Distributed Relational Database Service

Best Practice

Best Practice

Choose the DRDS and RDS instance specifications

Distributed Relational Database Service (DRDS) and ApsaraDB for RDS divide instance specifications based on factors such as CPU processing capacity, memory size, and disk space, and provide instances of different specifications. Higher specifications mean higher processing capacities.

DRDS instance series

DRDS provides dedicated instances each with at least two DRDS server nodes to ensure high availability and stability. You can find a comparison of different instance series as follows:

Starter Edition: Each DRDS server node has 4 cores and 16 GB memory. This edition is applicable to initial business development and test scenarios. It does not support complex query acceleration.

Standard Edition: Each DRDS server node has 8 cores and 32 GB memory. This edition provides a wide range of specifications and features high cost-effectiveness. This edition is applicable to online business scenarios highlighting ultra-high concurrency, complex queries, and lightweight analysis. Parallel query is supported by default to improve the efficiency of executing complex queries such as multi-table join, aggregation, and sorting for online businesses.

Enterprise Edition: Each DRDS server node has 16 cores and 64 GB memory. This edition features high-specification resources and is applicable to enterprise-level business scenarios highlighting ultra-high concurrency, complex queries for massive data, and analysis acceleration. Parallel query is supported by default to greatly improve the efficiency of executing complex queries and report analysis for large amounts of data.

Select DRDS instance type and specification

DRDS is a computing-intensive service. Its processing capability depends on the CPU performance and is measured by QPS. We recommend that you select instances based on the maximum QPS value supported by each specification and the estimated maximum QPS value required for your business. For example, if you estimate that the maximum QPS of your business application to a DRDS instance is 50,000, you can select the Standard Edition instance with 16 cores and 96 GB memory.

Select RDS instance specification

- Estimate the data increment in the next one to two years to determine the maximum disk space required.
- Estimate the maximum IOPS required by an ApsaraDB RDS for MySQL instance.

We recommend that you purchase multiple ApsaraDB RDS for MySQL instances of low or medium specifications, so that you can rapidly upgrade ApsaraDB RDS for MySQL instance specifications in the event of storage bottlenecks. For more information about ApsaraDB RDS for MySQL instance specifications, see Create an ApsaraDB RDS for MySQL instance.

Choose a shard key

A shard key is a field for database sharding and table sharding, which is used to create sharding rules during horizontal partitioning. Distributed Relational Database Service (DRDS) calculates the shard key value by using a sharding function to get a result, and then shards the data to ApsaraDB RDS for MySQL instances based on this result.

The primary principle for table sharding is to try to find the business logic entity to which the data belongs, and ensure that most or core SQL operations or the SQL operations of a certain concurrency level are performed for this entity. Then the field corresponding to this entity can be used as the shard key.

Business logic entities are typically related to application scenarios. The following typical application scenarios all have specific business logic entities, and the identifier fields of these entities can be used as shard keys:

- Operations for user-oriented Internet applications focus on users. Therefore, the business logic entities are users, and user ID can be used as the shard key for associated tables.
- Operations for seller-oriented e-commerce applications focus on sellers. Therefore, the business logic entities are sellers, and seller ID can be used as the shard key.
- Operations for online gaming applications focus on players. Therefore, the business logic entities are players, and player ID can be used as the shard key.
- Operations for online Internet-of-Vehicle (IoV) applications focus on vehicles. Therefore, the

business logic entities are vehicles, and vehicle ID can be used as the shard key.
- Online tax affair-related applications offer frontend services for taxpayers. Therefore, the business logic entities are taxpayers, and taxpayer ID can be used as the shard key.

You can identify the business logic entities for other scenarios in the same way.

For example, in a seller-oriented e-commerce application, the following single table needs to be horizontally partitioned:

```
CREATE TABLE sample_order (
id INT(11) NOT NULL,
sellerId INT(11) NOT NULL,
trade_id INT(11) NOT NULL,
buyer_id INT(11) NOT NULL,
buyer_nick VARCHAR(64) DEFAULT NULL,
PRIMARY KEY (id)
)
```

Sellers are the business logic entities. Then, you can use the sellerId field as the shard key. The distributed data definition language (DDL) statement for table creation is as follows:

```
CREATE TABLE sample_order (
id INT(11) NOT NULL,
sellerId INT(11) NOT NULL,
trade_id INT(11) NOT NULL,
buyer_id INT(11) NOT NULL,
buyer_nick VARCHAR(64) DEFAULT NULL,
PRIMARY KEY (id)
) DBPARTITION BY HASH(sellerId)
```

If you fail to identify any appropriate business logic entity as the shard key, which case is often true in traditional enterprise-level applications, use the following methods to identify an appropriate shard key:

Identify a shard key based on the balance of data distribution and access so that data in table can be evenly distributed to different table shards. DRDS will soon provide the global secondary index for ensuring strong consistency, which can be used with parallel query to improve the SQL query concurrency and reduce the response time in this scenario.

Determine the shard key by combining the fields of the numeric (string) type and the time type. This method is applicable to log retrieval.

For example, a log system records all user operations and needs to horizontally partition the following single log table:

```
CREATE TABLE user_log (
```

```
userId INT(11) NOT NULL,
name VARCHAR(64) NOT NULL,
operation VARCHAR(128) DEFAULT NULL,
actionDate DATE DEFAULT NULL
)
```

You can combine the user identifier and the time field as the shard key for table sharding by days of a week. The distributed DDL statement for table creation is as follows:

```
CREATE TABLE user_log (
userId INT(11) NOT NULL,
name VARCHAR(64) NOT NULL,
operation VARCHAR(128) DEFAULT NULL,
actionDate DATE DEFAULT NULL
) DBPARTITION BY HASH(userId) TBPARTITION BY WEEK(actionDate) TBPARTITIONS 7
```

Choose the number of shards

Distributed Relational Database Service (DRDS) supports the horizontal partitioning of databases and tables. By default, eight physical database shards are created on an ApsaraDB RDS for MySQL instance, and one or more physical table shards can be created on each physical database shard. The number of table shards is also called the number of shards.

Generally, we recommend that each physical table shard contain no more than 5 million rows of data. Generally, you can estimate the data increment in the next one to two years. Then you can divide the estimated total data size by the total number of physical database shards, and divide the result by the recommended maximum data size of 5 million rows, to figure out the number of physical table shards to be created on each physical database shard:

```
Number of physical table shards in each physical database shard = CEILING(Estimated total data size/(Number of ApsaraDB RDS for MySQL instances x 8)/5,000,000)
```

Therefore, when the calculated number of physical table shards is equal to 1, only database sharding needs to be performed, that is, a physical table shard is created in each physical database shard. If the calculation result is greater than 1, we recommend that you perform both database sharding and table sharding, that is, there are multiple physical table shards in each physical database shard.

For example, if you estimate that a table may contain about 100 million rows of data two years later and have purchased four ApsaraDB RDS for MySQL instances, the number of physical table shards in each physical database shard is calculated as follows:

```
Number of physical table shards in each physical database shard = CEILING(100,000,000/(4 x 8)/5,000,000) =
```

```
CEILING(0.625) = 1
```

If the result is 1, only database sharding is needed, that is, one physical table shard is created in each physical database shard.

If only one ApsaraDB RDS for MySQL instance is used in this example, the calculation result is:

```
Number of physical table shards in each physical database shard = CEILING(100,000,000/(1 x 8)/5,000,000) =
CEILING(2.5) = 3
```

The result is 3, therefore we recommend that you create three physical table shards in each physical database shard.

Determine the time for configuration upgrade

Database performance can be measured by the response time (RT) and queries per second (QPS). RT reflects the performance of a single SQL statement, which can be improved through SQL optimization and other methods. Distributed Relational Database Service (DRDS) upgrade expands the capacity to improve performance, and is suitable for database access with low latency and high QPS.

The performance of a DRDS instance depends on the performance of DRDS and ApsaraDB RDS for MySQL. Insufficient performance of any DRDS or ApsaraDB for RDS node can result in a bottleneck in the overall performance. This topic describes how to check the performance metrics of a DRDS instance and upgrade the DRDS instance to address performance bottlenecks.

Determine the performance bottleneck of a DRDS instance

The QPS and CPU utilization of a DRDS instance are positively correlated. When the DRDS instance has performance bottlenecks, the CPU utilization of the DRDS instance remains high.

View the CPU utilization

1. In the DRDS console, click **Instances** in the left-side navigation pane.
2. Click the name of the target instance to go to the Basic Information page of the instance.
3. In the left-side navigation pane, choose Monitoring and Alerts > **Instance Monitoring**.

If the CPU utilization **exceeds 90%** or **remains above 80%**, the DRDS instance has a performance bottleneck. If the ApsaraDB RDS for MySQL instance has no performance bottleneck, the current DRDS instance specifications cannot meet the QPS performance requirements of the business. In this case, upgrade the DRDS instance.

For more performance-related service monitoring scenarios and methods for configuring the DRDS CPU utilization alert, see DRDS instance monitoring.

Upgrade DRDS instances

QPS is an important metric for determining whether the DRDS instance specifications can meet the business requirements. A reference QPS value is available for each DRDS instance specification.

Note: Some special SQL statements require more computing (such as temporary table sorting and aggregate computing) on DRDS instances. In this case, the QPS supported by each DRDS instance is lower than the standard value in the specification.

You can upgrade the DRDS instance by adding DRDS server nodes for sharing the QPS load to improve the processing performance of the instance. DRDS server nodes are stateless. Therefore, this upgrade method increases the instance performance linearly.

For example, service A requires about 15,000 QPS. The current DRDS instance has 4 cores and 4 GB memory, with only two DRDS server nodes. The instance supports only 10,000 QPS. You can find that the CPU utilization of the DRDS instance is always high. After you upgrade the DRDS instance to the specification of 8 cores and 8 GB memory, each DRDS server node can handle about 4,000 QPS. Then, the performance meets the needs of the user, and the CPU utilization also drops to a reasonable level, as shown in the following figure:

For more information about how to upgrade a DRDS instance, see Change configurations.

Choose a database connection pool for an application

You can use a database connection pool to manage database connections in a centralized manner, so as to improve application performance and reduce database loads.

Reuse resources: Connections can be reused to avoid the high performance overheads caused by frequent connection creations and releases. Resource reuse can also improve system stability.

Improve the system response efficiency: After the connection initialization is completed, all

requests can directly use the existing connections, which avoids the overheads of connection initialization and release and improves the system response efficiency.

Prevent connection leakage: The connection pool forcibly revokes connections based on the preset revocation policy to prevent connection resource leakage.

Recommended connection pool

We recommend that you use a connection pool to connect applications and databases for service operations. For Java programs, we recommend that you use Druid connection pool. The Druid connection pool version is required to be 1.1.11 or later.

Standard Spring configuration for a Druid connection pool

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">
<property name="driverClassName" value="com.mysql.jdbc.Driver" />
<! -- Basic properties URL, user, and password -->
<property name="url" value="jdbc:mysql://ip:port/db?autoReconnect=true&rewriteBatchedStatements=true&socketTimeout=30000&connectTimeout=3000" />
<property name="username" value="root" />
<property name="password" value="123456" />
<! -- Configure the initial size, minimum value, and maximum value -->
<property name="maxActive" value="20" />
<property name="initialSize" value="3" />
<property name="minIdle" value="3" />
<! -- maxWait indicates the time-out period for getting the connection -->
<property name="maxWait" value="60000" />

<! -- timeBetweenEvictionRunsMillis indicates the interval for detecting idle connections to be closed, in milliseconds -->
<property name="timeBetweenEvictionRunsMillis" value="60000" />
<! -- minEvictableIdleTimeMillis indicates the minimum idle time of a connection in the connection pool, in milliseconds-->
<property name="minEvictableIdleTimeMillis" value="300000" />
<! -- SQL statement used to check whether a connection is available -->
<property name="validationQuery" value="select 'z' from dual" />
<! -- Whether to enable idle connection detection -->
<property name="testWhileIdle" value="true" />
<! -- Whether to check the connection status before getting a connection -->
<property name="testOnBorrow" value="false" />
<! -- Whether to check the connection status before releasing a connection -->
<property name="testOnReturn" value="false" />
<! -- Whether to close the connection at a specified time. This parameter is not required by default. However, you can add this parameter to balance the number of connections on each DRDS server node. -->
<property name="phyTimeoutMillis" value="1800000" />
<! -- Whether to close the connection after a specified number of SQL executions. This parameter is not required by default. However, you can add this parameter to balance the number of connections on each DRDS server node. -->
```

```
<property name="phyMaxUseCount" value="10000" />
</bean>
```

Connections in a DRDS instance

When an application connects to a Distributed Relational Database Service (DRDS) instance for operation, there are two types of connections from the perspective of the DRDS instance:

Frontend connection: a connection established by an application to the logical database on the DRDS instance.

Backend connection: a connection established by a DRDS server node in a DRDS instance to a physical database in a backend ApsaraDB RDS for MySQL instance.

Frontend connections

Theoretically, the number of frontend connections is limited by the available memory size and the number of network connections to the server nodes of a DRDS instance. However, when an application connects to a DRDS instance, the DRDS instance usually manages a limited number of connections to perform requested operations, and does not maintain a large number (tens of thousands, for example) of concurrent persistent connections. Therefore, the number of frontend connections allowed by a DRDS instance can be considered as unlimited.

The number of frontend connections is unlimited and a large number of idle connections are allowed. Therefore, this connection type is applicable to the scenarios where a large number of servers are deployed and their simultaneous connections to the DRDS instance are required.

Note: Operation requests obtained from frontend connections are processed by internal threads of the DRDS instance through backend connections, and the numbers of internal threads and backend connections are limited. Therefore, though the number of frontend connections is considered as unlimited, the overall concurrency of requests supported by a DRDS instance is limited.

Backend connections

Each server node of a DRDS instance creates a backend connection pool to automatically manage and maintain the backend connections to the physical databases in the ApsaraDB RDS for MySQL instance. Therefore, the maximum number of connections in each backend connection pool of a DRDS instance depends on the maximum number of connections supported by an ApsaraDB RDS for MySQL instance. The maximum number of connections in a backend connection pool of a DRDS

instance can be calculated using the following formula:

```
Maximum number of connections in a backend connection pool of a DRDS instance = FLOOR(Maximum number of connections in an ApsaraDB RDS for MySQL instance/Number of physical database shards in the ApsaraDB RDS for MySQL instance/Number of server nodes in the DRDS instance)
```

For example, you have purchased an ApsaraDB RDS for MySQL instance and a DRDS instance of the following specifications:

- The ApsaraDB RDS for MySQL instance has 4 cores and 16 GB memory. It has eight physical database shards and supports a maximum of 4,000 connections.
- The DRDS dedicated instance has 32 cores and 32 GB memory, with each DRDS server node having two cores and 2 GB memory. That is, the instance has 16 DRDS server nodes.

With the preceding formula, the maximum number of connections in each backend connection pool of the DRDS instance is:

```
Maximum number of connections in the backend connection pool of the DRDS instance = FLOOR (4000/8/16) = FLOOR (31.25) = 31
```

Note:

The result of the preceding formula is the maximum number of connections in the backend connection pool of the DRDS instance. In practice, the maximum number of connections in each backend connection pool of a DRDS instance is controlled below the upper limit to alleviate pressure on ApsaraDB RDS for MySQL instances.

We recommend that you create databases of a DRDS instance on dedicated ApsaraDB RDS for MySQL instances. Do not create databases for other applications or DRDS instances on the dedicated ApsaraDB RDS for MySQL instances.

Troubleshoot DDL exceptions

About DDL

When you execute a DDL command of Distributed Relational Database Service (DRDS), DRDS performs the corresponding DDL operation on all table shards. Failures of such operations can be divided into two types:

- The DDL operation execution fails in a database shard. DDL execution failure in any database shard may result in inconsistent table shard structures.
- The system does not respond for a long time after the DDL statement is executed. When you perform a DDL operation on a large table, the system may make no response for a long time due to the long execution of a DDL statement in a database shard.

Execution failures in database shards may occur for various reasons. For example, the table you want to create already exists, the column you want to add already exists, or the disk space is insufficient.

In general, no response for a long time is caused by the long execution of a DDL statement in a database shard. Taking ApsaraDB RDS for MySQL as an example, the DDL execution time depends mostly on whether the operation is an In-Place (directly modifying the source table) or Copy Table (copying data in the table) operation. An In-Place operation only modifies metadata in a table, whereas a Copy Table operation reconstructs data in the entire table and also involves log and buffer operations.

For more information on the relations between DDL operations and the two types of failures, see Summary of Online Status for DDL Operations.

To determine whether a DDL operation is an In-Place or Copy Table operation, you can view the return value of “rows affected” after the operation is completed.

Examples

Modify the default value of a column. This operation is very fast and does not affect the table data at all:

Query OK, 0 rows affected (0.07 sec)

Add an index. This operation takes some time, but “0 rows affected” indicates that the table data is not copied:

Query OK, 0 rows affected (21.42 sec)

Modify the data type of a column. This operation takes a long time and reconstructs all data rows in the table:

Query OK, 1671168 rows affected (1 min 35.54 sec)

Therefore, before you executing a DDL operation on a large table, perform the following steps to determine whether the operation is a fast or a slow one:

1. Copy the table structure to generate a cloned table.
2. Insert some data.

3. Perform the DDL operation on the cloned table.
4. Check whether the value of rows affected is 0 after the operation is completed. A non-zero value means that this operation reconstructs the entire table. In this case, you need to perform this operation in off-peak hours.

Handle failures

DRDS DDL operations distribute all SQL statements to all database shards for parallel execution. Execution failure on any database shard does not affect the execution on other database shards. In addition, DRDS provides the CHECK TABLE command to check the structure consistency of the table shards. Therefore, failed DDL operations can be performed again, and errors reported on database shards on which the operations have been successfully executed do not affect the execution on other database shards. Make sure that all table shards ultimately have the same structure.

Procedure for handling DDL failures

1. Run the **CHECK TABLE** command to check the table structure. If the returned result contains only one row and the status is normal, **the table statuses are consistent**. In this case, go to step 2. Otherwise, go to step 3.
2. Run the **SHOW CREATE TABLE** command to check the table structure. If the displayed table structure is the same as the expected structure after the DDL statement is executed, the **DDL statement is executed successfully**. Otherwise, go to step 3.
3. Run the **SHOW PROCESSLIST** command to check the statuses of all SQL statements being executed. If any ongoing DDL operations are detected, wait until these operations are completed, and then perform steps 1 and 2 to check the table structure. Otherwise, go to step 4.
4. Perform the DDL operation again on DRDS. If the **Lock conflict** error is reported, go to step 5. Otherwise, go to step 3.
5. Run the **RELEASE DBLOCK** command to release the DDL operation lock, and then go to step 4.

The procedure is as follows:

Check the table structure consistency
Run the CHECK TABLE command to check the table structure. When the returned result contains only one row and the displayed status is OK, **the table structures are consistent.

```
mysql> check table `xxxx`;
+---------------------------+-------+----------+----------+
| TABLE | OP | MSG_TYPE | MSG_TEXT |
+---------------------------+-------+----------+----------+
| TDDL5_APP.xxxx | check | status | OK |
+---------------------------+-------+----------+----------+
1 row in set (0.05 sec)
```

If no result is returned after you run CHECK TABLE on Data Management Service (DMS), try again by using the CLI.

Check the table structure

Run the SHOW CREATE TABLE command to check the table structure. If **table structures are consistent** and **correct**, the DDL statement has been executed successfully.

```
mysql> show create table `xxxx`;
+---------+------------------------------------------------------------------------------------------------------------
--------+
| Table | Create Table |
+---------+------------------------------------------------------------------------------------------------------------
--------+
| xxxx | CREATE TABLE `xxxx` (
`id` int(11) NOT NULL DEFAULT '0',
`NAME` varchar(1024) NOT NULL DEFAULT '',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 dbpartition by hash(`id`) tbpartition by hash(`id`) tbpartitions 3
|
+---------+------------------------------------------------------------------------------------------------------------
--------+
1 row in set (0.05 sec)
```

Check the SQL statements being executed

If some DDL executions are slow and no response is received for a long time, you can run the SHOW PROCESSLIST command to check the statuses of all SQL statements being executed.

```
mysql> SHOW PROCESSLIST WHERE COMMAND != 'Sleep';
+--------------+----------+------------------+-----------+---------+------------------------------------
--------------------------------+-------------------------------------------------------------------------------
--------------------+----------+-------------+----------+
| ID | USER | DB | COMMAND | TIME | STATE | INFO | ROWS_SENT | ROWS_EXAMINED | ROWS_READ |
+--------------+----------+------------------+-----------+---------+------------------------------------
--------------------------------+-------------------------------------------------------------------------------
--------------------+----------+-------------+----------+
| 0-0-352724126 | ifisibhk0 | test_123_wvvp_0000 | Query | 15 | Sending data | /*DRDS
/42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,SUM(`t_item`.`price`) from `t_i | NULL | NULL
| NULL |
| 0-0-352864311 | cowxhthg0 | NULL | Binlog Dump | 13 | Master has sent all binlog to slave; waiting for
binlog to be updated | NULL | NULL | NULL | NULL |
| 0-0-402714566 | ifisibhk0 | test_123_wvvp_0005 | Query | 14 | Sending data | /*DRDS
/42.120.74.88/ac47e5a72801000/ */select `t_item`.`detail_url`,`t_item`.`price` from `t_i | NULL | NULL |
NULL |
| 0-0-402714795 | ifisibhk0 | test_123_wvvp_0005 | Alter | 114 | Sending data | /*DRDS
/42.120.74.88/ac47e5a72801000/ */ALTER TABLE `Persons` ADD `Birthday` date | NULL | NULL | NULL |
......
+--------------+----------+------------------+-----------+---------+------------------------------------
--------------------------------+-------------------------------------------------------------------------------
```

```
----------------------+-----------+---------------+-----------+
12 rows in set (0.03 sec)
```

The value in the TIME column indicates the number of seconds that the command has been run for. If a command execution is too slow, as shown in the figure, you can run the **KILL '0-0-402714795'** command to cancel the slow command.

> In DRDS, one logical SQL statement corresponds to multiple statements on database shards. Therefore, you may need to kill multiple commands to stop a logical DDL statement. You can determine the logical SQL statement to which the command belongs based on the INFO column in the SHOW PROCESSLIST result set.

Handle lock conflict errors
DRDS adds a database lock before performing a DDL operation and releases the lock after the operation. The KILL DDL operation may cause a failure to release the lock. If you perform the DDL operation again, the following error message is returned:

```
Lock conflict , maybe last DDL is still running
```

Then, run **RELEASE DBLOCK** to release the lock. After the command is canceled and the lock is released, execute the DDL statement again during off-peak hours or when the service is stopped.

Other problems

DMS or other clients cannot display the modified table structure.
To enable some clients to obtain table structures from system tables (such as COLUMNS or TABLES), DRDS creates a shadow database in database shard 0 on your ApsaraDB RDS for MySQL instance. The shadow database name must be the same as the name of your DRDS logical database. It stores the table structures and other information of all the user databases.
DMS obtains DRDS table structures from the system table in the shadow database. When you are troubleshooting DDL exceptions, the table structure may be modified in the user database but not in the shadow database due to some reasons. In this case, you need to connect to the shadow database and perform the DDL operation on the table again in the database.

> CHECK TABLE does not check whether the table structure in the shadow database is consistent with that in the user database.

Scan DRDS data efficiently

Distributed Relational Database Service (DRDS) supports efficient data scanning and uses aggregate functions for statistical summary during full table scans.

Common scanning scenarios include the following:

Scan of table without database or table shards: DRDS transmits the original SQL statement to the backend ApsaraDB RDS for MySQL database for execution. In this case, DRDS supports any aggregate functions.

Non-full table scan: DRDS transmits the original SQL statement to each single ApsaraDB RDS for MySQL database for execution. For example, when the shard key in a WHERE clause indicates “equal to”, a non-full table scan is performed. In this case, DRDS also supports all aggregate functions.

Full table scan: Currently, supported aggregate functions are COUNT, MAX, MIN, and SUM. In addition, LIKE, ORDER BY, LIMIT, and GROUP BY are also supported during full table scan.

Parallel scan of all table shards: If you need to export data from all databases, you can run the SHOW command to view the table topology and scan all table shards in parallel. For more information, see the following.

Traverse tables by using a hint

1. Run SHOW TOPOLOGY FROM TABLE_NAME to obtain the table topology.

```
mysql:> SHOW TOPOLOGY FROM DRDS_USERS;
+------+------------------+--------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+------------------+--------------+
| 0 | DRDS_00_RDS | drds_users |
| 1 | DRDS_01_RDS | drds_users |
+------+------------------+--------------+
2 rows in set (0.06 sec)
```

By default, the non-sharding tables are stored in database shard 0.

1. Traverse each table based on the topology.

   Execute the current SQL statement on database shard 0.

```
/! TDDL:node='DRDS_00_RDS'*/ SELECT * FROM DRDS_USERS;
```

Execute the current SQL statement on database shard 1.

```
/! TDDL:node='DRDS_01_RDS'*/ SELECT * FROM DRDS_USERS;
```

Note: We recommend that you execute **SHOW TOPOLOGY FROM TABLE_NAME** to obtain the latest table topology before each scanning operation.

Parallel scans

DRDS allows you to run mysqldump to export data. However, if you want to scan data faster, you can enable multiple sessions for each table shard to scan tables in parallel.

```
mysql> SHOW TOPOLOGY FROM LJLTEST;
+------+----------------+------------+
| ID | GROUP_NAME | TABLE_NAME |
+------+----------------+------------+
| 0 | TDDL5_00_GROUP | ljltest_00 |
| 1 | TDDL5_00_GROUP | ljltest_01 |
| 2 | TDDL5_00_GROUP | ljltest_02 |
| 3 | TDDL5_01_GROUP | ljltest_03 |
| 4 | TDDL5_01_GROUP | ljltest_04 |
| 5 | TDDL5_01_GROUP | ljltest_05 |
| 6 | TDDL5_02_GROUP | ljltest_06 |
| 7 | TDDL5_02_GROUP | ljltest_07 |
| 8 | TDDL5_02_GROUP | ljltest_08 |
| 9 | TDDL5_03_GROUP | ljltest_09 |
| 10 | TDDL5_03_GROUP | ljltest_10 |
| 11 | TDDL5_03_GROUP | ljltest_11 |
+------+----------------+------------+
12 rows in set (0.06 sec)
```

As shown above, the table has four database shards, and each database shard has three table shards. Execute the following SQL statement on the table shards of the TDDL5_00_GROUP database.

```
/!TDDL:node='TDDL5_00_GROUP'*/ select * from ljltest_00;
```

Note: TDDL5_00_GROUP in HINT corresponds to the GROUP_NAME column in the output of the SHOW TOPOLOGY command. In addition, the table name in the SQL statement is the table shard name.

At this time, you can enable up to 12 sessions (corresponding to 12 table shards respectively) to process data in parallel.