Distributed Relational Database Service

Best Practice

Best Practice

Data Splitting Strategy

Sharding dimension is the most important factor to decide the distributed data. For this reason, it shall be selected with caution. In general, you can consider it based on these five dimensions: data balance, transaction boundary factor, common query efficiency, heterogeneous index, and simple strategy. The business type may be different for each application, and may fail to match the optimal strategy of all factors. Specifically, a business type shall be subject to comprehensive consideration. All factors are not allowed in extreme conditions to avoid a sharp increase of costs arising from development, operation and maintenance.

Capacity and access balance

Generally, the data capacity and access balance needs to be first considered. Unbalanced data distribution and access may fail to fulfill the data splitting capacity, worsening the access experience and increasing the cost. This equals the effect of 1+1<2. Therefore, data distribution and access is more balanced upon great discrimination of the split field. However, the hot issue shall also be considered for a split value.

According to the preceding principle, can data be split by the primary key (subject to the greatest discrimination)? We do not forbid splitting data by the primary key, but do not recommend doing so. Each sql needs to contain the primary key field to keep the best performance if you split data by the primary key (Certainly, your sql can fail to contain data fragments, only deteriorating the performance).

Transaction boundary

The bigger a transaction boundary (or the number of data fragments executed by single sql) is, the higher the probability of lock conflict of a system is. Consequently, the system is more difficult to expand, deteriorating its performance. To achieve a highly-expandable system, you need to make best efforts to narrow the transaction boundary and limit it to a single computer if possible. Three methods are available to narrow the transaction boundary, and they are described as follows:

Method 1: Ensure that the transaction boundary is small by nature.

For example, if the data are distributed evenly as per a splitting condition, and if the transaction boundary lies only on a machine and does not involve multi-machine transactions, the sharding

condition is an applicable sharding dimension.

Method 2: Use an eventually message-based consistent model to change a strongly consistent transaction into an eventually consistent transaction.

Splitting a transaction involves a complicated concept, and it will be addressed separately. A brief description is presented here only for the model of splitting a most commonly and eventually consistent transaction. For instance, when transferring data between two databases through a distributed transaction, we will find that some operations have to be transmitted over the Internet. As a result, a transaction is greatly delayed, dramatically deteriorating the performance.

Method 3: Exercise caution when using distributed transactions.

An eventually consistent transaction generally fulfills 90% business scenarios. Other scenarios may still be fulfilled in compliance with distributed transactions. However, distributed transactions will cause many performance issues. For this reason, it is recommended to use distributed transactions when required.

Common query

The core idea for optimizing common query is to directly and physically send a front-end request to a storage machine rather than send it to multiple storage machines for query if possible.

When being sent to multiple machines for query, a request will be physically sent to a large number of storage machines although the query is not delayed at each time. This occupies additional resources on lower-layer data nodes. Therefore, this circumstance shall be avoided if possible.

Heterogeneous index

The preceding example shows the method for querying data only based on one dimension. How can we query data if several key dimensions are available for query in the system?

The first method

The first method is ful table scan, increasing the duty of reading more data. However, we can expand the read capacity infinitely by horizontally adding a standby database. Therefore, full table scan is a feasible scheme at a slightly high cost.

The second method

The second method is to use the heterogeneous index table if we want to further decrease the cost. The essence is to use an asynchronous trigger and write each update of the original table into a new table in compliance with another dimension. When being familiar with a database, you map the database. The heterogeneous index table basically functions as an index in the traditional database, excepting that the index creation process is changed from synchronization to asynchronization, and about 100ms delay may exist between the index table and the main table.

Keep it simple

Various conflicts need to be addressed. In practice, the shard key is generally selected based on different attractions. Scheme A will bring some advantages, whereas scheme B will bring other advantages.

If query optimization conflicts with balanced read-write access, take priority to select the balanced read-write access. Query-related issues are easy to solve regardless of full table scan or replication of heterogeneous index with an additional machine. However, if the write operation or the stand-alone capacity is not balanced, the conflict will be more difficult to solve.

The complicated sharding rule or opportunistic program code can bring short-term favorable performance or cost of the system. However, the resultant complexity for operation and maintenance will offset most advantages obtained in the system. Therefore, a simple and direct method based on the 82 Rule is often the most efficient method according to the system architecture.

Selection of application connection pool

It is highly recommended that you connect a task with a connection pool. In the Java environment, DRUID serves as an application connection pool (https://github.com/alibaba/druid/), and is the most recommended connection pool. The components in the DRUID connection pool are well-tested and stable standard components released by Alibaba Corporation. They also implement monitoring and other functions.

Spring standard configuration of DRUID

```
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-
method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!—essential attribute url, user, password -->
        <property name="url"
value="jdbc:mysql://ip:port/db?autoReconnect=true&amp;rewriteBatchedStatements=true&amp;socketTimeout=3
0000&amp;connectTimeout=3000" />
        <property name="username" value="root" />
        <property name="password" value="123456" />
        <!—configuration initialization size, minimum and maximum -->
        <property name="maxActive" value="20" />
        <property name="initialSize" value="3" />
        <property name="minIdle" value="3" />
        <!—maxWait timeout for waiting for obtaining of connection -->
        <property name="maxWait" value="60000" />

        <!—timeBetweenEvictionRunsMillis detection interval, idle connection to be closed for detection, with
millisecond as unit -->
        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <!—minEvictableIdleTimeMillis minimum idle time of a connection in pool, with millisecond as unit-->
```

```
    <property name="minEvictableIdleTimeMillis" value="300000" />
    <property name="validationQuery" value="SELECT 'z'" />
    <property name="testWhileIdle" value="true" />
    <property name="testOnBorrow" value="false" />
    <property name="testOnReturn" value="false" />
  </bean>
```

Data import and export

Import of small data

If any small data (at the ten-million level) do not need to be incrementally imported or need to be one-off imported for compatibility test, it is recommended to directly use the Mysql source or Navicat for single-thread transfer, or write data into DRDS by hand coding of multi-thread batch.

Single-thread data are imported slowly. This fails to take the advantages of the parallel system with highly-distributed databases, However, such data can be easily imported in practice. For example, you can use the mysql source command to import the data from http://www.blogjava.net/hh-lux/archive/2007/05/05/115419.html, or use the navicat command to import the data from xxx.sql or xxx.csv.

Import of big data

As DRDS can give play to the 100% read-write capacity of the underlying storage, so you do not have to worry about it. If you want to write performance authentication with your own data, we recommend you to write in batches (multi-thread reading at the same time has a better effect). If you use java, you can submit the interface of sql by using the batch in jdbc api. If you want your code to implement a high efficiency, you must add the parameter rewriteBatchedStatements=true (druid data source https://github.com/alibaba/druid is used as the code snippet)) into jdbc url. Mysql connector is allowed to merge multiple insert sentences into an insert sentence in multi values format, and send it to mysql server for execution. Furthermore, PrepareStatement is required to execute sql.

To analyze the specific principle, refer to the code example in http://www.cnblogs.com/xhan/p/3958521.html:

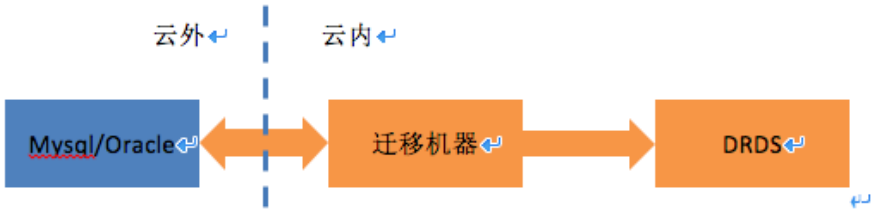
```
//Connection setting and creation
ds = new DruidDataSource();
ds.setUrl("jdbc:mysql://" + c.getHost() + ":" + c.getPort() + "/" + c.getSchema());
ds.setConnectionProperties("autoReconnect=true;socketTimeout=600000;rewriteBatchedStatements=true");
ds.setDriverClassName("com.mysql.jdbc.Driver");
ds.setUsername(c.getUser());
ds.setPassword(c.getPassword());
ds.setMaxActive(16);
ds.setMaxWait(5000);
```

```
ds.init();

//Data import code
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class BatchedInsertDemo {
public static void doBatchedInsert(Connection conn, int batchSize, int insertCount) throws SQLException {
    PreparedStatement ps = conn.prepareStatement("insert into Test (name,gmt_created,gmt_modified) values
(?,now(),now())");
    for (int i = 0; i < insertCount; i++) {
        ps.setString(1, i+" ");
        ps.addBatch();
        if((i+1) % batchSize == 0) {
          ps.executeBatch();
        }
     }
     ps.executeBatch();
     ps.close();
     }
}
```

Support of DRDS for data import

For formal on-line operation of application, we have a special tool for data migration. In this case, one or more migration machines shall be prepared. The following two methods are available for deploying these machines, depending on the network situation of the user’ s database.



数据导入场景 1，云内机器可以访问用户数据库

数据导入场景 2，用户环境只能单向接入云环境

1） if user’ s database can be directly accessed from the inside cloud, the ECS is recommended as the mitigation machine. As data are transmitted from the ECS to the DRDS over an intranet with high bandwidth, and the bandwidth of the public network of the ECS can be elastically upgraded, data mitigation efficiency can implement the best. In order to reduce cloud loading cost, we have arranged some ECS dedicated to ECS, and additional preparation is not required in general.

2) In some cases, for safety or other aspects, a user can access Aliyun over its network, but the machine in Aliyun environment cannot access the user’ s network. In this case, the mitigation procedure shall be arranged on the user's machine for data mitigation.

Data export

The DRDS supports data exported (mysqldump command). It can also skip to the console of the data node RDS from the DRDS console for database backup and backup file downloading. See http://help.aliyun.com/view/13440586.html for specific steps

Globally unique ID

The globally unique ID of DRDS ensures that the unique filed data is exclusively defined rather than being strictly increased. Therefore, operations such as sequencing cannot be performed according to the sequence.

Automatic filling of the primary key

The current DREDS is capable of supporting automatic filling of the primary key. This means that DRDS will automatically fill the primary key only if you do not specify the primary key in the insert statement. An example is shown as follows:

```
mysql> insert into users (name,address,gmt_create,gmt_modified,intro) values ('sun','hz',now(),now(),'aa');
Query OK, 1 row affected (0.02 sec)

mysql> select last_insert_id();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|             5018 |
+------------------+
1 row in set (0.00 sec)
```

You can also use the standard jdbc interface to obtain LastInsertId, which is consistent with the ordinary MySQL

Manual creation and obtaining

You can create a sequence through the specified SQL of DRDS, with the syntax being as follows:

```
mysql> create sequence sample_seq start with 100000;
Query OK, 1 row affected (0.01 sec)
```

You can view the currently existing sequence through the specified SQL of DRDS, with the syntax being as follows:

```
mysql> show sequences;
+------+-------------------------------+--------+---------------------+
| ID   | NAME                          | VALUE  | GMT_MODIFIED        |
+------+-------------------------------+--------+---------------------+
|    1 | AUTO_SEQ_USERS                |   5000 | 2014-08-09 18:20:28 |
|    2 | AUTO_SEQ_ITEMS                |   5000 | 2014-06-18 16:08:06 |
|    3 | AUTO_SEQ_ORDERS               |   6000 | 2014-06-18 16:08:06 |
|    4 | demo_seq                      |  12000 | 2014-08-09 18:21:28 |
|    5 | AUTO_SEQ_DEF                  |   2000 | 2014-07-04 17:55:36 |
|    6 | WEBO_COMMENTS                 |      0 | 2014-07-13 19:38:55 |
|    7 | AUTO_SEQ_ABC                  |  66000 | 2014-07-16 15:10:43 |
|    8 | AUTO_SEQ_CATEGORY             |   2000 | 2014-07-17 15:53:27 |
|    9 | AUTO_SEQ_USER                 |   2000 | 2014-07-25 16:54:29 |
|   10 | AUTO_SEQ_CLUSTER              |   2000 | 2014-07-27 15:58:34 |
|   11 | AUTO_SEQ_SPE_CLUSTER          |   2000 | 2014-07-28 16:21:30 |
|   12 | AUTO_SEQ_A                    |   2000 | 2014-07-28 16:22:08 |
+------+-------------------------------+--------+---------------------+
17 rows in set (0.04 sec)
```

You can obtain the latest ID through the specified SQL of DRDS, with the syntax being as follows:

```
select sample_seq.nextVal from dual;
+--------------------+
| SAMPLE_SEQ.NEXTVAL |
+--------------------+
|             101001 |
+--------------------+
1 row in set (0.04 sec)
```

SQL optimization

The DRDS is an efficient and stable distributed relation database system. However, as distributed relation query is processed by DRDS, its query optimization for SQL is different with that of the traditional single database (e.g. mysql, oracle). When in query optimization, the traditional single database mainly considers the expense of disk IO, but DRDS, during optimization, will also consider another more important IO expense, i.e. network. For optimizing SQL execution of DRDS, the core optimization idea is to reduce network IO. For this purpose, the DRDS will possibly distribute the work originally belonging to DRDS to each sub-database (e.g. RDS) at the bottommost layer, which can transfer IO expense originally needing network into stand-alone disk IO expense, thus increasing the execution efficiency of query. Therefore, if we encounter slow SQL when using DRDS, we shall properly rewrite SQL as per the characteristics of DRDS.

Condition optimization of SQL

As data of DRDS are horizontally sharded as per the split key, use of split key in query will be very meaningful to reduce execution time of SQL in DRDS. The query condition shall be possibly provided with a sub-database key, which can enable DRDS to directly route query to a specific sub-database according to the value of the sub-database key, which is useful to avoid whole database scan by DRDS. The higher the selectivity of the condition including the shard key is (or the higher the discrimination is), the easier it will be to increase DRDS’ s query speed. For example, equality query will be executed more quickly than range query.

JOIN optimization of SQL

In SQL, Join operation will often be the most time-consuming. Join algorithm used by DRDS in most cases is Nested Loop and its derived algorithm (if Join has a sequencing request, Sort Merge algorithm is used). DRDS’ s Join process based on Nested Loop algorithm is as follows: for left and right tables of Join, DRDS will fetch data from the left table (also called as driving table) of Join, and then put the value in the Column Join in data fetched into the right table for IN query so as to finish Join process. Thus, the less the data quantity of Join's left table is, the less the times of IN query made by DRDS for the right table is. The less the data volume of right table is or index is created, the quicker Join will be. Therefore, in DRDS, selection of Join’ s driving table is very important for optimization of Join.

Short table as driving table of Join

The so-called short table does not say that the table is the record number of the table in database, but the number of records returned after the table is subject to condition filtering in query. Thus, the most simple method for determining the actual data volume of a table is to attach where conditions and join on conditions related to the table, and put them in DRDS for a count (*) query independently to view data volume. For example, assuming that there is SQL shown as follows:

```
select t.title, t.price
from t_order o,
   ( select * from t_item i where i.id=242002396687 ) t
where  t.source_id=o.source_item_id and o.sellerId<1733635660;
```

Its query speed is very slow, shown as follows:

```
mysql> select t.title, t.price
    -> from
    ->     t_order o,
    ->     ( select * from t_item i where i.id=242002396687 ) t
    -> where  t.source_id=o.source_item_id and o.sellerId<1733635660
    -> ;
+--------------------------------------------------------+----------+
| title                                                  | price    |
+--------------------------------------------------------+----------+
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
| 木木兔女童皮鞋水钻豆豆鞋儿童童鞋韩版女大童公主单鞋新子款 | 239.00 |
+--------------------------------------------------------+----------+
10 rows in set (23.79 sec)
```

About 24 seconds are required. Seeing the SQL, it is an inner JOIN. We do not know the actual data volume of table o and table t in JOIN process, but we can conduct count () query on the table o and table t respectively to obtain the group of data. For table o, we observe that o.sellerId<173363560 in where conditions is only related to table o, we will extract it out, and attach into the count () query of table o, thus obtaining the following query results;

```
mysql> select count(*) from t_order o where o.sellerId<1733635660
+----------+
| count(*) |
+----------+
|   504018 |
+----------+
1 row in set (0.10 sec)
```

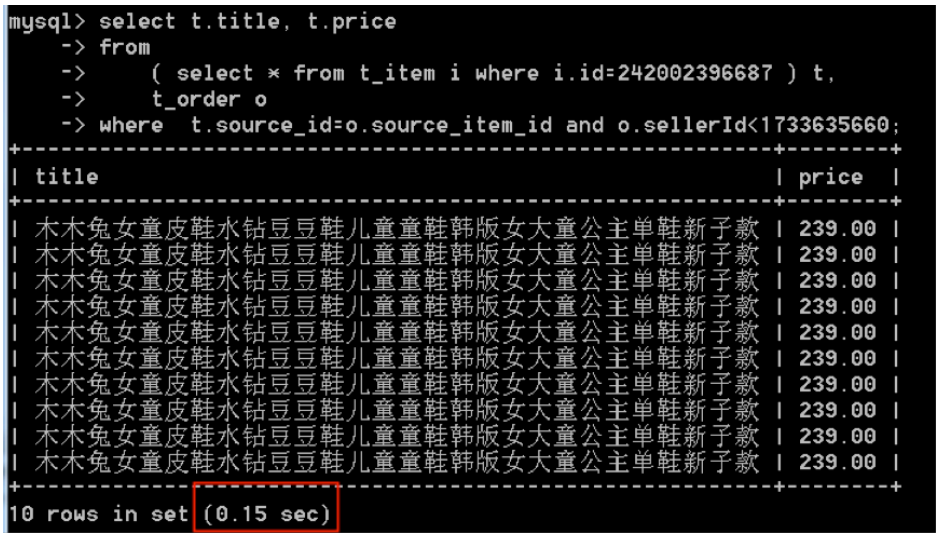
And then we may know that table o has 50W records. Similarly, for table t, as this is a sub-query, table t is extracted directly for count (*) query, then:

```
mysql> select count(*) from t_item i where i.id=242002396687;
+----------+
| count(*) |
+----------+
|        1 |
+----------+
1 row in set (0.01 sec)
```

We can know the data volume of table t is only one. Therefore, we can determine that table o is a long table and table t is a short table. Based on the principle of possibly using short table as Join driving table, we will adjust SQL into:

```
select t.title, t.price
from
( select * from t_item i where i.id=242002396687 ) t,
 t_order o
where  t.source_id=o.source_item_id and o.sellerId<1733635660
```

The query results are as follows:



The query time is reduced to 0.15 seconds from 24 seconds, with large increase. A broadcast table is used as the driving table of Join

broadcast table as driving table of Join

As the broadcast table of DRDS will be stored in every sub-database, the broadcast table, when being used as the driving table of Join, will be converted into stand-alone Join together with Join of other tables, thus increasing query performance. For example, assuming that there are SQL (where table t_area is broadcast table) shown as follows:

```
select t_area.name
from  t_item i join t_buyer b on i.sellerId=b.sellerId join t_area a on b.province=a.id
where  a.id < 110107
limit 0, 10
```

The three tables are used as JOIN, with query results as follows:

The execution time is fairly long, about 15 seconds. Now, we will adjust the sequence of join, and put

the broadcast table at the leftmost side as the driving table of join, that is:

```
select t_area.name
   from t_area a join t_buyer b on b.province=a.id join t_item i on i.sellerId=b.sellerId
   where  a.id < 110107
limit 0, 10
```

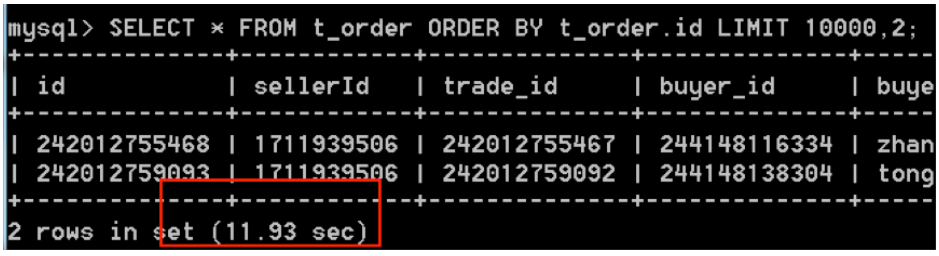
Then, the whole join will be pushed down to be stand-alone join in DRDS. We will observe the execution results SQL adjusted again:

Limit optimization of SQL

When DRDS is executing limit offset, count sentence, it actually reads the previous records of offset in order and discards them directly, which will result in very slow query when offset is very big, even though count is very small. Taking SQL as an example:

```
SELECT *
FROM t_order
ORDER BY t_order.id
LIMIT 10000,2
```

Although SQL only fetches 2 records (i.e.10000 and 10001), but its execution time is about 12 seconds, which is because that the number of record actually read by DRDS is 10002, as shown in figure below:

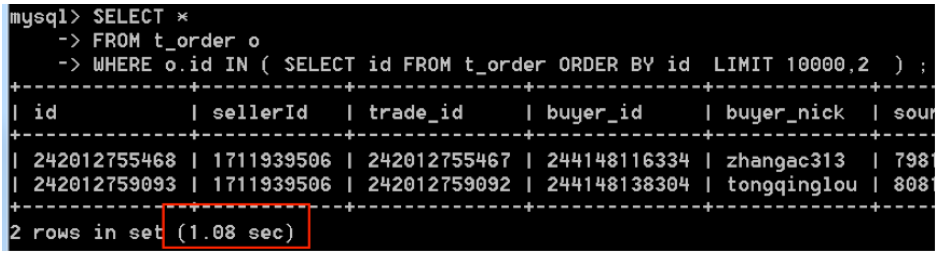


In light of the above conditions, the optimization direction of SQL is that: the ID set of SQL is checked firstly, and then the real record contents are queried through in query, with SQL rewritten shown as

follows:

```
SELECT *
FROM t_order o
WHERE o.id IN (
      SELECT id
      FROM t_order
      ORDER BY id
LIMIT 10000,2  )
```

The purpose of rewriting is to buffer ID (ID is not too much) with internal memory, so that disk IO will be reduced. If the sub-database key of table t_order is id, DRDS can route the in query to different sub-databases through rule calculation for query, which avoids whole database scan. We will observe the query results of SQL rewritten again



The execution time is changed into 1.08 seconds from the original 12 seconds, thus reducing an order of magnitude.

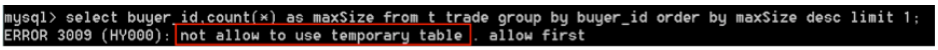
ORDER BY optimization of SQL

In DRDS, by default, ensure that column name behind Distinct, Group and Order By sentences are identical, and the final SQL only returns a small number of data. This is because that we, in this case, can minimize the network band width consumed in distributed query without fetching a large number of data for sequencing in table, and enable system performance to implement the optimal status.

For example, for SQL shown below:

```
select buyer_id,
      count(*) as maxSize
from t_trade
group by buyer_id
order by maxSize desc
limit 1
```

During execution, DRDS shall sequence data by buyer_id for aggregation, and then sequence the aggregation results by maxSize. Because of existence of two different sequencing requirements, DRDS cannot finish the SQL at one time if middle results are not preserved with the temporary table, thus during actual execution of the SQL, the following errors will be reported:



Error reporting contents prompt that the temporary is not allowed to use. DRDS supports use of temporary table, but does not recommend by default. As use of temporary table generally means that the system has a performance bottleneck, avoid use of temporary table by making column names of Distimct Group by and Order by identical possibly. If the SQL is executed successfully, DRDS can be told, by adding HINT in SQL, that use of temporary table is allowed, with SQL modified as follows (green part):

```
/*+TDDL({'extra':{'ALLOW_TEMPORARY_TABLE':'TRUE'}})*/
select buyer_id,
       count(*) as maxSize
from t_trade
group by buyer_id
order by maxSize desc
limit 1
```

In this way, the DRDS can execute the SQL successfully. However, as the HINT may be ignored as a note in some mysql clients, hint is generally added when SQL is sent to DRDS through the mysql connector. Note: Confirm that the temporary table can be used only when there is a small number of data in the temporary table. Otherwise, the system will have a serious performance problem ####.