate log clearing and dumping, select the range algorithm and convert the time field value into year using the date function so that logs are stored in shards by year. For details, see **Figure 4-2**.

SQL statement for creating a table:

```
CREATE TABLE LOG (
LOGTIME DATETIME NOT NULL,
LOGSOURCESYSTEM VARCHAR(100),
```

```
LOGDETAIL VARCHAR(10000)
);
```

Figure 4-2 Table creation statement for a log analysis scenario



5 Formulating Sharding Rules

When creating a schema and selecting **Sharded Table** for **Table Type**, you need to specify a sharding algorithm and a sharding key.

If an entity-relationship exists between data tables, formulate the same sharding rule for these tables, and select associated table fields as the sharding key. In this way, associated data in different tables is stored in one shard to avoid cross-shard JOIN.

For example, when creating sharded tables for customer tables, order tables, or order details tables, you are advised to select the customer ID as the sharding key.

6 Using Global and Unsharded Tables

An unsharded table is a one whose data is stored in only one default shard. A global table is a table storing the same data in all shards to improve JOIN efficiency.

Unsharded Table

The DDM console does not allow creation of unsharded tables. Create unsharded tables after connecting to a DDM instance through a MySQL client or an application.

If a table contains fewer than 10 million entries and its data is not associated with other sharded tables for query, you are advised to set the table type to **Unsharded Table** and store data in the default shard.

Global Table

In business databases, some tables have a small data volume and low update frequency, but are often used for JOIN operations.

To support JOIN operations of such tables and sharded tables, DDM provides global tables, which have the following features:

- A global table stores the same data in each shard. Data insertion, update, and deletion in global tables are executed in each shard in real time.
- Queries of global tables are only executed in one shard.
- All tables support JOIN with global tables.

For example,

an e-commerce company wants to query orders from Guangdong Province in the order management system. JOIN query of regional tables and order tables is required. Order tables are generally stored in different shards because they have a large data volume. To prevent cross-shard JOIN operations, users can set the regional table as a global table that stores the same data in all shards.

For details on how to configure global tables, see "Creating a Schema" in the *Distributed Database Middleware User Guide*. When creating a schema, set the table type to **Global Table**.

NOTICE

Do not operate global tables or perform full table scanning operations such as executing SQL statements that do not contain sharding conditions in high concurrency. Otherwise, an error message may occur, indicating that the number of backend RDS connections is insufficient. You are advised to perform such operations during off-peak hours.

7 Selecting a Distributed Transaction Mode

DDM supports three distributed transaction modes, including single-shard, best-effort commit, and eventual atomicity.

Table 7-1 describes their features.

Table 7-1 Features of distributed transaction modes

Transaction Mode	Advantage	Disadvantage
Single-shard	The execution efficiency is high, and the underlying database ensures strong consistency.	If a transaction involves multiple shards, DDM refuses to execute the transaction and returns an error.
Best-effort commit	Cross-shard transactions are supported. Their execution results are independent and do not interfere with each other.	The execution efficiency is lower than that of the single-shard mode. If transaction commit fails for any shard, no compensating transaction is provided. In this case, transaction execution of some shards fails while that of other shards succeeds.
Eventual atomicity	Cross-shard transactions are supported, and the transaction execution results are eventually consistent for all shards.	The execution efficiency is lower than that of the best-effort commit mode. If transaction commit fails for any shard, a compensating transaction is provided. During the compensation, the result queried from the client is not the latest, which means that intermediate states exist.

NOTE

- For DDM, only one transaction mode can be selected for one schema. When creating a schema, plan the data structure for storage and required SQL statements, for example, determine whether cross-shard deployment is implemented and whether transaction consistency must be ensured. Select an appropriate distributed transaction mode.
- Currently, you can only customize hints for SELECT or UPDATE statements.
- In final atomicity mode, do not delete any fields in tables or add fields. If no exception occurs in the current transaction or the compensation is complete, you can add a field at the end of the table.

Single-Shard

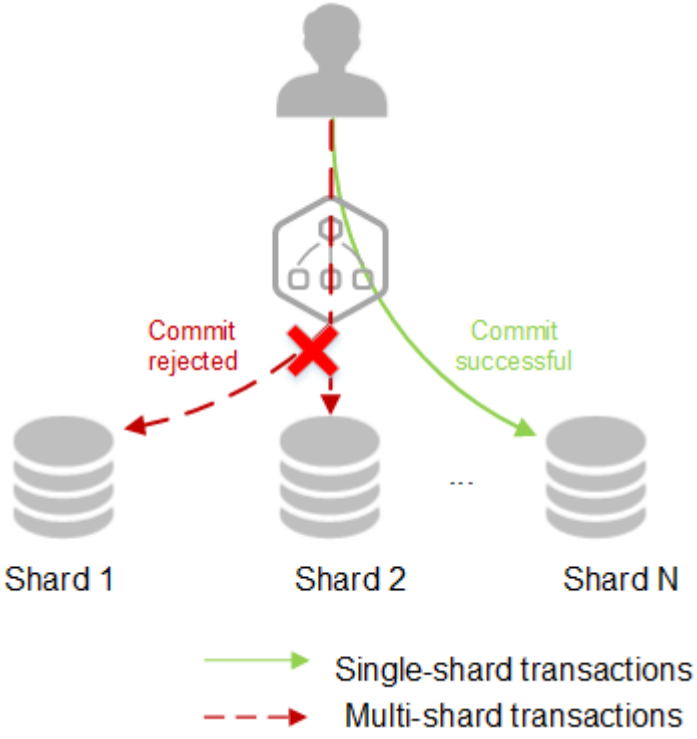
Overview

A transaction can be executed only on one shard. If a transaction involves multiple shards, DDM refuses to execute the transaction and returns an error. **Figure 7-1** illustrates the principle.

Application Scenario

This mode is suitable when service splitting is reasonable, a complete transaction processing framework is available at the application layer, and transactions are to be executed on only one shard. The underlying database ensures strong consistency of single-shard transactions. In this mode, DDM refuses to execute cross-shard transactions and returns an error so that expected results are achieved.

Figure 7-1 Single-shard transaction mode



Best-Effort Commit

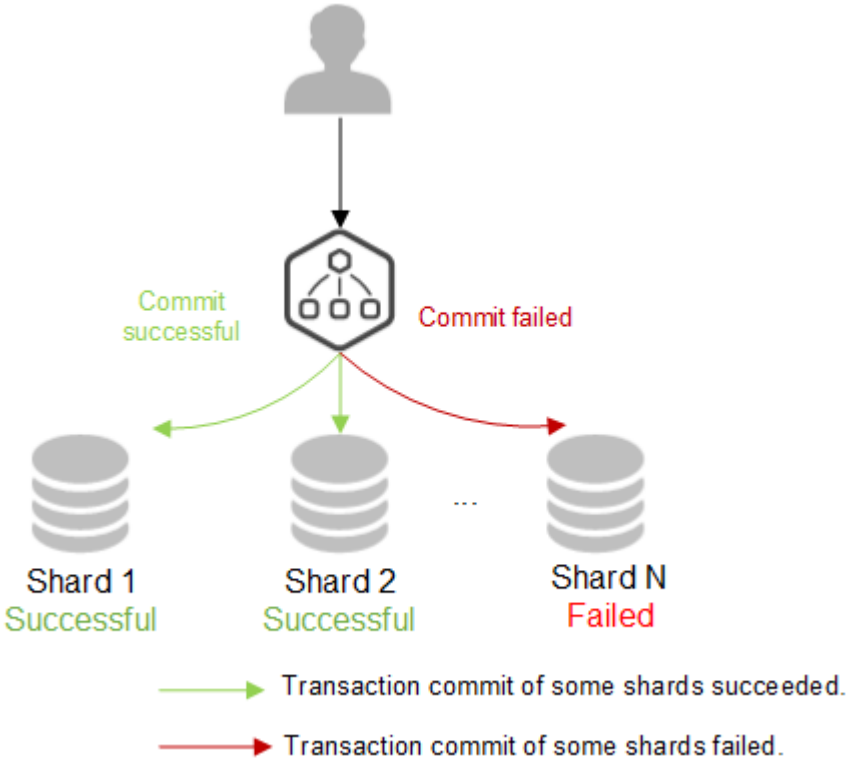
Overview

Transactions are independently committed on each shard. Transaction commit results remain consistent on all shards to the greatest extent possible, but it is likely that some shards succeed while others fail. **Figure 7-2** illustrates the principle.

Application Scenario

This mode is suitable for most services that do not involve monetary transactions. It achieves a proper tradeoff between performance and consistency. The transaction commit command is sent to multiple nodes, and there is a small chance that some nodes succeed while others fail in the event of a commit time window exception.

Figure 7-2 Best-effort commit transaction mode



Eventual Atomicity

Overview

Transaction commit results do not always remain consistent on all shards. If transaction commit fails for any shard, DDM enables the compensation mechanism on other shards with successful commit results to undo previous modifications. In this way, the transaction states of all shards remain eventually consistent.

NOTE

In eventual consistency scenarios, if there are concurrent query requests, the query results may not reflect the final states, but intermediate states. For example, the commit operation may be completed on some shards while being executed on other shards.

Application Scenario

This mode is suitable for scenarios with high consistency requirements. It solves the issue that transaction commit succeeds on some nodes while fails on others under the best-effort commit mode. If high consistency requirements are placed on some read SQL statements, you can use **select for update** or **lock in share mode** to avoid reading inconsistent states (coexistent failed and successful states).

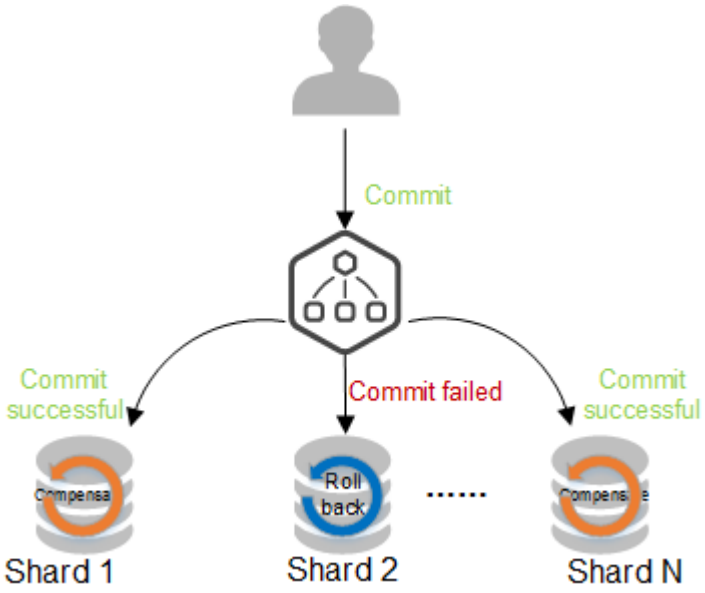
For example,

select col1, col2,...coln from table1 where col1=*{sharding key}* for update;

select col1, col2,...coln from table1 where col1=*{sharding key}* lock in share mode;

You are advised to specify a sharding key in the WHERE condition.

Figure 7-3 Eventual atomicity transaction mode



8 Migrating the Entire MyCat Database to DDM

Scenario

The following describes how to migrate an entire MyCat database to DDM.

NOTE

- Services may be interrupted during migration. The interruption duration depends on the amount of data to be migrated and network conditions.
- Data migration is complicated and is recommended during off-peak hours. This guide is for reference only. Design an appropriate migration scheme based on your service scenarios, data volume, and downtime requirements.
- If a large amount of data is involved, contact DDM technical engineers by submitting a service ticket or through after-sales services. Fully rehearse the migration before migrating data.
- You can access DDM only through an ECS. Therefore, you must export databases as files to the ECS, and then import data in the files into DDM from the ECS.

Preparations for Migration

- Prepare an ECS that can access MyCat, the target DDM instance, and the RDS DB instance associated with the target DDM instance.
  a. Ensure that MyCat, the target DDM instance, and the RDS DB instance associated with the target DDM instance are in the same VPC for network connectivity.
  b. Configure the same security group for the ECS running MyCat, the target DDM instance, and the RDS DB instance associated with the target DDM instance. If they belong to different security groups, configure their security group rules to allow them to access each other.
  c. Install an official MySQL client on the ECS. Version 5.6 or 5.7 is recommended.
     - Install a Linux OS of the Red Hat series: **yum install mysql mysql-devel**
     - Install a Linux OS of the Debian series: **apt install mysql-client-5.7 mysql-client-core-5.7**

d. Ensure that sufficient disk space and memory are available on the ECS for storing and comparing dump files.

- Prepare a DDM instance that has been associated with an RDS DB instance, and configure the DDM account, schema, and logical table for it.
- MyCat 1.6 is used as an example.

Migration Scheme

Table types in MyCat and DDM are different from each other and migration schemes vary by table type, as described in **Table 8-1**.

Table 8-1 Migration schemes

MyCat Table Type	DDM Table Type	Migration Scheme
Unsharded table	Unsharded table	1. Export the **structure data** and **table data** from MyCat. 2. Connect to the RDS DB instance associated with the target DDM instance and **import data to the target DDM instance**.
Sharded table: hash sharding (including by date)	Sharded table: hash sharding (including date functions)	1. Export all table structure data from MyCat. 2. Create tables with structures identical to those of the exported tables on the DDM console. For details, see section "Creating a Logical Table on the DDM Console". 3. **Export data from the entire MyCat database to DDM**. 4. **Connect to DDM to import the entire DB data**.
Sharded table: ranging sharding (including by date)	Sharded table: range sharding (including date functions)	
Global table	Global table	

Constraints

- To ensure data integrity, stop MyCat services before data migration.
- In this scenario, associating MyCat with DDM for data association is not supported. You must export data from MyCat and then import the data into DDM.

- The version of the RDS DB instance associated with the target DDM instance must be consistent with the version of MyCat.

Exporting Table Structure Data from MyCat

Step 1 Log in to the target ECS.

Step 2 Run the following command to export table structure data in the MyCat database. Set the parameters in italics to actual values. For details about the parameters, see **Table 8-2**.

```
mysqldump -h {DB_ADDRESS} -P {DB_PORT} -u {DB_USER} -p    --skip-lock-tables --add-locks=false --set-gtid-purged=OFF --no-data --order-by-primary {DB_NAME} > {mysql_schema.sql}
Enter password: **********
```

Table 8-2 Parameter description

Parameter	Description	Remarks
DB_ADDRESS	Connection address of a database whose data is to be exported	Mandatory
DB_PORT	Listening port of a database	Mandatory
DB_USER	Database user	Mandatory
DB_NAME	Database name	Mandatory
mysql_schema.sql	Generated name of a table structure file	The file varies depending on the table whose structure is exported. You are advised to name the file in the format of "schema name" + "_" + "schema" to prevent data from being overwritten, for example, **mysql_schema.sql**.
mysql_data.sql	Generated name of a DB data file	N/A

----End

Exporting Data from the Entire MyCat Database

Step 1 Log in to the ECS.

Step 2 Run the following command to export data of the MyCat database. Set the parameters in italics to actual values. For details about the parameters, see **Table 8-2**.

```
mysqldump -h {DB_ADDRESS} -P {DB_PORT} -u {DB_USER} -p    --hex-blob --complete-insert --skip-lock-tables --add-locks=false --set-gtid-purged=OFF --quick --no-create-info --order-
```



```
by-primary {DB_NAME} > {mysql_data.sql}
Enter password: **********
```

Step 3 Run the following command on the ECS to view the exported SQL file.

```
ls -l
```

----End

Importing Data in Unsharded Tables into the Target DDM Instance

On the ECS, use a MySQL client to connect to the RDS DB instance associated with the target DDM instance. Run the following commands to import data of the table structure text files and database files.

For unsharded or common tables:

```
mysql -f -h {RDS_ADDRESS} -P {RDS_PORT} -u {RDS_USER} -p {DB_NAME} < {mysql_table_schema.sql}
Enter password: **********
mysql -f -h {RDS_ADDRESS} -P {RDS_PORT} -u {RDS_USER} -p {DB_NAME} < {mysq_table_data.sql}
Enter password: **********
```

- **RDS_ADDRESS** indicates the address of an RDS DB instance into which data is to be imported.
- **RDS_PORT** indicates the port number of an RDS DB instance.
- **RDS_USER** indicates the username of an RDS DB instance.
- **DB_NAME** indicates the name of an RDS database. If data of unsharded tables is to be imported, **DB_NAME** indicates the name of the first shard of RDS.
- **mysql_table_schema.sql** indicates the name of a table structure file to be imported.
- **mysq_table_data.sql** indicates the name of a table data file to be imported.

Importing Data (in a Sharded or Global Table) into the Target DDM Instance

On the ECS, use a MySQL client to connect to DDM and run the following command to import the entire DB data into DDM.

```
mysql -f -h {DDM_ADDRESS} -P {DDM_PORT} -u {DDM_USER} -p {DB_NAME} < {mysql_schema.sql}
Enter password: **********
```

Table 8-3 Parameter description

Parameter	Description	Remarks
DDM_ADDRESS	Connection address of the DDM instance into which data is to be imported	View the connection address and port number on the **Basic Information** tab on the DDM console.
DDM_PORT	Listening port of the DDM instance into which data is to be imported	

Parameter	Description	Remarks
DDM_USER	User accessing the DDM instance	Account for creating a schema. The account must have both read and write permissions.
DB_NAME	Name of the schema into which data is to be imported	N/A
mysql_data.sql	Name of the entire DB data file to be imported	Name of the file exported in **Step 2**

9 Accessing DDM Through the JDBC Connection Pool

Scenario

The principle of a connection pool is as follows: The system stores database connections in memory during initialization. When you send a request to access a database, the system directly selects an available connection in the connection pool. If the connection is not required, the system returns the connection to the pool for another request to access. Connection establishment and disconnection are both managed by the connection pool itself. In addition, you can set connection pool parameters to control the number of initial connections in the connection pool, upper and lower limits of connections, maximum number of times that each connection can be used, and maximum idle time for each connection. You can also monitor the number and usage of database connections through your own management mechanism.

This section describes how to access DDM through the JDBC connection pool to perform data operations. For Java programs, **HikariCP** is recommended.

- Java 8: Version 3.3.1 is recommended.
- Java 7: Version 2.4.13 is recommended.

Procedure

Step 1 Configure Maven.

- Java 8:

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>3.3.1</version>
</dependency>
```

- Java 7:

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP-java7</artifactId>
    <version>2.4.13</version>
</dependency>
```

Step 2 Create a table.

Table 9-1 Parameters for creating a table

Table Name	Field	Type	Primary Key
account	account_number	bigint	Yes
	account_type	varchar (45)	No
	account_name	varchar (50)	No

Step 3 Connect to a DDM instance.

1. Set the number of connections by configuring parameters in the JDBC URL and HikariCP parameters.
2. Insert a data record.

Example:

```
package com.huawei.ddm.examples;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

public class HikariCPDemo {
    private static DataSource datasource;
    private static DataSource getDataSource() {
        if (datasource == null) {
            HikariConfig config = new HikariConfig();
            // Configure parameters in the JDBC URL:
            config.setJdbcUrl("jdbc:mysql:loadbalance://192.168.0.10:5066,192.168.0.11:5066/db_name?loadBalanceAutoCommitStatementThreshold=5&loadBalanceHostRemovalGracePeriod=15000&loadBalanceBlacklistTimeout=60000&loadBalancePingTimeout=5000&retriesAllDown=10&connectTimeout=10000&socketTimeout=60000");
            /*
            // Configure parameters in the JDBC URL as follows:
            config.addDataSourceProperty("loadBalanceAutoCommitStatementThreshold",5);
            config.addDataSourceProperty("loadBalanceHostRemovalGracePeriod", 15000);
            config.addDataSourceProperty("loadBalanceBlacklistTimeout", 60000);
            config.addDataSourceProperty("loadBalancePingTimeout", 5000);
            config.addDataSourceProperty("retriesAllDown", 10);
            config.addDataSourceProperty("connectTimeout", 10000);
            */
            config.setUsername("username");
            config.setPassword("password");
            config.setMaximumPoolSize(10);
            config.setAutoCommit(true);

            // Configure HikariCP parameters
            config.addDataSourceProperty("cachePrepStmts", true);
            config.addDataSourceProperty("prepStmtCacheSize", 250);
            config.addDataSourceProperty("prepStmtCacheSqlLimit", 2048);
            config.addDataSourceProperty("minimumIdle", 5);
            config.addDataSourceProperty("maximumPoolSize", 10);
            config.addDataSourceProperty("idleTimeout", 30000);

            datasource = new HikariDataSource(config);
        }
        return datasource;
    }

    public static void main(String[] args) {
```



```
        Connection connection = null;
        PreparedStatement pstmt = null;
        ResultSet resultSet = null;
        try {
            DataSource dataSource = getDataSource();
            connection = dataSource.getConnection();
            System.out.println("The Connection Object is of Class: " + connection.getClass());

            // Insert test data
            String insertSql = "insert into account(account_number, account_type, account_name)
values(?, ?, ?);";
            PreparedStatement insertStmt = connection.prepareStatement(insertSql);
            insertStmt.setLong (1, 1L);
            insertStmt.setString (2, "manager");
            insertStmt.setString (3, "demotest01");
            insertStmt.executeUpdate();
            connection.commit ();

            // Query data
            pstmt = connection.prepareStatement("SELECT * FROM account");
            resultSet = pstmt.executeQuery();
            while (resultSet.next()) {
                String accountNumber = resultSet.getString("account_number");
                String accountType = resultSet.getString("account_type");
                String accountName = resultSet.getString("account_name");
                System.out.println(accountNumber + "," + accountType + "," + accountName);
            }
        } catch (Exception e) {
            try {
                if (null != connection) {
                    connection.rollback();
                }
            } catch (SQLException e1) {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
    }
}
```

----End

10 Connecting to DDM Instances

Number of Connections

max_connection indicates the maximum number of clients that can be concurrently connected to a DDM instance. The value ranges from **10** to **10000**. Set the value of **max_connection** as required.

The **max_connection** value of RDS ranges from **1** to **100000**. Set the value as required.

Maximum number of connections from DDM to RDS = (Maximum number of RDS connections – 10) / Number of DDM instance nodes

The value of **max_connection** corresponding to the number of client connections must be greater than or equal to the maximum number of connections from DDM to RDS.

Adjust the value of **max_connections** to prevent it from becoming a bottleneck for connections between DDM and RDS. Short connections are disconnected after they are idle for a specified long time and may be partially released to avoid frequent establishment. The number of connections is 1/3 of the maximum number.

Connection Modes

- Configure a whitelist for EIPs. You can bind the EIP to your DDM instance and use this EIP to connect to the DDM instance.
- Buy an ECS and ensure that the ECS is in the same security group and VPC as the target DDM instance.

Scenario

This section uses an ECS as an example to describe how to connect to a DDM instance.

Procedure

Step 1 Buy an ECS and ensure that the ECS is in the same security group and VPC as the target DDM instance.

Step 2 Log in to the DDM console, select **RDS DB Instance Import** in the left pane and click **Import RDS DB Instance** in the upper right corner.

Step 3 Select **DDM Instance Management** in the left pane, locate the target DDM instance in the instance list, and click the name of the instance. On the displayed page, select the **Schema Management** tab and click **Create Schema**.

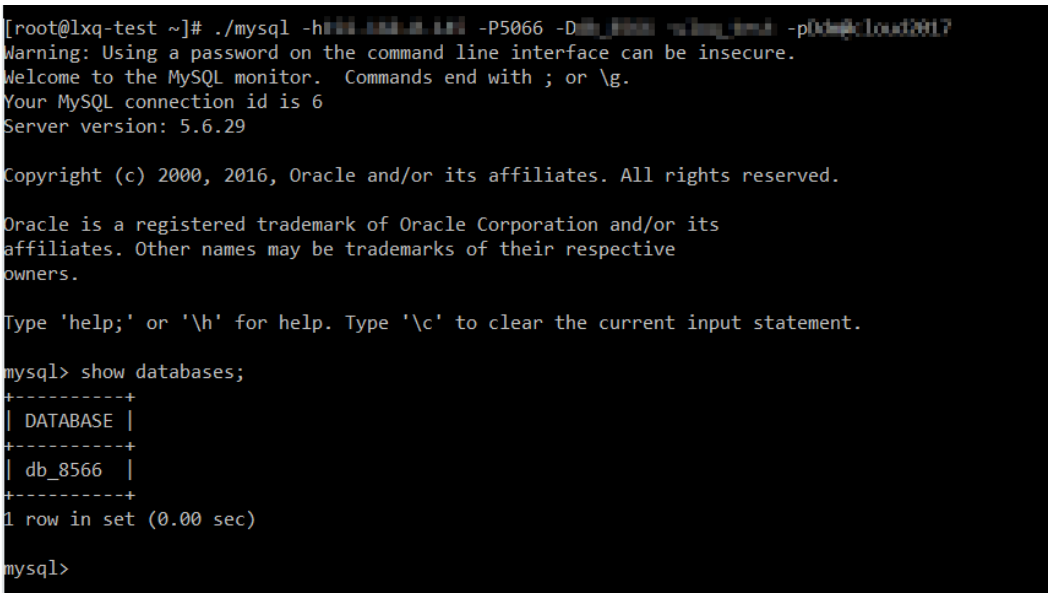
Step 4 On the displayed page, configure the required parameters. In the **Create DDM Account** dialog box, create a DDM account and use it to connect to the target DDM instance.

Step 5 After the schema is created, locate the created schema and copy its connection address (CLI).

Step 6 Bind an EIP to the ECS, enter the EIP in MobaXterm to connect to the ECS, and upload the MySQL client package using MobaXterm.

Step 7 In the ECS list, locate the target ECS, and click **Remote Login** in the **Operation** column. In the directory containing MySQL client, execute the connection address command copied in **5**.

If the following information is displayed, the connection is successful.



----End

