大数据技术之 Hive

咖啡熊

版本：V4.8.3

第 1 章 Hive 入门

1.1 什么是 Hive

1）Hive 简介

Hive 是由 Facebook 开源，基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张表，并提供类 SQL 查询功能。

那为什么会有 Hive 呢？它是为了解决什么问题而诞生的呢？

下面通过一个案例，来快速了解一下 Hive。

例如：需求，统计单词出现个数。

（1）在 Hadoop 课程中我们用 MapReduce 程序实现的，当时需要写 Mapper、Reducer 和 Driver 三个类，并实现对应逻辑，相对繁琐。

```
test 表
id 列

atguigu
atguigu
ss
ss
jiao
banzhang
xue
hadoop
```

（2）如果通过 Hive SQL 实现，一行就搞定了，简单方便，容易理解。

```
select count(*) from test group by id;
```

2）Hive 本质

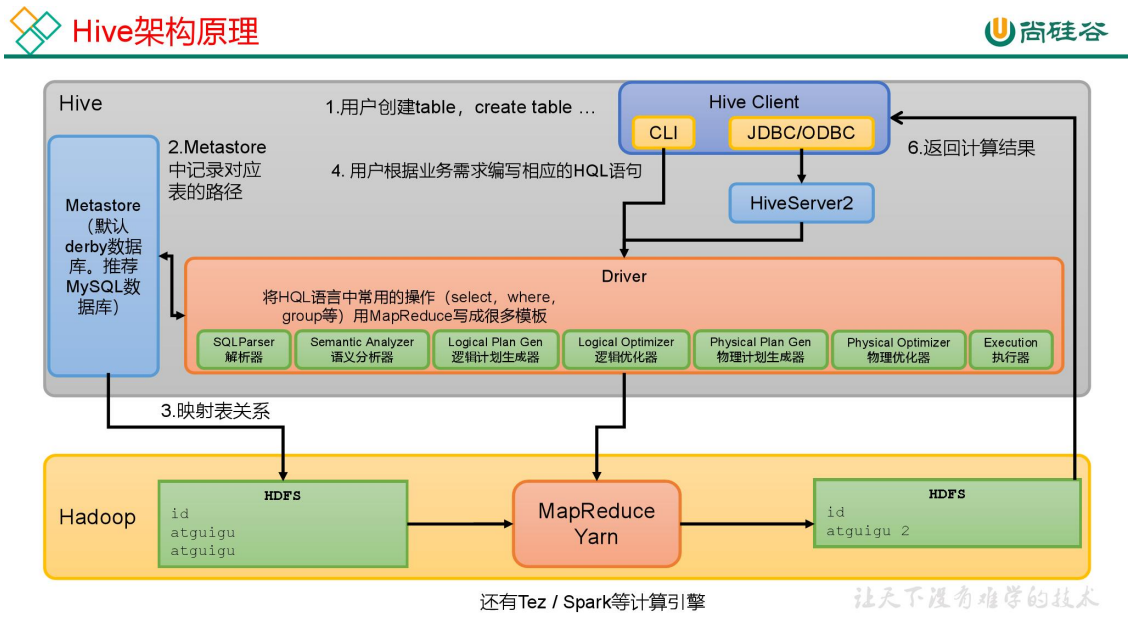
Hive 是一个 Hadoop 客户端，用于将 HQL（Hive SQL）转化成 MapReduce 程序。

（1）Hive 中每张表的数据存储在 HDFS

（2）Hive 分析数据底层的实现是 MapReduce（也可配置为 Spark 或者 Tez）

（3）执行程序运行在 Yarn 上

1.2 Hive 架构原理



1）用户接口：Client

CLI（command-line interface）、JDBC/ODBC。

说明：JDBC 和 ODBC 的区别。

（1）JDBC 的移植性比 ODBC 好；（通常情况下，安装完 ODBC 驱动程序之后，还需要经过确定的配置才能够应用。而不相同的配置在不相同数据库服务器之间不能够通用。所以，安装一次就需要再配置一次。JDBC 只需要选取适当的 JDBC 数据库驱动程序，就不需要额外的配置。在安装过程中，JDBC 数据库驱动程序会自己完成有关的配置。）

（2）两者使用的语言不同，JDBC 在 Java 编程时使用，ODBC 一般在 C/C++编程时使用。

2）元数据：Metastore

元数据包括：数据库（默认是 default）、表名、表的拥有者、列/分区字段、表的类型（是否是外部表）、表的数据所在目录等。

默认存储在自带的 derby 数据库中，由于 derby 数据库只支持单客户端访问，生产环境中为了多人开发，推荐使用 MySQL 存储 Metastore。

3）驱动器：Driver

（1）解析器（SQLParser）：将 SQL 字符串转换成抽象语法树（AST）

（2）语义分析（Semantic Analyzer）：将 AST 进一步划分为 QeuryBlock

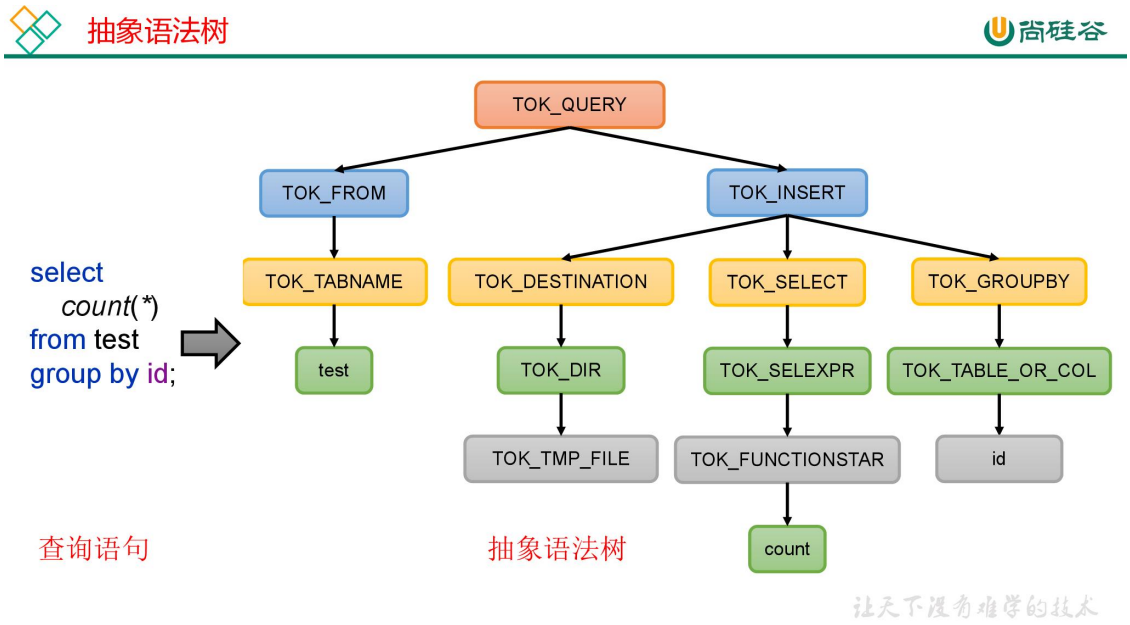
（3）逻辑计划生成器（Logical Plan Gen）：将语法树生成逻辑计划

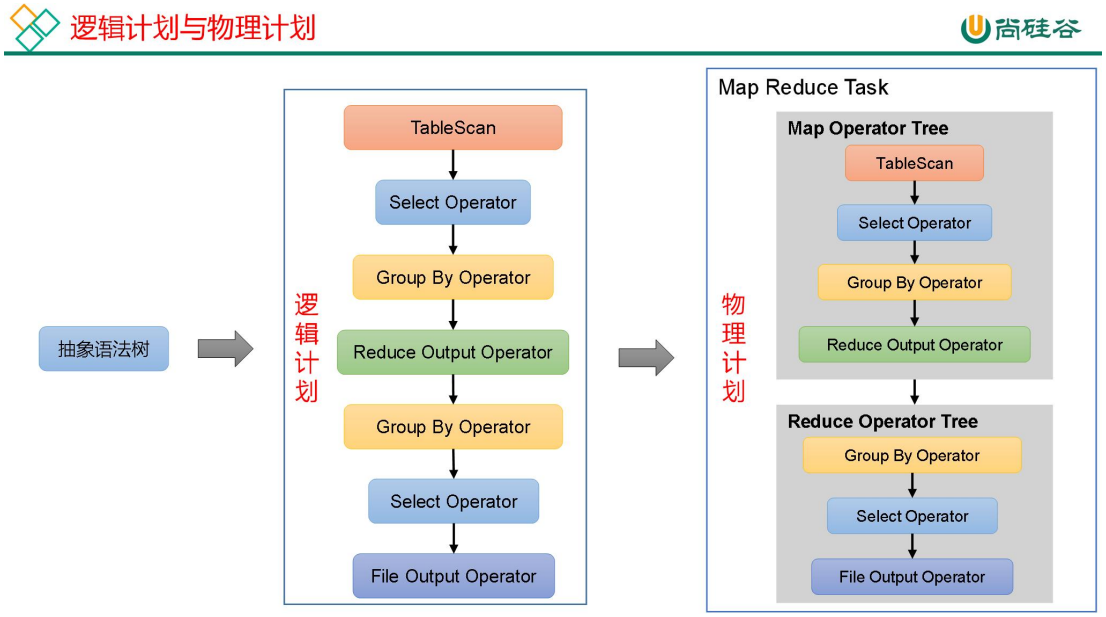
（4）逻辑优化器（Logical Optimizer）：对逻辑计划进行优化

（5）物理计划生成器（Physical Plan Gen）：根据优化后的逻辑计划生成物理计划

（6）物理优化器（Physical Optimizer）：对物理计划进行优化

（7）执行器（Execution）：执行该计划，得到查询结果并返回给客户端





4）Hadoop

使用 HDFS 进行存储，可以选择 MapReduce/Tez/Spark 进行计算。

第 2 章 Hive 安装

2.1 Hive 安装地址

1）Hive 官网地址：http://hive.apache.org/

2）文档查看地址：https://cwiki.apache.org/confluence/display/Hive/GettingStarted

3）下载地址：http://archive.apache.org/dist/hive/

4）github 地：https://github.com/apache/hive

2.2 Hive 安装部署

2.2.1 安装 Hive

1）把 apache-hive-3.1.3-bin.tar.gz 上传到 Linux 的/opt/software 目录下

2）解压 apache-hive-3.1.3-bin.tar.gz 到/opt/module/目录下面

```
[atguigu@hadoop102 software]$ tar -zxvf /opt/software/apache-hive-3.1.3-bin.tar.gz -C /opt/module/
```

3）修改 apache-hive-3.1.3-bin.tar.gz 的名称为 hive

```
[atguigu@hadoop102 software]$ mv /opt/module/apache-hive-3.1.3-bin/ /opt/module/hive
```

4）修改/etc/profile.d/my_env.sh，添加环境变量

```
[atguigu@hadoop102 software]$ sudo vim /etc/profile.d/my_env.sh
```

（1）添加内容

```
#HIVE_HOME
export HIVE_HOME=/opt/module/hive
export PATH=$PATH:$HIVE_HOME/bin
```

（2）source 一下

```
[atguigu@hadoop102 hive]$ source /etc/profile.d/my_env.sh
```

5）初始化元数据库（默认是 derby 数据库）

```
[atguigu@hadoop102 hive]$ bin/schematool -dbType derby -initSchema
```

2.2.2 使用 Hive

1）启动 Hive

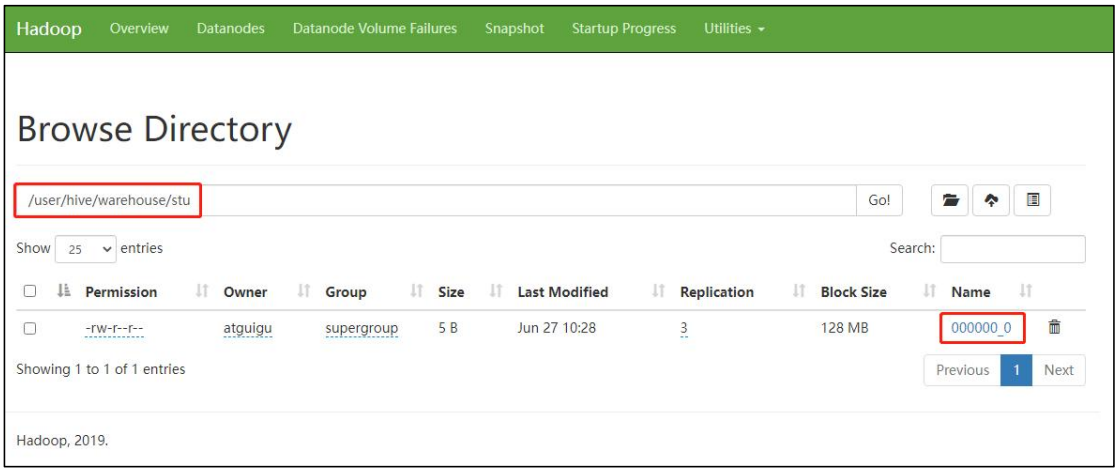
```
[atguigu@hadoop102 hive]$ bin/hive
```

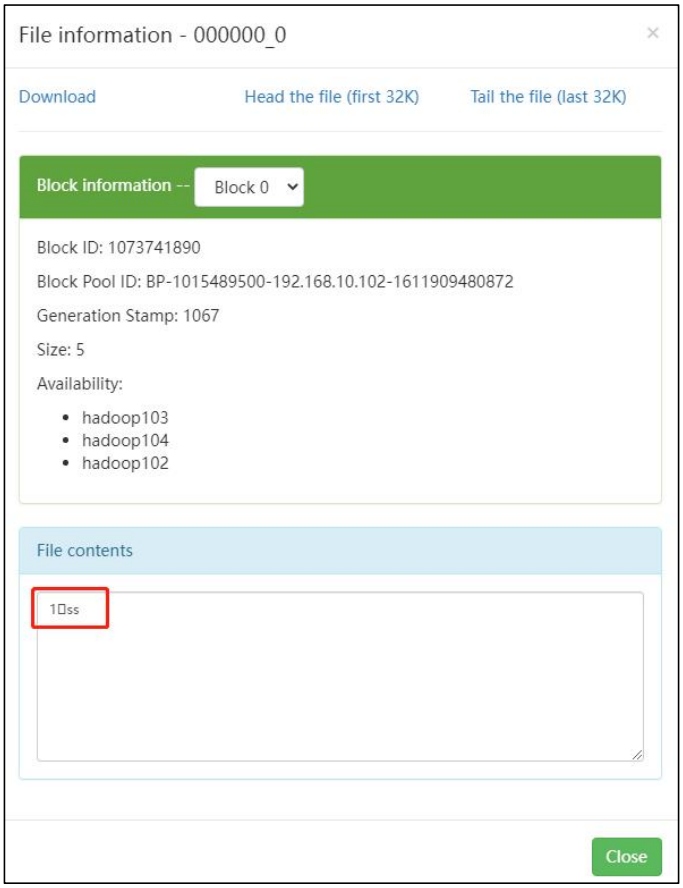
2）使用 Hive

```
hive> show databases;
hive> show tables;
hive> create table stu(id int, name string);
hive> insert into stu values(1,"ss");
hive> select * from stu;
```

观察 HDFS 的路径/user/hive/warehouse/stu，体会 Hive 与 Hadoop 之间的关系。

Hive 中的表在 Hadoop 中是目录；Hive 中的数据在 Hadoop 中是文件。



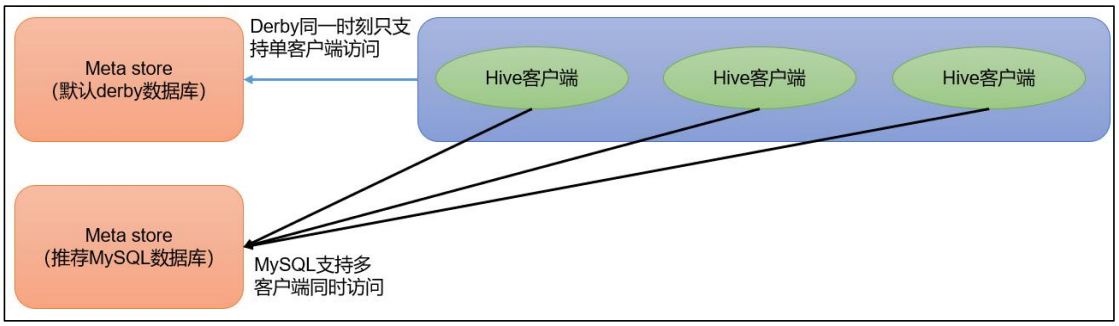


3）在 Xshell 窗口中开启另一个窗口开启 Hive，在/tmp/atguigu 目录下监控 hive.log 文件

```
[atguigu@hadoop102 atguigu]$ tail -f hive.log


Caused by: ERROR XSDB6: Another instance of Derby may have
already booted the database /opt/module/hive/metastore_db.
        at
org.apache.derby.iapi.error.StandardException.newException(Unknow
```

```
n Source)
        at
org.apache.derby.iapi.error.StandardException.newException(Unknow
n Source)
        at
org.apache.derby.impl.store.raw.data.BaseDataFileFactory.privGetJ
BMSLockOnDB(Unknown Source)
        at
org.apache.derby.impl.store.raw.data.BaseDataFileFactory.run(Unkn
own Source)
...
```

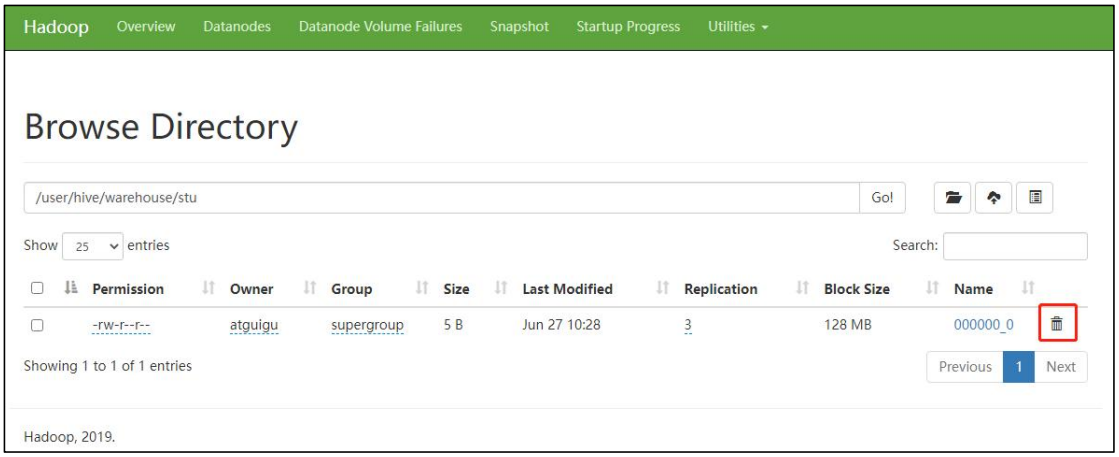
原因在于 Hive 默认使用的元数据库为 **derby**。**derby 数据库的特点是同一时间只允许一个客户端访问。如果多个 Hive 客户端同时访问，就会报错。**由于在企业开发中，都是多人协作开发，需要多客户端同时访问 Hive，怎么解决呢？我们可以将 Hive 的元数据改为用 MySQL 存储，MySQL 支持多客户端同时访问。

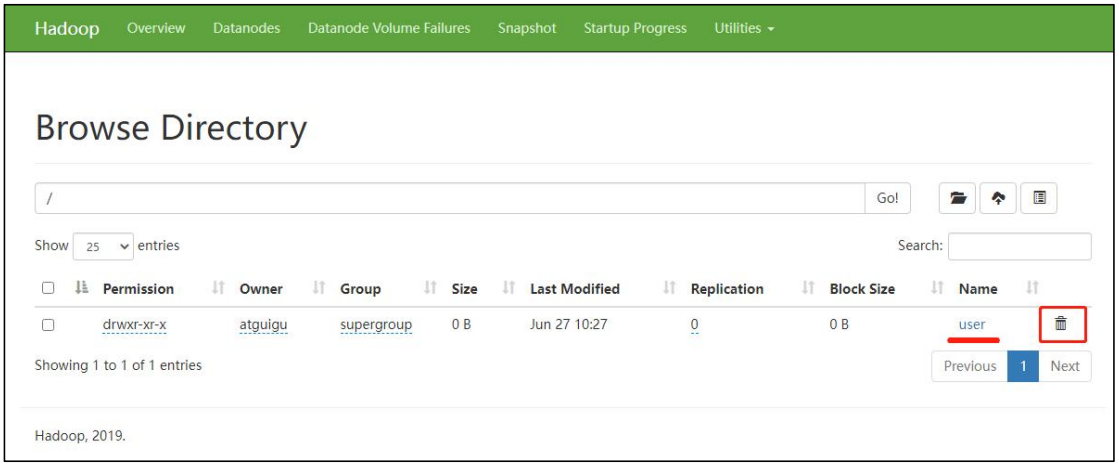


4）首先退出 hive 客户端。然后在 Hive 的安装目录下将 derby.log 和 metastore_db 删除，顺便将 HDFS 上目录删除

```
hive> quit;
[atguigu@hadoop102 hive]$ rm -rf derby.log metastore_db
[atguigu@hadoop102 hive]$ hadoop fs -rm -r /user
```

5）删除 HDFS 中/user/hive/warehouse/stu 中数据





2.3 MySQL 安装

2.3.1 安装 MySQL

1）上传 MySQL 安装包以及 MySQL 驱动 jar 包

```
mysql-5.7.28-1.el7.x86_64.rpm-bundle.tar
mysql-connector-java-5.1.37.jar
```

2）解压 MySQL 安装包

```
[atguigu@hadoop102 software]$ mkdir mysql_lib
[atguigu@hadoop102 software]$ tar -xf mysql-5.7.28-
1.el7.x86_64.rpm-bundle.tar -C mysql_lib/
```

3）卸载系统自带的 mariadb

```
[atguigu@hadoop102 ~]$ sudo rpm -qa | grep mariadb | xargs sudo
rpm -e --nodeps
```

4）安装 MySQL 依赖

```
[atguigu@hadoop102 software]$ cd mysql_lib
[atguigu@hadoop102 mysql_lib]$ sudo rpm -ivh mysql-community-
common-5.7.28-1.el7.x86_64.rpm
[atguigu@hadoop102 mysql_lib]$ sudo rpm -ivh mysql-community-
libs-5.7.28-1.el7.x86_64.rpm
[atguigu@hadoop102 mysql_lib]$ sudo rpm -ivh mysql-community-
libs-compat-5.7.28-1.el7.x86_64.rpm
```

5）安装 mysql-client

```
[atguigu@hadoop102 mysql_lib]$ sudo rpm -ivh mysql-community-
client-5.7.28-1.el7.x86_64.rpm
```

6）安装 mysql-server

```
[atguigu@hadoop102 mysql_lib]$ sudo rpm -ivh mysql-community-
server-5.7.28-1.el7.x86_64.rpm
```

注意：若出现以下错误

```
warning: 05_mysql-community-server-5.7.16-1.el7.x86_64.rpm:
Header V3 DSA/SHA1 Signature, key ID 5072e1f5: NOKEY
error: Failed dependencies:
libaio.so.1()(64bit) is needed by mysql-community-server-5.7.16-
```

```
1.el7.x86_64
```

解决办法：

```
[atguigu@hadoop102 software]$ sudo yum -y install libaio
```

7）启动 MySQL

```
[atguigu@hadoop102 software]$ sudo systemctl start mysqld
```

8）查看 MySQL 密码

```
[atguigu@hadoop102 software]$ sudo cat /var/log/mysqld.log | grep
password
```

2.3.2 配置 MySQL

配置主要是 root 用户 + 密码，在任何主机上都能登录 MySQL 数据库。

1）用刚刚查到的密码进入 MySQL（如果报错，给密码加单引号）

```
[atguigu@hadoop102 software]$ mysql -uroot -p'password'
```

2）设置复杂密码（由于 MySQL 密码策略，此密码必须足够复杂）

```
mysql> set password=password("Qs23=zs32");
```

3）更改 MySQL 密码策略

```
mysql> set global validate_password_policy=0;
mysql> set global validate_password_length=4;
```

4）设置简单好记的密码

```
mysql> set password=password("123456");
```

5）进入 MySQL 库

```
mysql> use mysql
```

6）查询 user 表

```
mysql> select user, host from user;
```

7）修改 user 表，把 Host 表内容修改为%

```
mysql> update user set host="%" where user="root";
```

8）刷新

```
mysql> flush privileges;
```

9）退出

```
mysql> quit;
```

2.3.3 卸载 MySQL

若因为安装失败或者其他原因，MySQL 需要卸载重装，可参考以下内容。

1）清空原有数据

（1）通过/etc/my.cnf 查看 MySQL 数据的存储位置

```
[atguigu@hadoop102 software]$ sudo cat /etc/my.cnf
[mysqld]
datadir=/var/lib/mysql
```

（2）去往/var/lib/mysql 路径需要 root 权限

```
[atguigu@hadoop102 mysql]$ su - root
[root@hadoop102 ~]# cd /var/lib/mysql
[root@hadoop102 mysql]# rm -rf *   （注意敲击命令的位置）
```

2）卸载 MySQL 相关包

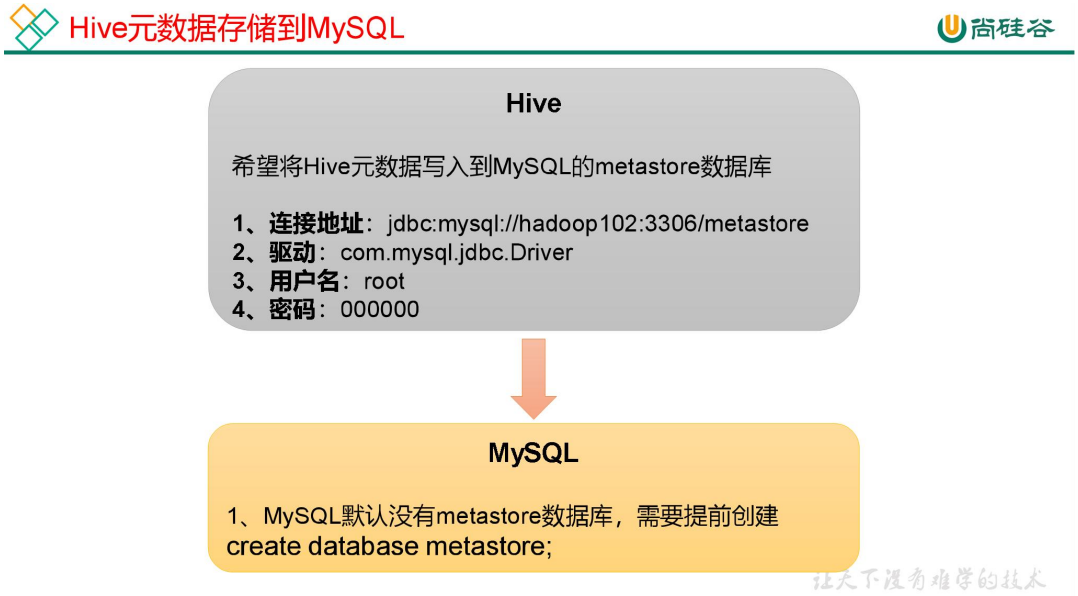
（1）查看安装过的 MySQL 相关包

```
[atguigu@hadoop102 software]$ sudo rpm -qa | grep -i -E mysql

mysql-community-libs-5.7.16-1.el7.x86_64
mysql-community-client-5.7.16-1.el7.x86_64
mysql-community-common-5.7.16-1.el7.x86_64
mysql-community-libs-compat-5.7.16-1.el7.x86_64
mysql-community-server-5.7.16-1.el7.x86_64
```

（2）一键卸载命令

```
[atguigu@hadoop102 software]$ rpm -qa | grep -i -E mysql\|mariadb
| xargs -n1 sudo rpm -e --nodeps
```

2.4 配置 Hive 元数据存储到 MySQL



2.4.1 配置元数据到 MySQL

1）新建 Hive 元数据库

```
#登录 MySQL
[atguigu@hadoop102 software]$ mysql -uroot -p123456

#创建 Hive 元数据库
mysql> create database metastore;
mysql> quit;
```

2）将 MySQL 的 JDBC 驱动拷贝到 Hive 的 lib 目录下。

```
[atguigu@hadoop102 software]$ cp /opt/software/mysql-connector-
java-5.1.37.jar $HIVE_HOME/lib
```

3）在$HIVE_HOME/conf 目录下新建 hive-site.xml 文件

```
[atguigu@hadoop102 software]$ vim $HIVE_HOME/conf/hive-site.xml
```

添加如下内容：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>
    <!-- jdbc 连接的 URL -->
    <property>
        <name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://hadoop102:3306/metastore?useSSL=false</value>
    </property>

    <!-- jdbc 连接的 Driver-->
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>

   <!-- jdbc 连接的 username-->
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>root</value>
    </property>

    <!-- jdbc 连接的 password -->
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>123456</value>
    </property>

    <!-- Hive 默认在 HDFS 的工作目录 -->
    <property>
        <name>hive.metastore.warehouse.dir</name>
        <value>/user/hive/warehouse</value>
    </property>
</configuration>
```

5）初始化 Hive 元数据库（修改为采用 MySQL 存储元数据）

```
[atguigu@hadoop102 hive]$ bin/schematool -dbType mysql -
initSchema -verbose
```

2.4.2 验证元数据是否配置成功

1）再次启动 Hive

```
[atguigu@hadoop102 hive]$ bin/hive
```

2）使用 Hive

```
hive> show databases;
hive> show tables;
hive> create table stu(id int, name string);
hive> insert into stu values(1,"ss");
hive> select * from stu;
```

3）在 Xshell 窗口中开启另一个窗口开启 Hive（两个窗口都可以操作 Hive，没有出现异常）

```
hive> show databases;
hive> show tables;
hive> select * from stu;
```

2.4.3 查看 MySQL 中的元数据

1）登录 MySQL

```
[atguigu@hadoop102 hive]$ mysql -uroot -p123456
```

2）查看元数据库 metastore

```
mysql> show databases;
mysql> use metastore;
mysql> show tables;
```

（1）查看元数据库中存储的库信息

```
mysql> select * from DBS;
+-------+----------------------+-------------------------------------------+---------+------------+------------+-----------+
| DB_ID | DESC                 | DB_LOCATION_URI                           | NAME    | OWNER_NAME | OWNER_TYPE | CTLG_NAME |
+-------+----------------------+-------------------------------------------+---------+------------+------------+-----------+
|     1 | Default Hive database | hdfs://hadoop102:8020/user/hive/warehouse | default | public     | ROLE       | hive      |
+-------+----------------------+-------------------------------------------+---------+------------+------------+-----------+
```

（2）查看元数据库中存储的表信息

```
mysql> select * from TBLS;
+--------+-------------+-------+------------------+---------+------------+-----------+-------+----------+---------------+
| TBL_ID | CREATE_TIME | DB_ID | LAST_ACCESS_TIME | OWNER   | OWNER_TYPE | RETENTION | SD_ID | TBL_NAME | TBL_TYPE      |
+--------+-------------+-------+------------------+---------+------------+-----------+-------+----------+---------------+
|      1 |  1656318303 |     1 |                0 | atguigu | USER       |         0 |     1 | stu      | MANAGED_TABLE |
+--------+-------------+-------+------------------+---------+------------+-----------+-------+----------+---------------+
```

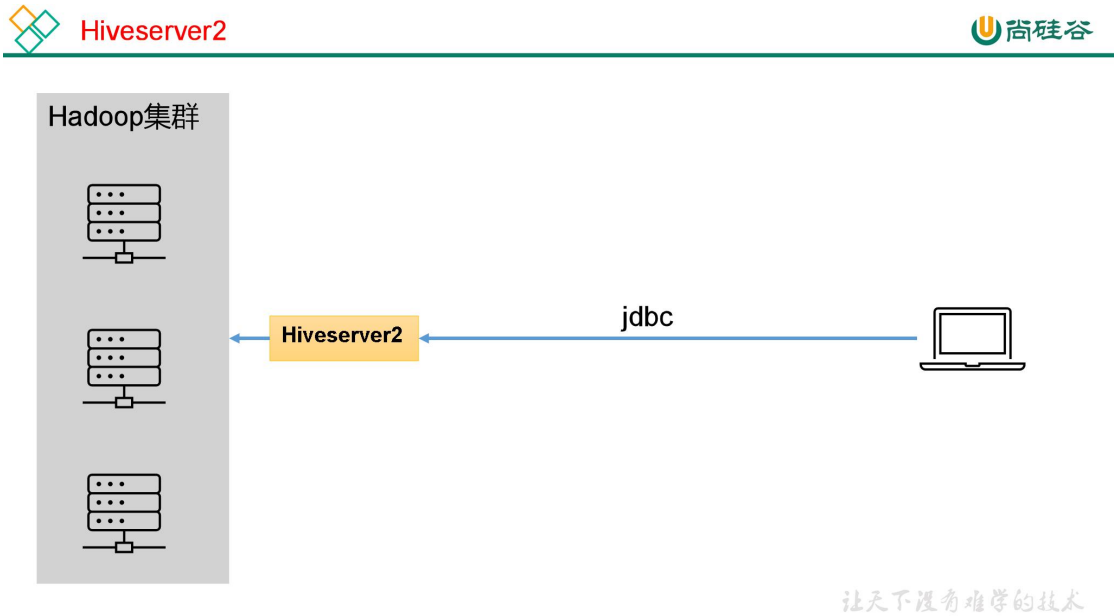
（3）查看元数据库中存储的表中列相关信息

```
mysql> select * from COLUMNS_V2;
+-------+----------+---------+------------+-------------+-------------+--------+
| CS_ID | CAT_NAME | DB_NAME | TABLE_NAME | COLUMN_NAME | COLUMN_TYPE | TBL_ID |
+-------+----------+---------+------------+-------------+-------------+--------+
|     1 | hive     | default | stu        | id          | int         |      1 |
|     2 | hive     | default | stu        | name        | string      |      1 |
+-------+----------+---------+------------+-------------+-------------+--------+
```

2.5 Hive 服务部署

2.5.1 hiveserver2 服务

Hive 的 hiveserver2 服务的作用是提供 jdbc/odbc 接口，为用户提供远程访问 Hive 数据的功能，例如用户期望在个人电脑中访问远程服务中的 Hive 数据，就需要用到 Hiveserver2。



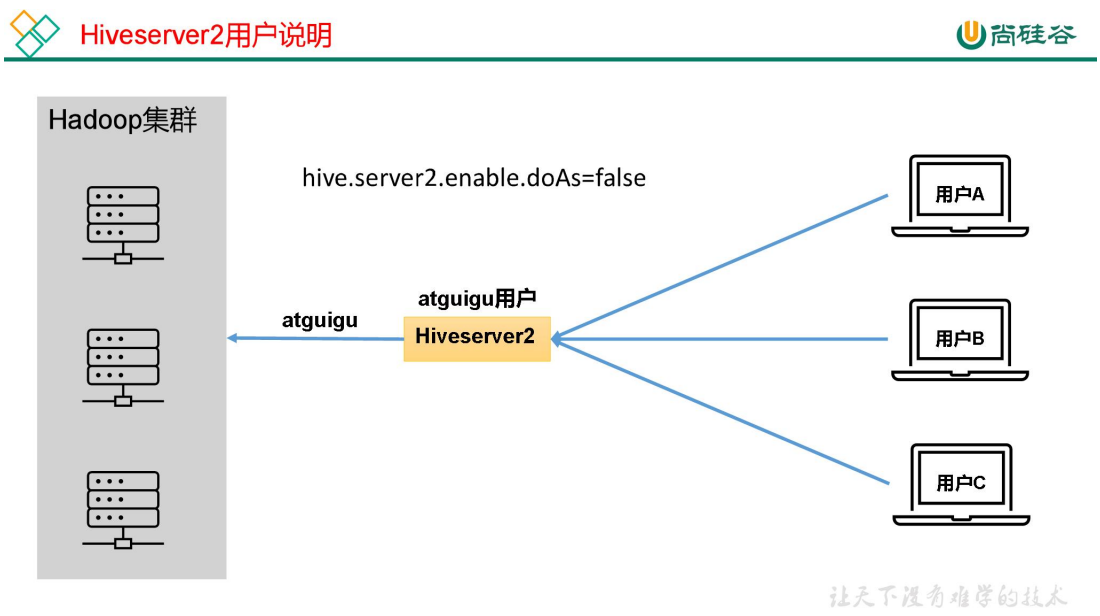
1）用户说明

在远程访问 Hive 数据时，客户端并未直接访问 Hadoop 集群，而是由 Hivesever2 代理访问。由于 Hadoop 集群中的数据具备访问权限控制，所以此时需考虑一个问题：那就是访问 Hadoop 集群的用户身份是谁？是 Hiveserver2 的启动用户？还是客户端的登录用户？

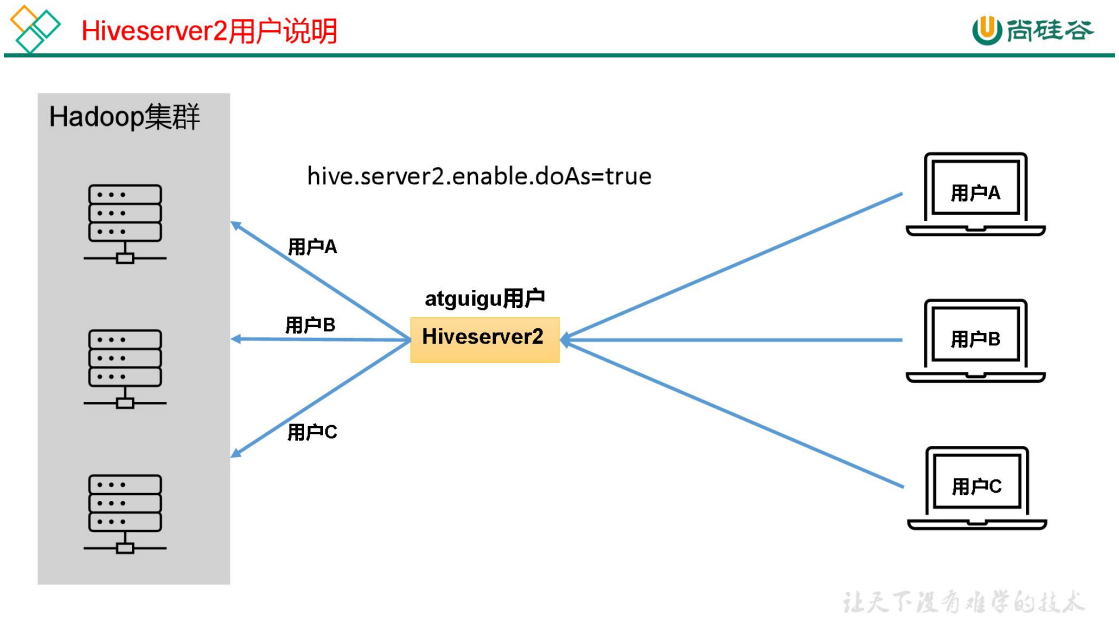
答案是都有可能，具体是谁，由 Hiveserver2 的 hive.server2.enable.doAs 参数决定，该参数的含义是是否启用 Hiveserver2 用户模拟的功能。若启用，则 Hiveserver2 会模拟成客户端的登录用户去访问 Hadoop 集群的数据，不启用，则 Hivesever2 会直接使用启动用户访问 Hadoop 集群数据。模拟用户的功能，默认是开启的。

具体逻辑如下：

（1）未开启用户模拟功能：



（2）开启用户模拟功能：



生产环境，推荐开启用户模拟功能，因为开启后才能保证各用户之间的权限隔离。

2）hiveserver2 部署

（1）Hadoop 端配置

hivesever2 的模拟用户功能，依赖于 Hadoop 提供的 proxy user（代理用户功能），只有 Hadoop 中的代理用户才能模拟其他用户的身份访问 Hadoop 集群。因此，需要将 hiveserver2 的启动用户设置为 Hadoop 的代理用户，配置方式如下：

修改配置文件 core-site.xml，然后记得分发三台机器

```
[atguigu@hadoop102 ~]$ cd $HADOOP_HOME/etc/hadoop
[atguigu@hadoop102 hadoop]$ vim core-site.xml
```

增加如下配置：

```
<!--配置所有节点的 atguigu 用户都可作为代理用户-->
<property>
    <name>hadoop.proxyuser.atguigu.hosts</name>
    <value>*</value>
</property>

<!--配置 atguigu 用户能够代理的用户组为任意组-->
<property>
    <name>hadoop.proxyuser.atguigu.groups</name>
    <value>*</value>
</property>

<!--配置 atguigu 用户能够代理的用户为任意用户-->
<property>
    <name>hadoop.proxyuser.atguigu.users</name>
    <value>*</value>
</property>
```

（2）Hive 端配置

在 hive-site.xml 文件中添加如下配置信息

```
[atguigu@hadoop102 conf]$ vim hive-site.xml

<!-- 指定 hiveserver2 连接的 host -->
<property>
   <name>hive.server2.thrift.bind.host</name>
   <value>hadoop102</value>
</property>

<!-- 指定 hiveserver2 连接的端口号 -->
<property>
   <name>hive.server2.thrift.port</name>
   <value>10000</value>
</property>
```

3）测试

（1）启动 hiveserver2

```
[atguigu@hadoop102 hive]$ bin/hive --service hiveserver2
```

（2）使用命令行客户端 beeline 进行远程访问

启动 beeline 客户端

```
[atguigu@hadoop102 hive]$ bin/beeline -u
jdbc:hive2://hadoop102:10000 -n atguigu
```

看到如下界面

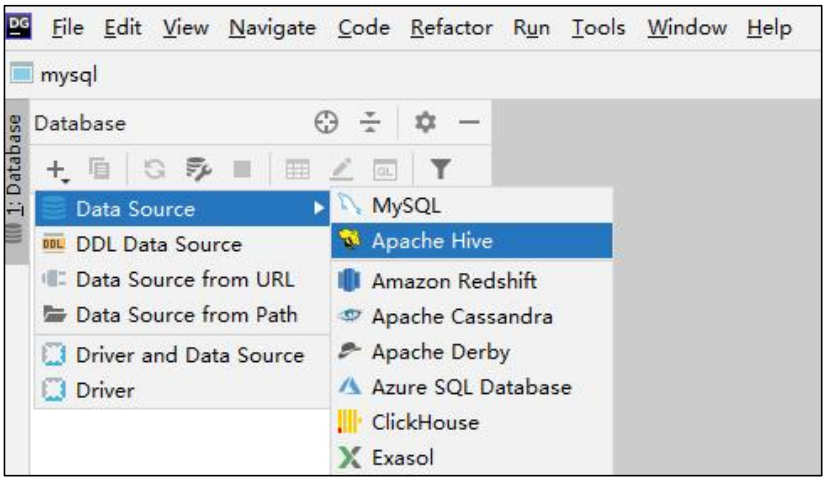
```
Connecting to jdbc:hive2://hadoop102:10000
Connected to: Apache Hive (version 3.1.3)
Driver: Hive JDBC (version 3.1.3)
Transaction isolation: TRANSACTION_REPEATABLE_READ
```

```
Beeline version 3.1.3 by Apache Hive
0: jdbc:hive2://hadoop102:10000>
```

（3）使用 Datagrip 图形化客户端进行远程访问

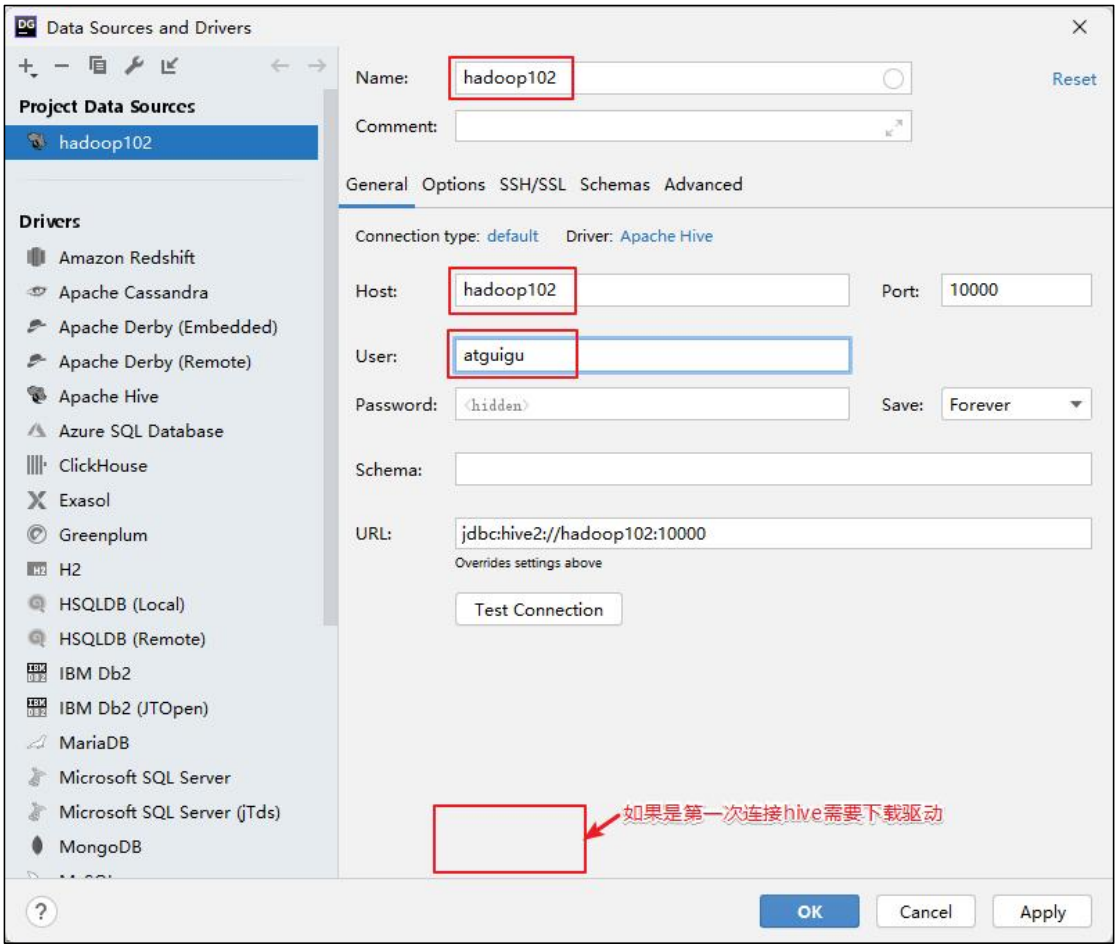
4）配置 DataGrip 连接

（1）创建连接



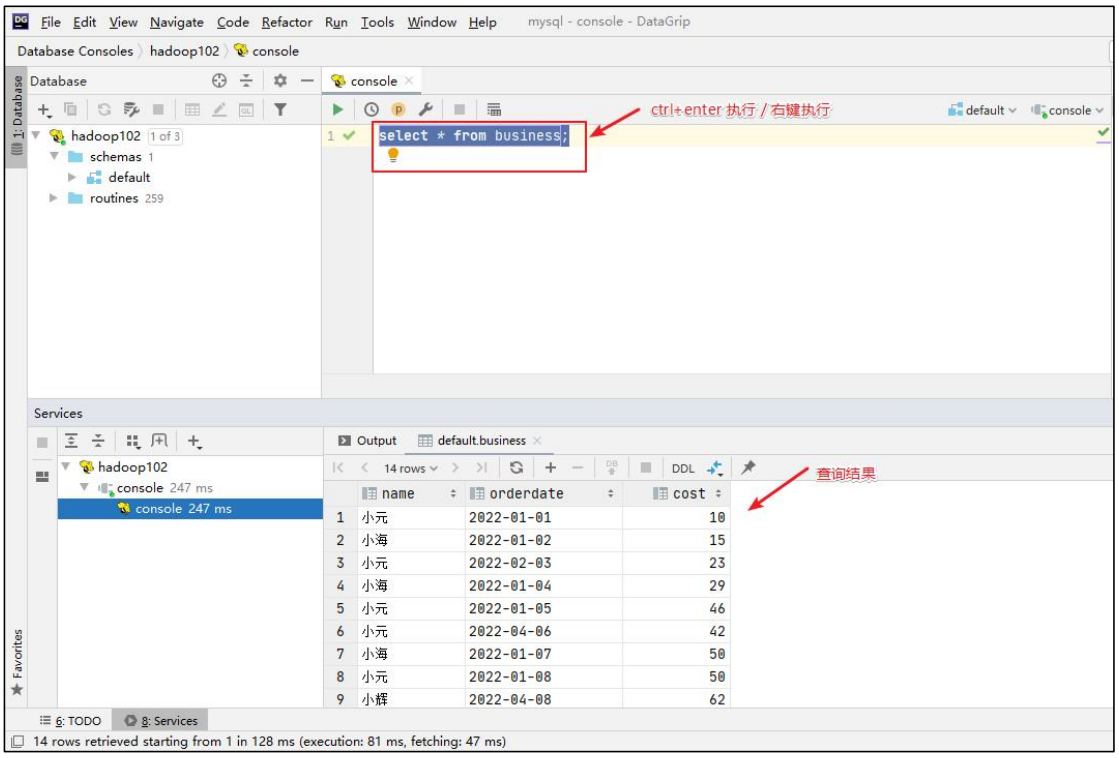
（2）配置连接属性

所有属性配置，和 Hive 的 beeline 客户端配置一致即可。初次使用，配置过程会提示缺少 JDBC 驱动，按照提示下载即可。

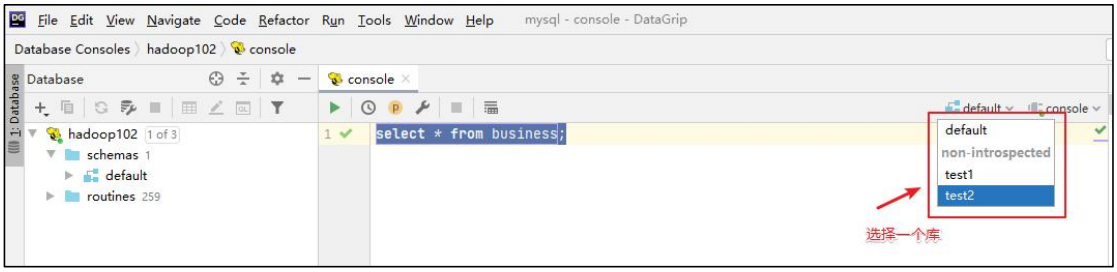


（3）界面介绍

（4）测试 sql 执行



（5）修改数据库



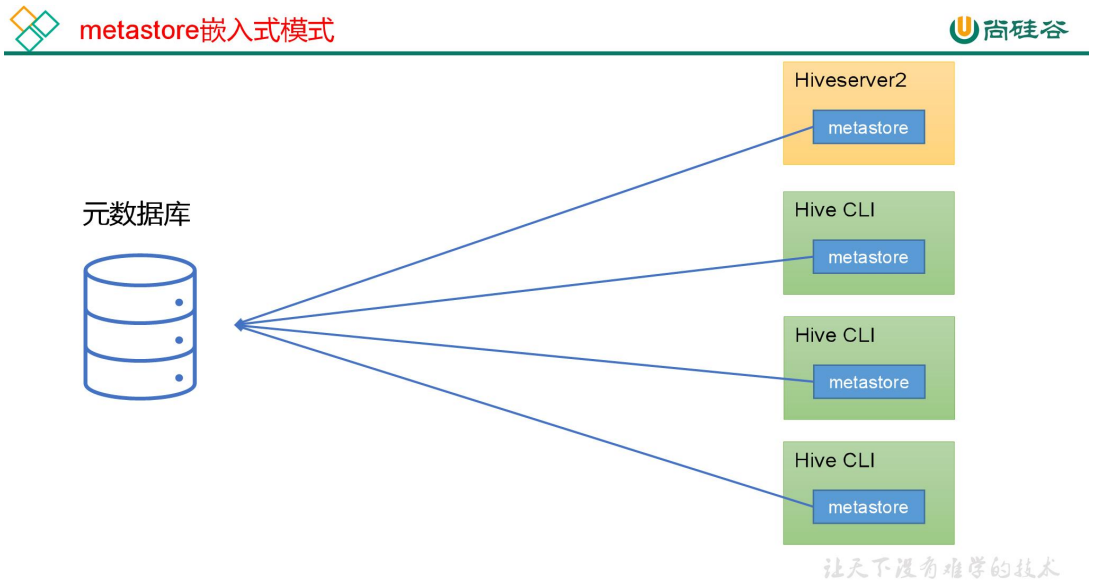
2.5.2 metastore 服务

Hive 的 metastore 服务的作用是为 Hive CLI 或者 Hiveserver2 提供元数据访问接口。

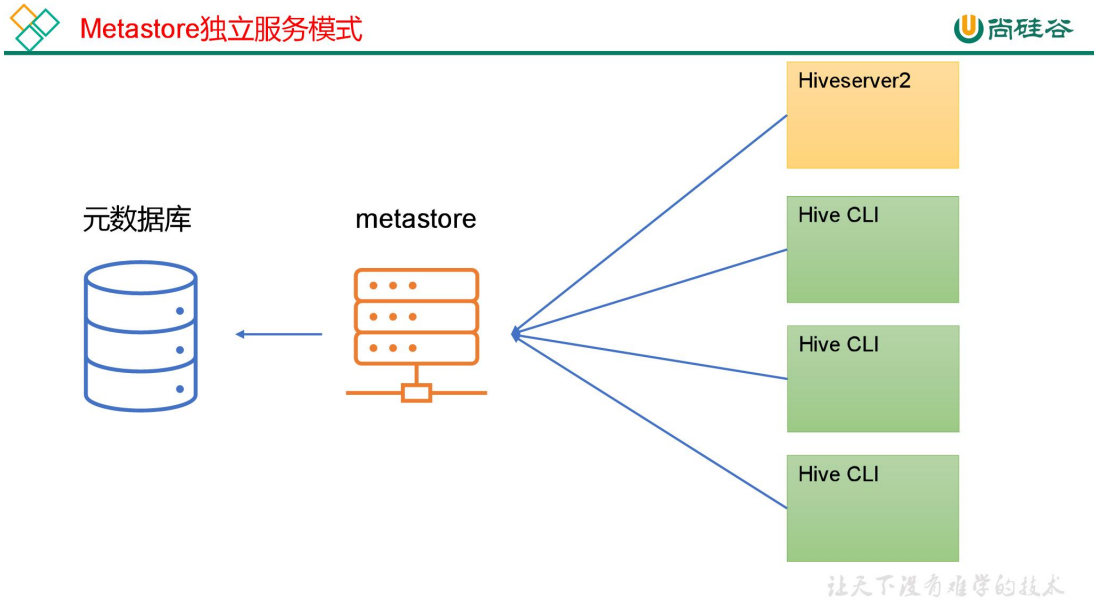
1）metastore 运行模式

metastore 有两种运行模式，分别为嵌入式模式和独立服务模式。下面分别对两种模式进行说明：

（1）嵌入式模式



（2）独立服务模式



生产环境中，不推荐使用嵌入式模式。因为其存在以下两个问题：

（1）嵌入式模式下，每个 Hive CLI 都需要直接连接元数据库，当 Hive CLI 较多时，数据库压力会比较大。

（2）每个客户端都需要用户元数据库的读写权限，元数据库的安全得不到很好的保证。

2）metastore 部署

（1）嵌入式模式

嵌入式模式下，只需保证 Hiveserver2 和每个 Hive CLI 的配置文件 hive-site.xml 中包含连接元数据库所需要的以下参数即可：

```
<!-- jdbc 连接的 URL -->
```

```
    <property>
        <name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://hadoop102:3306/metastore?useSSL=false</value>
    </property>

    <!-- jdbc 连接的 Driver-->
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>

   <!-- jdbc 连接的 username-->
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>root</value>
    </property>

    <!-- jdbc 连接的 password -->
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>123456</value>
    </property>
```

（2）独立服务模式

独立服务模式需做以下配置：

首先，保证 metastore 服务的配置文件 hive-site.xml 中包含连接元数据库所需的以下参数：

```
    <!-- jdbc 连接的 URL -->
    <property>
        <name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:mysql://hadoop102:3306/metastore?useSSL=false</value>
    </property>

    <!-- jdbc 连接的 Driver-->
    <property>
        <name>javax.jdo.option.ConnectionDriverName</name>
        <value>com.mysql.jdbc.Driver</value>
    </property>

   <!-- jdbc 连接的 username-->
    <property>
        <name>javax.jdo.option.ConnectionUserName</name>
        <value>root</value>
    </property>

    <!-- jdbc 连接的 password -->
    <property>
        <name>javax.jdo.option.ConnectionPassword</name>
        <value>123456</value>
```

```
    </property>
```

其次，保证 Hiveserver2 和每个 Hive CLI 的配置文件 hive-site.xml 中包含访问 metastore 服务所需的以下参数：

```
<!-- 指定 metastore 服务的地址 -->
<property>
   <name>hive.metastore.uris</name>
   <value>thrift://hadoop102:9083</value>
</property>
```

注意：主机名需要改为 metastore 服务所在节点，端口号无需修改，metastore 服务的默认端口就是 9083。

3）测试

此时启动 Hive CLI，执行 shou databases 语句，会出现一下错误提示信息：

```
hive (default)> show databases;
FAILED: HiveException java.lang.RuntimeException: Unable to
instantiate
org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClient
```

这是因为我们在 Hive CLI 的配置文件中配置了 hive.metastore.uris 参数，此时 Hive CLI 会去请求我们执行的 metastore 服务地址，所以必须启动 metastore 服务才能正常使用。

metastore 服务的启动命令如下：

```
[atguigu@hadoop202 hive]$ hive --service metastore
2022-04-24 16:58:08: Starting Hive Metastore Server
```

注意：启动后该窗口不能再操作，需打开一个新的 Xshell 窗口来对 Hive 操作。

重新启动 Hive CLI，并执行 shou databases 语句，就能正常访问了

```
[atguigu@hadoop202 hive]$ bin/hive
```

2.5.3 编写 Hive 服务启动脚本（了解）

1）前台启动的方式导致需要打开多个 Xshell 窗口，可以使用如下方式后台方式启动

- nohup：放在命令开头，表示关闭终端进程也继续保持运行状态
- /dev/null：是 Linux 文件系统中的一个文件，被称为黑洞，所有写入该文件的内容都会被自动丢弃
- 2>&1：表示将错误重定向到标准输出上
- &：放在命令结尾，表示后台运行

一般会组合使用：nohup [xxx 命令操作]> file 2>&1 &，表示将 xxx 命令运行的结果输出到 file 中，并保持命令启动的进程在后台运行。

如上命令不要求掌握。

```
[atguigu@hadoop202 hive]$ nohup hive --service metastore 2>&1 &
[atguigu@hadoop202 hive]$ nohup hive --service hiveserver2 2>&1 &
```

2）为了方便使用，可以直接编写脚本来管理服务的启动和关闭

```
[atguigu@hadoop102 hive]$ vim $HIVE_HOME/bin/hiveservices.sh
```

内容如下：此脚本的编写不要求掌握。直接拿来使用即可。

```
#!/bin/bash

HIVE_LOG_DIR=$HIVE_HOME/logs
if [ ! -d $HIVE_LOG_DIR ]
then
    mkdir -p $HIVE_LOG_DIR
fi

#检查进程是否运行正常，参数 1 为进程名，参数 2 为进程端口
function check_process()
{
    pid=$(ps -ef 2>/dev/null | grep -v grep | grep -i $1 | awk
'{print $2}')
    ppid=$(netstat -nltp 2>/dev/null | grep $2 | awk '{print $7}'
| cut -d '/' -f 1)
    echo $pid
    [[ "$pid" =~ "$ppid" ]] && [ "$ppid" ] && return 0 || return
1
}

function hive_start()
{
    metapid=$(check_process HiveMetastore 9083)
    cmd="nohup hive --service
metastore >$HIVE_LOG_DIR/metastore.log 2>&1 &"
    [ -z "$metapid" ] && eval $cmd || echo "Metastroe 服务已启动"
    server2pid=$(check_process HiveServer2 10000)
    cmd="nohup hive --service
hiveserver2 >$HIVE_LOG_DIR/hiveServer2.log 2>&1 &"
    [ -z "$server2pid" ] && eval $cmd || echo "HiveServer2 服务已启
动"
}

function hive_stop()
{
   metapid=$(check_process HiveMetastore 9083)
    [ "$metapid" ] && kill $metapid || echo "Metastore 服务未启动"
    server2pid=$(check_process HiveServer2 10000)
    [ "$server2pid" ] && kill $server2pid || echo "HiveServer2 服务
未启动"
}

case $1 in
"start")
    hive_start
    ;;
"stop")
    hive_stop
```

```
    ;;
"restart")
    hive_stop
    sleep 2
    hive_start
    ;;
"status")
    check_process HiveMetastore 9083 >/dev/null && echo
"Metastore 服务运行正常" || echo "Metastore 服务运行异常"
    check_process HiveServer2 10000 >/dev/null && echo
"HiveServer2 服务运行正常" || echo "HiveServer2 服务运行异常"
    ;;
*)
    echo Invalid Args!
    echo 'Usage: '$(basename $0)' start|stop|restart|status'
    ;;
esac
```

3）添加执行权限

```
[atguigu@hadoop102 hive]$ chmod +x $HIVE_HOME/bin/hiveservices.sh
```

4）启动 Hive 后台服务

```
[atguigu@hadoop102 hive]$ hiveservices.sh start
```

2.6 Hive 使用技巧

2.6.1 Hive 常用交互命令

```
[atguigu@hadoop102 hive]$ bin/hive -help
usage: hive
 -d,--define <key=value>          Variable subsitution to apply to hive
                                  commands. e.g. -d A=B or --define A=B
    --database <databasename>     Specify the database to use
 -e <quoted-query-string>         SQL from command line
 -f <filename>                      SQL from files
 -H,--help                        Print help information
    --hiveconf <property=value>   Use value for given property
    --hivevar <key=value>         Variable subsitution to apply to hive
                                  commands. e.g. --hivevar A=B
 -i <filename>                    Initialization SQL file
 -S,--silent                      Silent mode in interactive shell
 -v,--verbose                     Verbose mode (echo executed SQL to the
console)
```

1）在 Hive 命令行里创建一个表 student，并插入 1 条数据

```
hive (default)> create table student(id int,name string);
OK
Time taken: 1.291 seconds

hive (default)> insert into table student values(1,"zhangsan");
hive (default)> select * from student;
OK
student.id    student.name
1   zhangsan
Time taken: 0.144 seconds, Fetched: 1 row(s)
```

2）“-e”不进入 hive 的交互窗口执行 hql 语句

```
[atguigu@hadoop102 hive]$ bin/hive -e "select id from student;"
```

3）“-f”执行脚本中的 hql 语句

（1）在/opt/module/hive/下创建 datas 目录并在 datas 目录下创建 hivef.sql 文件

```
[atguigu@hadoop102 hive]$ mkdir datas
[atguigu@hadoop102 datas]$ vim hivef.sql
```

（2）文件中写入正确的 **hql** 语句

```
select * from student;
```

（3）执行文件中的 **hql** 语句

```
[atguigu@hadoop102 hive]$ bin/hive -f
/opt/module/hive/datas/hivef.sql
```

（4）执行文件中的 **hql** 语句并将结果写入文件中

```
[atguigu@hadoop102 hive]$ bin/hive -f
/opt/module/hive/datas/hivef.sql  >
/opt/module/hive/datas/hive_result.txt
```

2.6.2 Hive 参数配置方式

1）查看当前所有的配置信息

```
hive>set;
```

2）参数的配置三种方式

（1）配置文件方式

- 默认配置文件：hive-default.xml
- 用户自定义配置文件：hive-site.xml

注意：用户自定义配置会覆盖默认配置。另外，Hive 也会读入 Hadoop 的配置，因为 Hive 是作为 Hadoop 的客户端启动的，Hive 的配置会覆盖 Hadoop 的配置。配置文件的设定对本机启动的所有 Hive 进程都有效。

（2）命令行参数方式

①启动 Hive 时，可以在命令行添加-hiveconf param=value 来设定参数。例如：

```
[atguigu@hadoop103 hive]$ bin/hive -hiveconf
mapreduce.job.reduces=10;
```

注意：仅对本次 Hive 启动有效。

②查看参数设置

```
hive (default)> set mapreduce.job.reduces;
```

（3）参数声明方式

可以在 HQL 中使用 SET 关键字设定参数，例如：

```
hive(default)> set mapreduce.job.reduces=10;
```

———

注意：仅对本次 Hive 启动有效。

查看参数设置：

```
hive(default)> set mapreduce.job.reduces;
```

上述三种设定方式的优先级依次递增。即**配置文件 < 命令行参数 < 参数声明**。注意某些系统级的参数，例如 log4j 相关的设定，必须用前两种方式设定，因为那些参数的读取在会话建立以前已经完成了。

2.6.3 Hive 常见属性配置

1）Hive 客户端显示当前库和表头

（1）在 hive-site.xml 中加入如下两个配置:

```
[atguigu@hadoop102 conf]$ vim hive-site.xml

<property>
    <name>hive.cli.print.header</name>
    <value>true</value>
    <description>Whether to print the names of the columns in
query output.</description>
</property>
<property>
    <name>hive.cli.print.current.db</name>
    <value>true</value>
    <description>Whether to include the current database in the
Hive prompt.</description>
</property>
```

（2）hive 客户端在运行时可以显示当前使用的库和表头信息

```
[atguigu@hadoop102 conf]$ hive

hive (default)> select * from stu;
OK
stu.id stu.name
1   ss
Time taken: 1.874 seconds, Fetched: 1 row(s)
hive (default)>
```

2）Hive 运行日志路径配置

（1）Hive 的 log 默认存放在/tmp/atguigu/hive.log 目录下（当前用户名下）

```
[atguigu@hadoop102 atguigu]$ pwd
/tmp/atguigu
[atguigu@hadoop102 atguigu]$ ls
hive.log
hive.log.2022-06-27
```

（2）修改 Hive 的 log 存放日志到/opt/module/hive/logs

①修改$HIVE_HOME/conf/hive-log4j2.properties.template 文件名称为

hive-log4j2.properties

```
[atguigu@hadoop102 conf]$ pwd
/opt/module/hive/conf

[atguigu@hadoop102 conf]$ mv hive-log4j2.properties.template
hive-log4j2.properties
```

②在 hive-log4j2.properties 文件中修改 log 存放位置

```
[atguigu@hadoop102 conf]$ vim hive-log4j2.properties
```

修改配置如下

```
property.hive.log.dir=/opt/module/hive/logs
```

3）Hive 的 JVM 堆内存设置

新版本的 Hive 启动的时候，默认申请的 JVM 堆内存大小为 256M，JVM 堆内存申请的太小，导致后期开启本地模式，执行复杂的 SQL 时经常会报错：java.lang.OutOfMemoryError: Java heap space，因此最好提前调整一下 HADOOP_HEAPSIZE 这个参数。

（1）修改$HIVE_HOME/conf 下的 hive-env.sh.template 为 hive-env.sh

```
[atguigu@hadoop102 conf]$ pwd
/opt/module/hive/conf

[atguigu@hadoop102 conf]$ mv hive-env.sh.template hive-env.sh
```

（2）将 hive-env.sh 其中的参数 export HADOOP_HEAPSIZE 修改为 2048，重启 Hive。

修改前

```
# The heap size of the jvm stared by hive shell script can be
controlled via:
# export HADOOP_HEAPSIZE=1024
```

修改后

```
# The heap size of the jvm stared by hive shell script can be
controlled via:
export HADOOP_HEAPSIZE=2048
```

4）关闭 Hadoop 虚拟内存检查

在 yarn-site.xml 中关闭虚拟内存检查（虚拟内存校验，如果已经关闭了，就不需要配了）。

（1）修改前记得先停 Hadoop

```
[atguigu@hadoop102 hadoop]$ pwd
/opt/module/hadoop-3.1.3/etc/hadoop

[atguigu@hadoop102 hadoop]$ vim yarn-site.xml
```

（2）添加如下配置

```
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
```

（3）修改完后记得分发 yarn-site.xml，并重启 yarn。

第 3 章 DDL 数据定义语言

3.1 数据库（database）

3.1.1 创建数据库

1）语法

```
CREATE DATABASE [IF NOT EXISTS] database_name
[COMMENT database_comment]
[LOCATION hdfs_path]
[WITH DBPROPERTIES (property_name=property_value, ...)];
```

2）案例

（1）创建一个数据库，不指定路径

```
hive (default)> create database db_hive1;
```

注：若不指定路径，其默认路径为${hive.metastore.warehouse.dir}/database_name.db

（2）创建一个数据库，指定路径

```
hive (default)> create database db_hive2 location '/db_hive2';
```

（2）创建一个数据库，带有 dbproperties

```
hive (default)> create database db_hive3 with
dbproperties('create_date'='2022-11-18');
```

3.1.2 查询数据库

1）展示所有数据库

（1）语法

```
SHOW DATABASES [LIKE 'identifier_with_wildcards'];
```

注：like 通配表达式说明：*表示任意个任意字符，|表示或的关系。

（2）案例

```
hive> show databases like 'db_hive*';
OK
db_hive_1
db_hive_2
```

2）查看数据库信息

（1）语法

```
DESCRIBE DATABASE [EXTENDED] db_name;
```

（2）案例

- 查看基本信息

```
hive> desc database db_hive3;
OK
db_hive       hdfs://hadoop102:8020/user/hive/warehouse/db_hive.db
    atguigu    USER
```

➢ 查看更多信息

```
hive> desc database extended db_hive3;
OK
db_name    comment    location   owner_name    owner_type
    parameters
db_hive3
    hdfs://hadoop102:8020/user/hive/warehouse/db_hive3.db atguigu
    USER    {create_date=2022-11-18}
```

3.1.3 修改数据库

用户可以使用 alter database 命令修改数据库某些信息，其中能够修改的信息包括 dbproperties、location、owner user。需要注意的是：修改数据库 location，不会改变当前已有表的路径信息，而只是改变后续创建的新表的默认的父目录。

1）语法

```
--修改 dbproperties
ALTER DATABASE database_name SET DBPROPERTIES
(property_name=property_value, ...);

--修改 location
ALTER DATABASE database_name SET LOCATION hdfs_path;

--修改 owner user
ALTER DATABASE database_name SET OWNER USER user_name;
```

2）案例

（1）修改 dbproperties

```
hive> ALTER DATABASE db_hive3 SET DBPROPERTIES
('create_date'='2022-11-20');
```

3.1.4 删除数据库

1）语法

```
DROP DATABASE [IF EXISTS] database_name [RESTRICT|CASCADE];
```

注：RESTRICT：严格模式，若数据库不为空，则会删除失败，默认为该模式。

CASCADE：级联模式，若数据库不为空，则会将库中的表一并删除。

2）案例

（1）删除空数据库

```
hive> drop database db_hive2;
```

（2）删除非空数据库

```
hive> drop database db_hive3 cascade;
```

3.1.5 切换当前数据库

1）语法

```
USE database_name;
```

3.2 表（table）

3.2.1 创建表

3.2.1.1 创建普通表

1）完整语法

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS]
[db_name.]table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
[CLUSTERED BY (col_name, col_name, ...)
[SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
```

2）关键字说明：

（1）TEMPORARY

临时表，该表只在当前会话可见，会话结束，表会被删除。

（2）EXTERNAL（重点）

外部表，与之相对应的是内部表（管理表）。管理表意味着 Hive 会完全接管该表，包括元数据和 HDFS 中的数据。而外部表则意味着 Hive 只接管元数据，而不完全接管 HDFS 中的数据。

（3）data_type（重点）

Hive 中的字段类型可分为基本数据类型和复杂数据类型。

基本数据类型如下：

Hive	说明	定义
tinyint	1byte 有符号整数	
smallint	2byte 有符号整数	
int	4byte 有符号整数	
bigint	8byte 有符号整数	
boolean	布尔类型，true 或者 false	
float	单精度浮点数	

double	双精度浮点数	
decimal	十进制精准数字类型	decimal(16,2)
varchar	字符序列，需指定最大长度，最大长度的范围是[1,65535]	varchar(32)
string	字符串，无需指定最大长度	
timestamp	时间类型	
binary	二进制数据	

复杂数据类型如下；

类型	说明	定义	取值
array	数组是一组相同类型的值的集合	array<string>	arr[0]
map	map 是一组相同类型的键-值对集合	map<string, int>	map['key']
struct	结构体由多个属性组成，每个属性都有自己的属性名和数据类型	struct<id:int, name:string>	struct.id

注：类型转换

Hive 的基本数据类型可以做类型转换，转换的方式包括隐式转换以及显示转换。

方式一：隐式转换

具体规则如下：

①任何整数类型都可以隐式地转换为一个范围更广的类型，如 tinyint 可以转换成 int，int 可以转换成 bigint。

②所有整数类型、float 和 string 类型都可以隐式地转换成 double。

③tinyint、smallint、int 都可以转换为 float。

④boolean 类型不可以转换为任何其它的类型。

详情可参考 Hive 官方说明：Allowed Implicit Conversions

方式二：显示转换

可以借助 cast 函数完成显示的类型转换

①语法

```
cast(expr as <type>)
```

②案例

```
hive (default)> select '1' + 2, cast('1' as int) + 2;

_c0    _c1
3.0     3
```

（4）PARTITIONED BY（重点）

创建分区表

（5）CLUSTERED BY ... SORTED BY...INTO ... BUCKETS（重点）

创建分桶表

（6）ROW FORMAT（重点）

指定 SERDE，SERDE 是 Serializer and Deserializer 的简写。Hive 使用 SERDE 序列化和反序列化每行数据。详情可参考 Hive-Serde。语法说明如下：

语法一：DELIMITED 关键字表示对文件中的每个字段按照特定分割符进行分割，其会使用默认的 SERDE 对每行数据进行序列化和反序列化。

```
ROW FORAMT DELIMITED
[FIELDS TERMINATED BY char]
[COLLECTION ITEMS TERMINATED BY char]
[MAP KEYS TERMINATED BY char]
[LINES TERMINATED BY char]
[NULL DEFINED AS char]
```

注：

- **fields terminated by：**列分隔符
- **collection items terminated by：** map、struct 和 array 中每个元素之间的分隔符
- **map keys terminated by：**map 中的 key 与 value 的分隔符
- **lines terminated by：**行分隔符

语法二：SERDE 关键字可用于指定其他内置的 SERDE 或者用户自定义的 SERDE。例如 JSON SERDE，可用于处理 JSON 字符串。

```
ROW FORMAT SERDE serde_name [WITH SERDEPROPERTIES
(property_name=property_value,property_name=property_value, ...)]
```

（7）STORED AS（重点）

指定文件格式，常用的文件格式有，textfile（默认值），sequence file，orc file、parquet file 等等。

（8）LOCATION

指定表所对应的 HDFS 路径，若不指定路径，其默认值为

${hive.metastore.warehouse.dir}/db_name.db/table_name

（9）TBLPROPERTIES

用于配置表的一些 KV 键值对参数

3.2.1.2 案例演示

1）内部表与外部表

（1）内部表

Hive 中默认创建的表都是的**内部表**，有时也被称为**管理表**。对于内部表，Hive 会完全管理表的元数据和数据文件。

创建内部表如下：

```
create table if not exists student(
    id int,
    name string
)
row format delimited fields terminated by '\t'
location '/user/hive/warehouse/student';
```

准备其需要的文件如下，注意字段之间的分隔符。

```
[atguigu@hadoop102 datas]$ vim /opt/module/datas/student.txt

1001   student1
1002   student2
1003   student3
1004   student4
1005   student5
1006   student6
1007   student7
1008   student8
1009   student9
1010   student10
1011   student11
1012   student12
1013   student13
1014   student14
1015   student15
1016   student16
```

上传文件到 Hive 表指定的路径

```
[atguigu@hadoop102 datas]$ hadoop fs -put student.txt
/user/hive/warehouse/student
```

删除表，观察数据 HDFS 中的数据文件是否还在

```
hive (default)> drop table student;
```

（2）外部表

外部表通常可用于处理其他工具上传的数据文件，对于外部表，Hive 只负责管理元数据，不负责管理 HDFS 中的数据文件。

创建外部表如下：

```
create external table if not exists student(
    id int,
    name string
)
row format delimited fields terminated by '\t'
location '/user/hive/warehouse/student';
```

上传文件到 Hive 表指定的路径

```
[atguigu@hadoop102 datas]$ hadoop fs -put student.txt
/user/hive/warehouse/student
```

删除表，观察数据 HDFS 中的数据文件是否还在

```
hive (default)> drop table student;
```

2）SERDE 和复杂数据类型

本案例重点练习 SERDE 和复杂数据类型的使用。

若现有如下格式的 JSON 文件需要由 Hive 进行分析处理，请考虑如何设计表？

注：以下内容为格式化之后的结果，文件中每行数据为一个完整的 JSON 字符串。

```
{
    "name": "dasongsong",
    "friends": [
        "bingbing",
        "lili"
    ],
    "students": {
        "xiaohaihai": 18,
        "xiaoyangyang": 16
    },
    "address": {
        "street": "hui long guan",
        "city": "beijing",
        "postal_code": 10010
    }
}
```

我们可以考虑使用专门负责 JSON 文件的 JSON Serde，设计表字段时，表的字段与 JSON 字符串中的一级字段保持一致，对于具有嵌套结构的 JSON 字符串，考虑使用合适复杂数据类型保存其内容。最终设计出的表结构如下：

```
hive>
create table teacher
(
    name      string,
    friends   array<string>,
    students  map<string,int>,
    address   struct<city:string,street:string,postal_code:int>
)
row format serde 'org.apache.hadoop.hive.serde2.JsonSerDe'
location '/user/hive/warehouse/teacher';
```

创建该表，并准备以下文件。注意，需要确保文件中每行数据都是一个完整的 JSON 字符串，JSON SERDE 才能正确的处理。

```
[atguigu@hadoop102 datas]$ vim /opt/module/datas/teacher.txt

{"name":"dasongsong","friends":["bingbing","lili"],"students":{"x
iaohaihai":18,"xiaoyangyang":16},"address":{"street":"hui long
guan","city":"beijing","postal_code":10010}}
```

上传文件到 Hive 表指定的路径

```
[atguigu@hadoop102 datas]$ hadoop fs -put teacher.txt
/user/hive/warehouse/teacher
```

尝试从复杂数据类型的字段中取值

3.2.1.3 特殊建表

1）语法：Create Table As Select（CTAS）建表

该语法允许用户利用 select 查询语句返回的结果，直接建表，表的结构和查询语句的结构保持一致，且保证包含 select 查询语句放回的内容。

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name
[COMMENT table_comment]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
[AS select_statement]
```

2）语法：Create Table Like 表

该语法允许用户复刻一张已经存在的表结构，与上述的 CTAS 语法不同，该语法创建出来的表中不包含数据。

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS]
[db_name.]table_name
[LIKE exist_table_name]
[ROW FORMAT row_format]
[STORED AS file_format]
[LOCATION hdfs_path]
[TBLPROPERTIES (property_name=property_value, ...)]
```

3.2.1.4 案例演示

1）create table as select

```
hive>
create table teacher1 as select * from teacher;
```

2）create table like

```
hive>
create table teacher2 like teacher;
```

3.2.2 查看表

1）展示所有表

（1）语法

```
SHOW TABLES [IN database_name] LIKE ['identifier_with_wildcards'];
```

注：like 通配表达式说明：*表示任意个任意字符，|表示或的关系。

（2）案例

```
hive> show tables like 'stu*';
```

2）查看表信息

（1）语法

```
DESCRIBE [EXTENDED | FORMATTED] [db_name.]table_name
```

注：EXTENDED：展示详细信息

FORMATTED：对详细信息进行格式化的展示

（2）案例

①查看基本信息

```
hive> desc stu;
```

②查看更多信息

```
hive> desc formatted stu;
```

3.2.3 修改表

1）重命名表

（1）语法

```
ALTER TABLE table_name RENAME TO new_table_name
```

（2）案例

```
hive (default)> alter table stu rename to stu1;
```

2）修改列信息

（1）语法

- 增加列

该语句允许用户增加新的列，新增列的位置位于末尾。

```
ALTER TABLE table_name ADD COLUMNS (col_name data_type [COMMENT
col_comment], ...)
```

- 更新列

该语句允许用户修改指定列的列名、数据类型、注释信息以及在表中的位置。

```
ALTER TABLE table_name CHANGE [COLUMN] col_old_name col_new_name
column_type [COMMENT col_comment] [FIRST|AFTER column_name]
```

- 替换列

该语句允许用户用新的列集替换表中原有的全部列。

```
ALTER TABLE table_name REPLACE COLUMNS (col_name data_type
[COMMENT col_comment], ...)
```

2）案例

（1）查询表结构

```
hive (default)> desc stu;
```

（2）添加列

```
hive (default)> alter table stu add columns(age int);
```

（3）查询表结构

```
hive (default)> desc stu;
```

（4）更新列

```
hive (default)> alter table stu change column age ages double;
```

（6）替换列

```
hive (default)> alter table stu replace columns(id int, name
string);
```

3.2.4 删除表

1）语法

```
DROP TABLE [IF EXISTS] table_name;
```

2）案例

```
hive (default)> drop table stu;
```

3.2.5 清空表

1）语法

```
TRUNCATE [TABLE] table_name
```

注意：truncate 只能清空管理表，不能删除外部表中数据。

2）案例

```
hive (default)> truncate table student;
```

第 4 章 DML 数据操作语言

4.1 Load

Load 语句可将文件导入到 Hive 表中。

1）语法

```
hive>
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE
tablename [PARTITION (partcol1=val1, partcol2=val2 ...)];
```

关键字说明：

（1）local：表示从本地加载数据到 Hive 表；否则从 HDFS 加载数据到 Hive 表。

（2）overwrite：表示覆盖表中已有数据，否则表示追加。

（3）partition：表示上传到指定分区，若目标是分区表，需指定分区。

2）实操案例

（0）创建一张表

```
hive (default)>
create table student(
    id int,
```

```
    name string
)
row format delimited fields terminated by '\t';
```

（1）加载本地文件到 hive

```
hive (default)> load data local inpath
'/opt/module/datas/student.txt' into table student;
```

（2）加载 HDFS 文件到 hive 中

①上传文件到 HDFS

```
[atguigu@hadoop102 ~]$ hadoop fs -put
/opt/module/datas/student.txt /user/atguigu
```

②加载 HDFS 上数据，导入完成后去 HDFS 上查看文件是否还存在

```
hive (default)>
load data inpath '/user/atguigu/student.txt'
into table student;
```

（3）加载数据覆盖表中已有的数据

①上传文件到 HDFS

```
hive (default)> dfs -put /opt/module/datas/student.txt
/user/atguigu;
```

②加载数据覆盖表中已有的数据

```
hive (default)>
load data inpath '/user/atguigu/student.txt'
overwrite into table student;
```

4.2 Insert

4.2.1 将查询结果插入表中

1）语法

```
INSERT (INTO | OVERWRITE) TABLE tablename [PARTITION
(partcol1=val1, partcol2=val2 ...)] select_statement;
```

关键字说明：

（1）INTO：将结果追加到目标表

（2）OVERWRITE：用结果覆盖原有数据

2）案例

（1）新建一张表

```
hive (default)>
create table student1(
    id int,
    name string
)
row format delimited fields terminated by '\t';
```

（2）根据查询结果插入数据

```
hive (default)> insert overwrite table student3
```

```
select
    id,
    name
from student;
```

4.2.2 将给定 Values 插入表中

1）语法

```
INSERT (INTO | OVERWRITE) TABLE tablename [PARTITION
(partcol1[=val1], partcol2[=val2] ...)] VALUES values_row [,
values_row ...]
```

2）案例

```
hive (default)> insert into table  student1
values(1,'wangwu'),(2,'zhaoliu');
```

4.2.3 将查询结果写入目标路径

1）语法

```
INSERT OVERWRITE [LOCAL] DIRECTORY directory
  [ROW FORMAT row_format] [STORED AS
file_format] select_statement;
```

2）案例

```
insert overwrite local directory '/opt/module/datas/student' ROW
FORMAT SERDE 'org.apache.hadoop.hive.serde2.JsonSerDe'
select id,name from student;
```

4.3 Export&Import

Export 导出语句可将表的数据和元数据信息一并到处的 HDFS 路径，Import 可将 Export 导出的内容导入 Hive，表的数据和元数据信息都会恢复。Export 和 Import 可用于两个 Hive 实例之间的数据迁移。

1）语法

```
--导出
EXPORT TABLE tablename TO 'export_target_path'

--导入
IMPORT [EXTERNAL] TABLE new_or_original_tablename FROM
'source_path' [LOCATION 'import_target_path']
```

2）案例

```
--导出
hive>
export table default.student to
'/user/hive/warehouse/export/student';

--导入
hive>
import table student2 from '/user/hive/warehouse/export/student';
```

第 5 章 查询

5.1 基础语法

1）官网地址

https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Select

2）查询语句语法：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
  FROM table_reference        -- 从什么表查
  [WHERE where_condition]    -- 过滤
  [GROUP BY col_list]         -- 分组查询
   [HAVING col_list]           -- 分组后过滤
  [ORDER BY col_list]         -- 排序
  [CLUSTER BY col_list
    | [DISTRIBUTE BY col_list] [SORT BY col_list]
  ]
 [LIMIT number]                 -- 限制输出的行数
```

5.2 基本查询（Select…From）

5.2.1 数据准备

1）原始数据

（1）在/opt/module/hive/datas/路径上创建 dept.txt 文件，并赋值如下内容：

部门编号 部门名称 部门位置 id

```
[atguigu@hadoop102 datas]$ vim dept.txt

10  行政部  1700
20  财务部  1800
30  教学部  1900
40  销售部  1700
```

（2）在/opt/module/hive/datas/路径上创建 emp.txt 文件，并赋值如下内容：

员工编号 姓名 岗位 薪资 部门

```
[atguigu@hadoop102 datas]$ vim emp.txt

7369   张三    研发    800.00 30
7499   李四    财务    1600.00   20
7521   王五    行政    1250.00   10
7566   赵六    销售    2975.00   40
7654   侯七    研发    1250.00   30
7698   马八    研发    2850.00   30
7782   金九    \N  2450.0 30
7788   银十    行政    3000.00   10
```

```
7839    小芳    销售    5000.00    40
7844    小明    销售    1500.00    40
7876    小李    行政    1100.00    10
7900    小元    讲师    950.00 30
7902    小海    行政    3000.00    10
7934    小红明  讲师    1300.00    30
```

1）创建部门表

```
hive (default)>
create table if not exists dept(
    deptno int,     -- 部门编号
    dname string,   -- 部门名称
    loc int         -- 部门位置
)
row format delimited fields terminated by '\t';
```

2）创建员工表

```
hive (default)>
create table if not exists emp(
    empno int,       -- 员工编号
    ename string,    -- 员工姓名
    job string,      -- 员工岗位（大数据工程师、前端工程师、java 工程师）
    sal double,      -- 员工薪资
    deptno int       -- 部门编号
)
row format delimited fields terminated by '\t';
```

3）导入数据

```
hive (default)>
load data local inpath '/opt/module/hive/datas/dept.txt' into
table dept;
load data local inpath '/opt/module/hive/datas/emp.txt' into
table emp;
```

5.2.2 全表和特定列查询

1）全表查询

```
hive (default)> select * from emp;
```

2）选择特定列查询

```
hive (default)> select empno, ename from emp;
```

注意：

（1）SQL 语言大小写不敏感。

（2）SQL 可以写在一行或者多行。

（3）关键字不能被缩写也不能分行。

（4）各子句一般要分行写。

（5）使用缩进提高语句的可读性。

5.2.3 列别名

1）重命名一个列

2）便于计算

3）紧跟列名，也可以在列名和别名之间加入关键字‘AS’

4）案例实操

查询名称和部门。

```
hive (default)>
select
    ename AS name,
    deptno dn
from emp;
```

5.2.4 Limit 语句

典型的查询会返回多行数据。limit 子句用于限制返回的行数。

```
hive (default)> select * from emp limit 5;
hive (default)> select * from emp limit 2,3; -- 表示从第 2 行开始，向下抓取 3 行
```

5.2.5 Order By 全局排序

Order By：全局排序，只有一个 Reduce。

1）使用 Order By 子句排序

asc（ascend）：升序（默认）。

desc（descend）：降序。

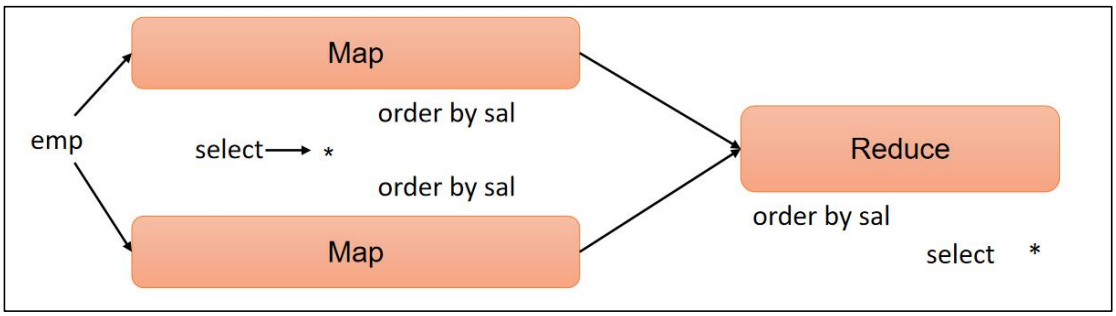
2）Order By 子句在 select 语句的结尾

3）基础案例实操

（1）查询员工信息按工资升序排列

```
hive (default)>
select
    *
from emp
order by sal;
```

hive sql 执行过程：



（2）查询员工信息按工资降序排列

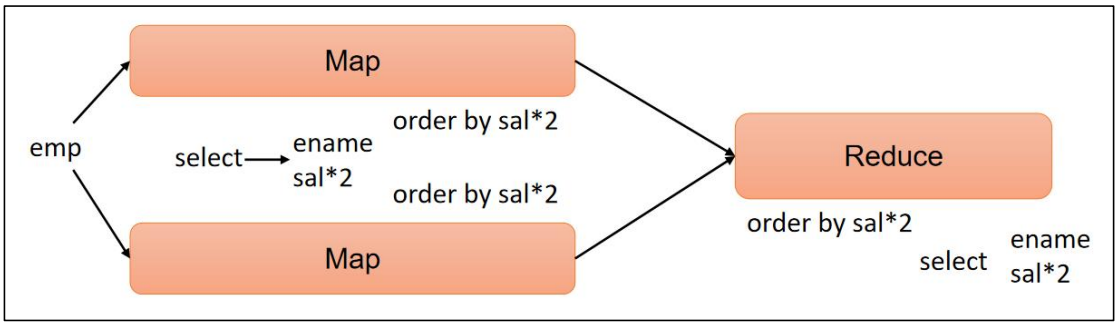
```
hive (default)>
select
    *
from emp
order by sal desc;
```

4）按照别名排序案例实操

按照员工薪水的 2 倍排序。

```
hive (default)>
select
    ename,
    sal * 2 twosal
from emp
order by twosal;
```

hive sql 执行过程：

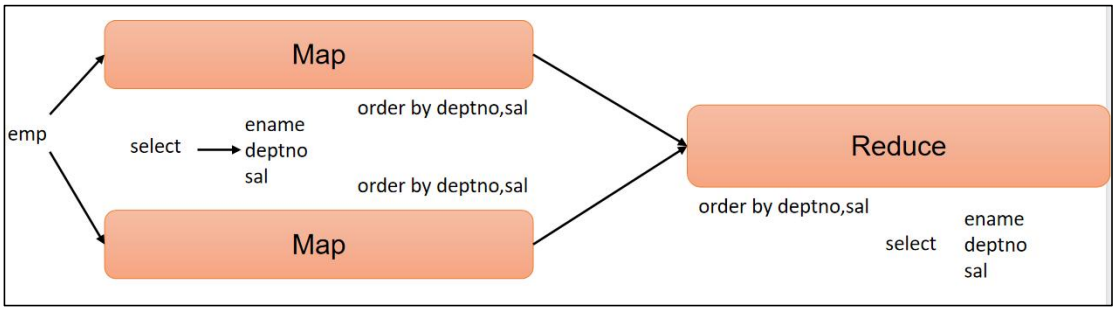


5）多个列排序案例实操

按照部门和工资升序排序。

```
hive (default)>
select
    ename,
    deptno,
    sal
from emp
order by deptno, sal;
```

hive sql 执行过程：



5.2.6 Where 语句

1）使用 where 子句，将不满足条件的行过滤掉

2）where 子句紧随 from 子句

3）案例实操

查询出薪水大于 1000 的所有员工。

```
hive (default)> select * from emp where sal > 1000;
```

注意：where 子句中不能使用字段别名。

5.2.7 关系运算函数

1）基本语法

如下操作符主要用于 where 和 having 语句中。

操作符	支持的数据类型	描述
A=B	基本数据类型	如果 A 等于 B 则返回 true，反之返回 false
A<=>B	基本数据类型	如果 A 和 B 都为 null 或者都不为 null，则返回 true，如果只有一边为 null，返回 false
A<>B, A!=B	基本数据类型	A 或者 B 为 null 则返回 null；如果 A 不等于 B，则返回 true，反之返回 false
A<B	基本数据类型	A 或者 B 为 null，则返回 null；如果 A 小于 B，则返回 true，反之返回 false
A<=B	基本数据类型	A 或者 B 为 null，则返回 null；如果 A 小于等于 B，则返回 true，反之返回 false
A>B	基本数据类型	A 或者 B 为 null，则返回 null；如果 A 大于 B，则返回 true，反之返回 false
A>=B	基本数据类型	A 或者 B 为 null，则返回 null；如果 A 大于等于 B，则返回 true，反之返回 false
A [not] between B and C	基本数据类型	如果 A，B 或者 C 任一为 null，则结果为 null。如果 A 的值大于等于 B 而且小于或等于 C，则结果为 true，反之为 false。如果使用 not 关键字则可达到相反的效果。
A is null	所有数据类型	如果 A 等于 null，则返回 true，反之返回 false
A is not null	所有数据类型	如果 A 不等于 null，则返回 true，反之返回 false
in（数值 1，数值 2）	所有数据类型	使用 in 运算显示列表中的值
A [not] like B	string 类型	B 是一个 SQL 下的简单正则表达式，也叫通配符模式，如果 A 与其匹配的话，则返回 true；反之返回 false。B 的表达式说明如下：‘x%’表示 A 必须以字母‘x’开头，

		‘%x’表示 A 必须以字母‘x’结尾，而‘%x%’表示 A 包含有字母‘x’,可以位于开头，结尾或者字符串中间。如果使用 not 关键字则可达到相反的效果。
A rlike B, A regexp B	string 类型	B 是基于 java 的正则表达式，如果 A 与其匹配，则返回 true；反之返回 false。匹配使用的是 JDK 中的正则表达式接口实现的，因为正则也依据其中的规则。例如，正则表达式必须和整个字符串 A 相匹配，而不是只需与其字符串匹配。

5.2.8 逻辑运算函数

1）基本语法（and/or/not）

操作符	含义
and	逻辑并
or	逻辑或
not	逻辑否

2）案例实操

（1）查询薪水大于 1000，部门是 30

```
hive (default)>
select
    *
from emp
where sal > 1000 and deptno = 30;
```

（2）查询薪水大于 1000，或者部门是 30

```
hive (default)>
select
    *
from emp
where sal>1000 or deptno=30;
```

（3）查询除了 20 部门和 30 部门以外的员工信息

```
hive (default)>
select
    *
from emp
where deptno not in(30, 20);
```

5.3 分组聚合

5.3.1 聚合函数

1）语法

count(*)，表示统计所有行数，包含 null 值；

count(某列)，表示该列一共有多少行，不包含 null 值；

max()，求最大值，不包含 null，除非所有值都是 null；

min()，求最小值，不包含 null，除非所有值都是 null；

sum()，求和，不包含 null。

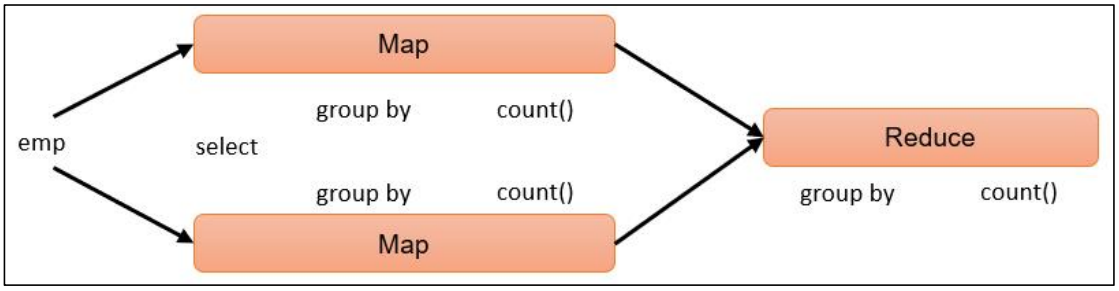
avg()，求平均值，不包含 null。

2）案例实操

（1）求总行数（count）

```
hive (default)> select count(*) cnt from emp;
```

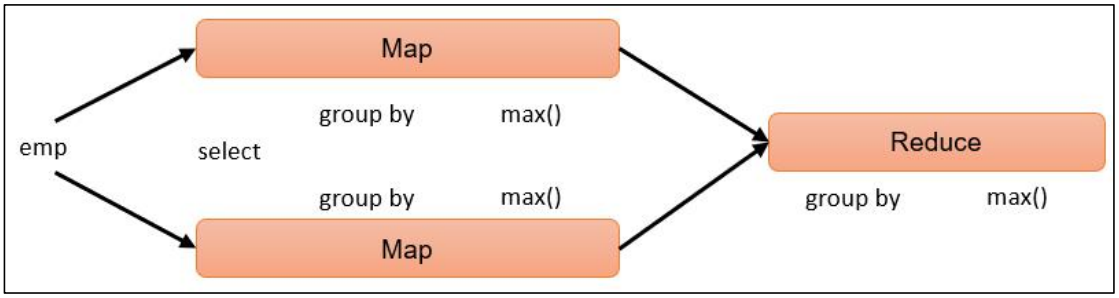
hive sql 执行过程：



（2）求工资的最大值（max）

```
hive (default)> select max(sal) max_sal from emp;
```

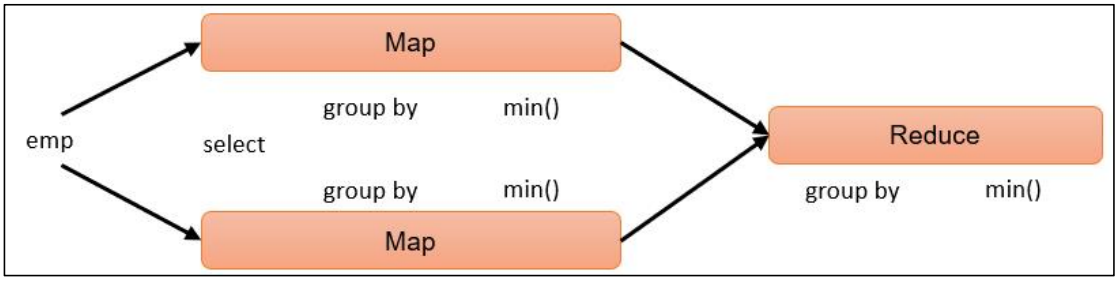
hive sql 执行过程：



（3）求工资的最小值（min）

```
hive (default)> select min(sal) min_sal from emp;
```

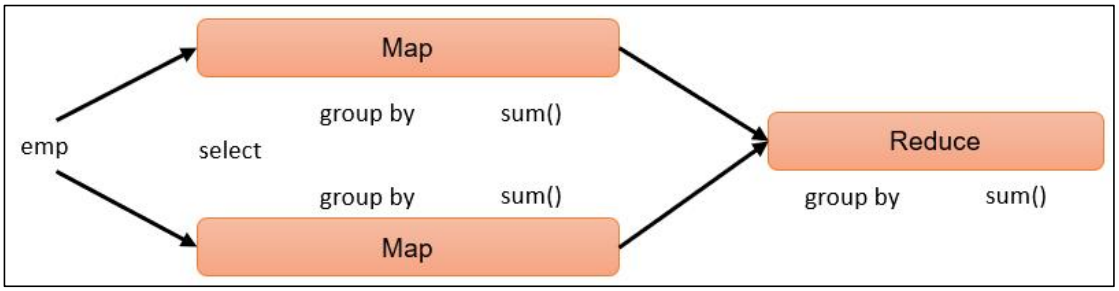
hive sql 执行过程：



（4）求工资的总和（sum）

```
hive (default)> select sum(sal) sum_sal from emp;
```

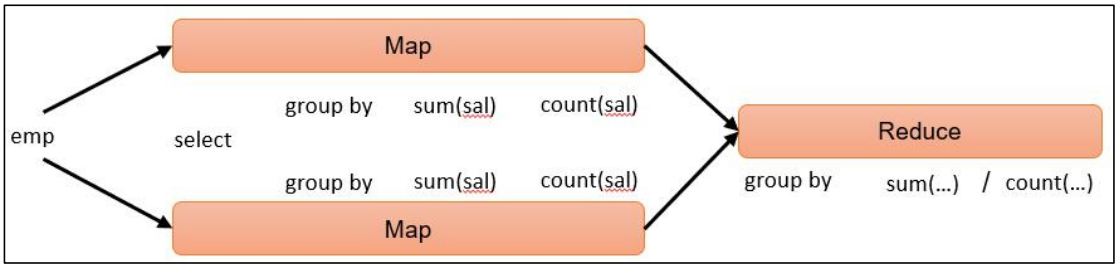
hive sql 执行过程：



（5）求工资的平均值（avg）

```
hive (default)> select avg(sal) avg_sal from emp;
```

hive sql 执行过程：



5.3.2 Group By 语句

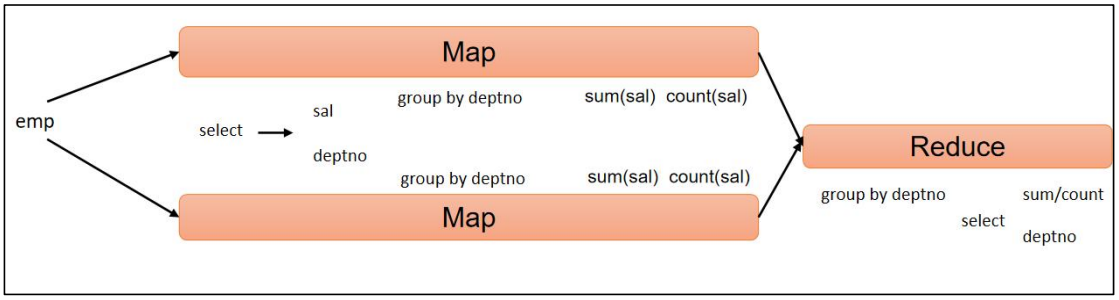
Group By 语句通常会和聚合函数一起使用，按照一个或者多个列队结果进行分组，然后对每个组执行聚合操作。

1）案例实操

（1）计算 emp 表每个部门的平均工资。

```
hive (default)>
select
    t.deptno,
    avg(t.sal) avg_sal
from emp t
group by t.deptno;
```

hive sql 执行过程：

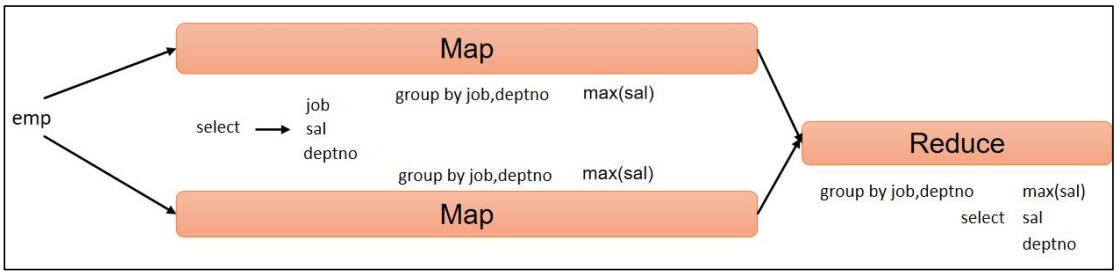


（2）计算 emp 每个部门中每个岗位的最高薪水。

```
hive (default)>
select
```

```
    t.deptno,
    t.job,
    max(t.sal) max_sal
from emp t
group by t.deptno, t.job;
```

hive sql 执行过程：



5.3.3 Having 语句

1）having 与 where 不同点

（1）where 后面不能写分组聚合函数，而 having 后面可以使用分组聚合函数。

（2）having 只用于 group by 分组统计语句。

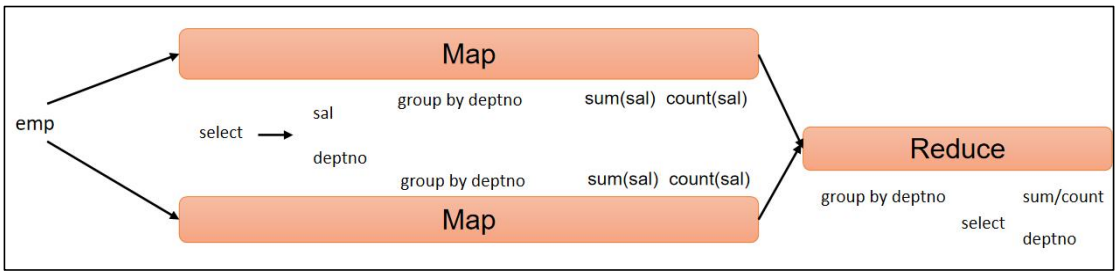
2）案例实操

（1）求每个部门的平均薪水大于 2000 的部门

①求每个部门的平均工资。

```
hive (default)>
select
    deptno,
    avg(sal)
from emp
group by deptno;
```

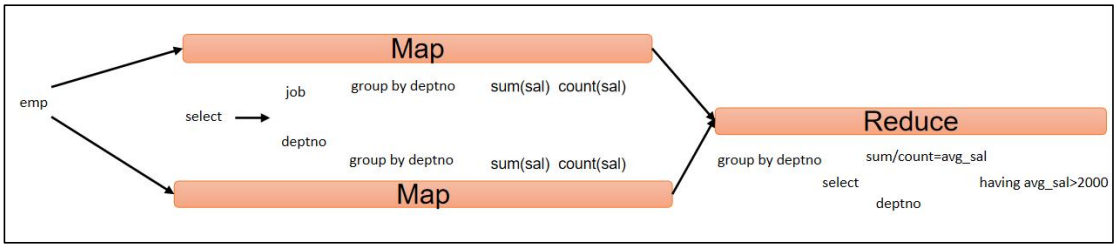
hive sql 执行过程：



②求每个部门的平均薪水大于 2000 的部门。

```
hive (default)>
select
    deptno,
    avg(sal) avg_sal
from emp
group by deptno
having avg_sal > 2000;
```

hive sql 执行过程：



5.4 Join 语句

5.4.1 等值 Join

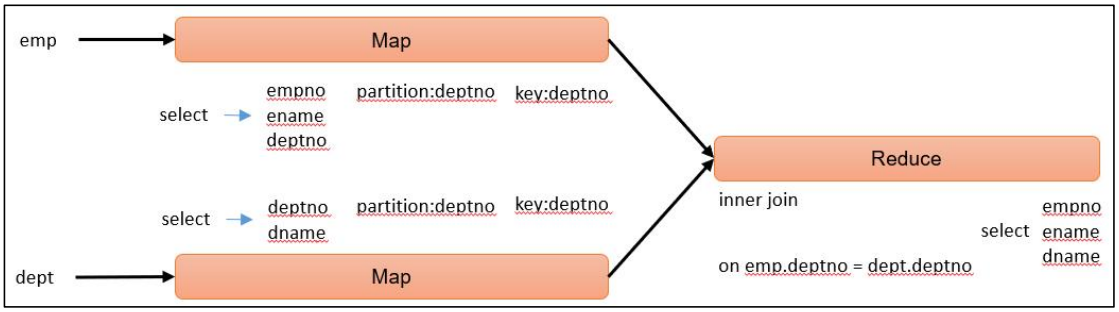
Hive 支持通常的 sql join 语句，但是只支持等值连接，不支持非等值连接。

1）案例实操

（1）根据员工表和部门表中的部门编号相等，查询员工编号、员工名称和部门名称。

```
hive (default)>
select
    e.empno,
    e.ename,
    d.dname
from emp e
join dept d
on e.deptno = d.deptno;
```

hive sql 执行过程：



5.4.2 表的别名

1）好处

（1）使用别名可以简化查询。

（2）区分字段的来源。

2）案例实操

合并员工表和部门表。

```
hive (default)>
select
```

```
    e.*,
    d.*
from emp e
join dept d
on e.deptno = d.deptno;
```

5.4.3 内连接

内连接：只有进行连接的两个表中都存在与连接条件相匹配的数据才会被保留下来。

```
hive (default)>
select
    e.empno,
    e.ename,
    d.deptno
from emp e
join dept d
on e.deptno = d.deptno;
```

5.4.4 左外连接

左外连接：join 操作符左边表中符合 where 子句的所有记录将会被返回。

```
hive (default)>
select
    e.empno,
    e.ename,
    d.deptno
from emp e
left join dept d
on e.deptno = d.deptno;
```

5.4.5 右外连接

右外连接：join 操作符右边表中符合 where 子句的所有记录将会被返回。

```
hive (default)>
select
    e.empno,
    e.ename,
    d.deptno
from emp e
right join dept d
on e.deptno = d.deptno;
```

5.4.6 满外连接

满外连接：将会返回所有表中符合 where 语句条件的所有记录。如果任一表的指定字段没有符合条件的值的话，那么就使用 null 值替代。

```
hive (default)>
select
    e.empno,
    e.ename,
    d.deptno
from emp e
```

```
full join dept d
on e.deptno = d.deptno;
```

5.4.7 多表连接

注意：连接 n 个表，至少需要 n-1 个连接条件。例如：连接三个表，至少需要两个连接条件。

数据准备，在/opt/module/hive/datas/下：vim location.txt

部门位置 id 部门位置

```
[atguigu@hadoop102 datas]$ vim location.txt

1700   北京
1800   上海
1900   深圳
```

1）创建位置表

```
hive (default)>
create table if not exists location(
    loc int,            -- 部门位置 id
    loc_name string    -- 部门位置
)
row format delimited fields terminated by '\t';
```

2）导入数据

```
hive (default)> load data local inpath
'/opt/module/hive/datas/location.txt' into table location;
```

3）多表连接查询

```
hive (default)>
select
    e.ename,
    d.dname,
    l.loc_name
from emp e
join dept d
on d.deptno = e.deptno
join location l
on d.loc = l.loc;
```

大多数情况下，Hive 会对每对 join 连接对象启动一个 MapReduce 任务。本例中会首先启动一个 MapReduce job 对表 e 和表 d 进行连接操作，然后会再启动一个 MapReduce job 将第一个 MapReduce job 的输出和表 l 进行连接操作。

注意：为什么不是表 d 和表 l 先进行连接操作呢？这是因为 Hive 总是按照从左到右的顺序执行的。

5.4.8 笛卡尔集

1）笛卡尔集会在下面条件下产生

（1）省略连接条件

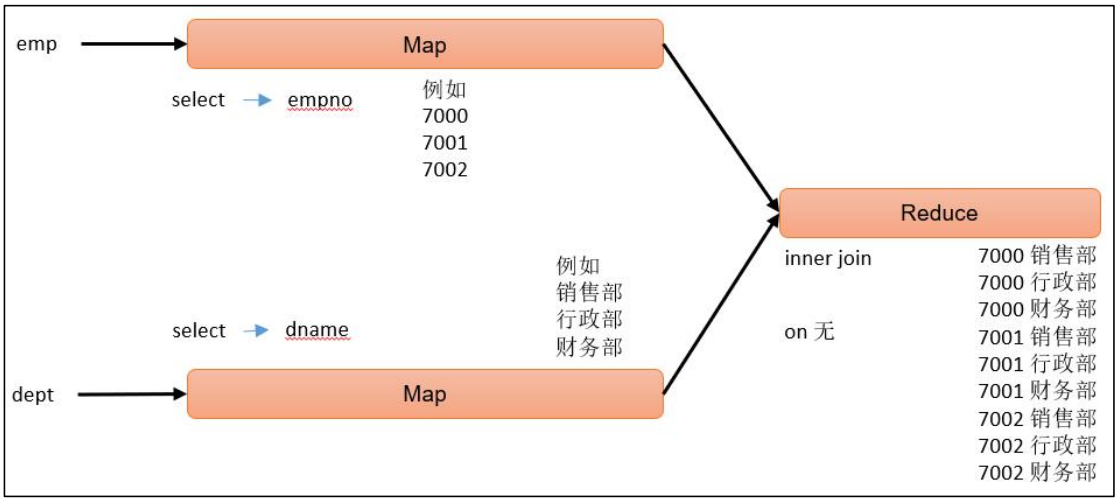
（2）连接条件无效

（3）所有表中的所有行互相连接

2）案例实操

```
hive (default)>
select
    empno,
    dname
from emp, dept;
```

hive sql 执行过程：



5.4.9 联合（union & union all）

1）union&union all 上下拼接

union 和 union all 都是上下拼接 sql 的结果，这点是和 join 有区别的，join 是左右关联，union 和 union all 是上下拼接。union 去重，union all 不去重。

union 和 union all 在上下拼接 sql 结果时有两个要求：

（1）两个 sql 的结果，列的个数必须相同。

（2）两个 sql 的结果，上下所对应列的类型必须一致。

2）案例实操

将员工表 30 部门的员工信息和 40 部门的员工信息，利用 union 进行拼接显示。

```
hive (default)>
select
    *
from emp
where deptno=30
union
select
```

```
    *
from emp
where deptno=40;
```

5.5 综合案例练习——基础查询（8 道题）

尚硅谷大数据之Hive SQL题库-初级.c

第 6 章 函数初级

6.1 函数简介

Hive 会将常用的逻辑封装成**函数**给用户进行使用，类似于 Java 中的函数。

好处：避免用户反复写逻辑，可以直接拿来使用。

重点：用户需要知道函数叫什么，能做什么。

Hive 提供了大量的内置函数，按照其特点可大致分为如下几类：单行函数、聚合函数、炸裂函数、窗口函数。

以下命令可用于查询所有内置函数的相关信息。

1）查看系统内置函数

```
hive> show functions;
```

2）查看内置函数用法

```
hive> desc function upper;
```

3）查看内置函数详细信息

```
hive> desc function extended upper;
```

6.2 单行函数

单行函数的特点是一进一出，即输入一行，输出一行。

单行函数按照功能可分为如下几类: 日期函数、字符串函数、集合函数、数学函数、流程控制函数等。

6.2.1 算术运算函数

运算符	描述
A+B	A 和 B 相加
A-B	A 减去 B
A*B	A 和 B 相乘
A/B	A 除以 B
A%B	A 对 B 取余
A&B	A 和 B 按位取与

A|B	A 和 B 按位取或
A^B	A 和 B 按位取异或
~A	A 按位取反

案例实操：查询出所有员工的薪水后加 1 显示。

```
hive (default)> select sal + 1 from emp;
```

6.2.2 数值函数

1）round：四舍五入

```
hive> select round(3.3);   3
```

2）ceil：向上取整

```
hive> select ceil(3.1) ;   4
```

3）floor：向下取整

```
hive> select floor(4.8);  4
```

6.2.3 字符串函数

1）substring：截取字符串

语法一：substring(string A, int start)

返回值：string

说明：返回字符串 A 从 start 位置到结尾的字符串

语法二：substring(string A, int start, int len)

返回值：string

说明：返回字符串 A 从 start 位置开始，长度为 len 的字符串

案例实操：

（1）获取第二个字符以后的所有字符

```
hive> select substring("atguigu",2);
```

输出：

```
tguigu
```

（2）获取倒数第三个字符以后的所有字符

```
hive> select substring("atguigu",-3);
```

输出：

```
igu
```

（3）从第 3 个字符开始，向后获取 2 个字符

```
hive> select substring("atguigu",3,2);
```

输出：

```
gu
```

2）replace ：替换

——

语法：replace(string A, string B, string C)

返回值：string

说明：将字符串 A 中的子字符串 B 替换为 C。

```
hive> select replace('atguigu', 'a', 'A')
```

输出：

```
hive> Atguigu
```

3）regexp_replace：正则替换

语法：regexp_replace(string A, string B, string C)

返回值：string

说明：将字符串 A 中的符合 java 正则表达式 B 的部分替换为 C。注意，在有些情况下要使用转义字符。

案例实操：

```
hive> select regexp_replace('100-200', '(\\d+)', 'num')
```

输出：

```
hive> num-num
```

4）regexp：正则匹配

语法：字符串 regexp 正则表达式

返回值：boolean

说明：若字符串符合正则表达式，则返回 true，否则返回 false。

（1）正则匹配成功，输出 true

```
hive> select 'dfsaaaa' regexp 'dfsa+'
```

输出：

```
hive> true
```

（2）正则匹配失败，输出 false

```
hive> select 'dfsaaaa' regexp 'dfsb+';
```

输出：

```
hive> false
```

5）repeat：重复字符串

语法：repeat(string A, int n)

返回值：string

说明：将字符串 A 重复 n 遍。

```
hive> select repeat('123', 3);
```

输出：

```
hive> 123123123
```

——

6）split ：字符串切割

语法：split(string str, string pat)

返回值：array

说明：按照正则表达式 pat 匹配到的内容分割 str，分割后的字符串，以数组的形式返回。

```
hive> select split('a-b-c-d','-');
```

输出：

```
hive> ["a","b","c","d"]
```

7）nvl ：替换 null 值

语法：nvl(A,B)

说明：若 A 的值不为 null，则返回 A，否则返回 B。

```
hive> select nvl(null,1);
```

输出：

```
hive> 1
```

8）concat ：拼接字符串

语法：concat(string A, string B, string C, ……)

返回：string

说明：将 A,B,C……等字符拼接为一个字符串

```
hive> select concat('beijing','-','shanghai','-','shenzhen');
```

输出：

```
hive> beijing-shanghai-shenzhen
```

9）concat_ws：以指定分隔符拼接字符串或者字符串数组

语法：concat_ws(string A, string… | array(string))

返回值：string

说明：使用分隔符 A 拼接多个字符串，或者一个数组的所有元素。

```
hive>select concat_ws('-','beijing','shanghai','shenzhen');
```

输出：

```
hive> beijing-shanghai-shenzhen
```

```
hive> select concat_ws('-
',array('beijing','shenzhen','shanghai'));
```

输出：

```
hive> beijing-shanghai-shenzhen
```

10）get_json_object：解析 json 字符串

语法：get_json_object(string json_string, string path)

返回值：string

说明：解析 json 的字符串 json_string，返回 path 指定的内容。如果输入的 json 字符串无效，那么返回 NULL。

案例实操：

（1）获取 json 数组里面的 json 具体数据

```
hive> select get_json_object('[{"name":"大海海","sex":"男
","age":"25"},{"name":"小宋宋","sex":"男
","age":"47"}]','$.[0].name');
```

输出：

```
hive> 大海海
```

（2）获取 json 数组里面的数据

```
hive> select get_json_object('[{"name":"大海海","sex":"男
","age":"25"},{"name":"小宋宋","sex":"男","age":"47"}]','$.[0]');
```

输出：

```
hive> {"name":"大海海","sex":"男","age":"25"}
```

6.2.4 日期函数

1）unix_timestamp：返回当前或指定时间的时间戳

语法：unix_timestamp()

返回值：bigint

案例实操：

```
hive> select unix_timestamp('2022/08/08 08-08-08','yyyy/MM/dd HH-
mm-ss');
```

输出：

```
1659946088
```

说明：-前面是日期后面是指，日期传进来的具体格式

2）from_unixtime：转化 UNIX 时间戳（从 1970-01-01 00:00:00 UTC 到指定时间的秒数）到当前时区的时间格式

语法：from_unixtime(bigint unixtime[, string format])

返回值：string

案例实操：

```
hive> select from_unixtime(1659946088);
```

输出：

```
2022-08-08 08:08:08
```

——

3）current_date：当前日期

```
hive> select current_date;
```

输出：

```
2022-07-11
```

4）current_timestamp：当前的日期加时间，并且精确的毫秒

```
hive> select current_timestamp;
```

输出：

```
2022-07-11 15:32:22.402
```

5）month：获取日期中的月

语法：month (string date)

返回值：int

案例实操：

```
hive> select month('2022-08-08 08:08:08');
```

输出：

```
8
```

6）day：获取日期中的日

语法：day (string date)

返回值：int

案例实操：

```
hive> select day('2022-08-08 08:08:08')
```

输出：

```
8
```

7）hour：获取日期中的小时

语法：hour (string date)

返回值：int

案例实操：

```
hive> select hour('2022-08-08 08:08:08');
```

输出：

```
8
```

8）datediff：两个日期相差的天数（结束日期减去开始日期的天数）

语法：datediff(string enddate, string startdate)

返回值：int

案例实操：

```
hive> select datediff('2021-08-08','2022-10-09');
```

——

输出：

```
-427
```

9）date_add：日期加天数

语法：date_add(string startdate, int days)

返回值：string

说明：返回开始日期 startdate 增加 days 天后的日期

案例实操：

```
hive> select date_add('2022-08-08',2);
```

输出：

```
2022-08-10
```

10）date_sub：日期减天数

语法：date_sub (string startdate, int days)

返回值：string

说明：返回开始日期 startdate 减少 days 天后的日期。

案例实操：

```
hive> select date_sub('2022-08-08',2);
```

输出：

```
2022-08-06
```

11）date_format:将标准日期解析成指定格式字符串

```
hive> select date_format('2022-08-08','yyyy 年-MM 月-dd 日')
```

输出：

```
2022 年-08 月-08 日
```

6.2.5 流程控制函数

1）case when：条件判断函数

语法一：case when a then b [when c then d]* [else e] end

返回值：T

说明：如果 a 为 true，则返回 b；如果 c 为 true，则返回 d；否则返回 e

```
hive> select case when 1=2 then 'tom' when 2=2 then 'mary' else
'tim' end from tabl eName;
mary
```

语法二： case a when b then c [when d then e]* [else f] end

返回值: T

说明：如果 a 等于 b，那么返回 c；如果 a 等于 d，那么返回 e；否则返回 f

```
hive> select case 100 when 50 then 'tom' when 100 then 'mary'
```

```
else 'tim' end from t ableName;
mary
```

2）if: 条件判断，类似于 Java 中三元运算符

语法：if（boolean testCondition, T valueTrue, T valueFalseOrNull）

返回值：T

说明：当条件 testCondition 为 true 时，返回 valueTrue；否则返回 valueFalseOrNull

（1）条件满足，输出正确

```
hive> select if(10 > 5,'正确','错误');
```

输出：正确

（2）条件满足，输出错误

```
hive> select if(10 < 5,'正确','错误');
```

输出：错误

6.2.6 集合函数

1）array 声明 array 集合

语法：array(val1, val2, …)

说明：根据输入的参数构建数组 array 类

案例实操：

```
hive> select array('1','2','3','4');
```

输出：

```
hive>["1","2","3","4"]
```

2）array_contains: 判断 array 中是否包含某个元素

```
hive> select array_contains(array('a','b','c','d'),'a');
```

输出：

```
hive> true
```

3）sort_array：将 array 中的元素排序

```
hive> select sort_array(array('a','d','c'));
```

输出：

```
hive> ["a","c","d"]
```

4）size：集合中元素的个数

```
hive> select size(friends) from test;  --2/2  每一行数据中的 friends 集合里的个数
```

5）map：创建 map 集合

语法：map (key1, value1, key2, value2, …)

说明：根据输入的 key 和 value 对构建 map 类型

——

案例实操：

```
hive> select map('xiaohai',1,'dahai',2);
```

输出：

```
hive> {"xiaohai":1,"dahai":2}
```

6）map_keys： 返回 map 中的 key

```
hive> select map_keys(map('xiaohai',1,'dahai',2));
```

输出：

```
hive>["xiaohai","dahai"]
```

7）map_values: 返回 map 中的 value

```
hive> select map_values(map('xiaohai',1,'dahai',2));
```

输出：

```
hive>[1,2]
```

8）struct 声明 struct 中的各属性

语法：struct(val1, val2, val3, …)

说明：根据输入的参数构建结构体 struct 类

案例实操：

```
hive> select struct('name','age','weight');
```

输出：

```
hive> {"col1":"name","col2":"age","col3":"weight"}
```

9）named_struct 声明 struct 的属性和值

```
hive> select named_struct('name','xiaosong','age',18,'weight',80);
```

输出：

```
hive> {"name":"xiaosong","age":18,"weight":80}
```

6.2.7 案例演示

1. 数据准备

1）表结构

name	sex	birthday	hiredate	job	salary	bonus	friends	children
张无忌	男	1980/02/12	2022/08/09	销售	3000	12000	[阿朱，小昭]	{张小无:8,张小忌:9}
赵敏	女	1982/05/18	2022/09/10	行政	9000	2000	[阿三，阿四]	{赵小敏:8}
黄蓉	女	1982/04/13	2022/06/11	行政	12000	Null	[东邪，西毒]	{郭芙:5,郭襄:4}

2）建表语句

```
hive>
create  table  employee(
    name string,  --姓名
    sex  string,  --性别
    birthday string, --出生年月
```

```
    hiredate string, --入职日期
    job string,   --岗位
    salary double, --薪资
    bonus double,  --奖金
    friends array<string>, --朋友
    children map<string,int> --孩子
)
```

3）插入数据

```
hive> insert into employee
  values('张无忌','男','1980/02/12','2022/08/09','销售
',3000,12000,array('阿朱','小昭'),map('张小无',8,'张小忌',9)),
        ('赵敏','女','1982/05/18','2022/09/10','行政
',9000,2000,array('阿三','阿四'),map('赵小敏',8)),
        ('宋青书','男','1981/03/15','2022/04/09','研发
',18000,1000,array('王五','赵六'),map('宋小青',7,'宋小书',5)),
        ('周芷若','女','1981/03/17','2022/04/10','研发
',18000,1000,array('王五','赵六'),map('宋小青',7,'宋小书',5)),
        ('郭靖','男','1985/03/11','2022/07/19','销售
',2000,13000,array('南帝','北丐'),map('郭芙',5,'郭襄',4)),
        ('黄蓉','女','1982/12/13','2022/06/11','行政
',12000,null,array('东邪','西毒'),map('郭芙',5,'郭襄',4)),
        ('杨过','男','1988/01/30','2022/08/13','前台
',5000,null,array('郭靖','黄蓉'),map('杨小过',2)),
        ('小龙女','女','1985/02/12','2022/09/24','前台
',6000,null,array('张三','李四'),map('杨小过',2))
```

2. 需求

1）统计每个月的入职人数

（1）期望结果

month	cnt
4	2
6	1
7	1
8	2
9	2

（2）需求实现

```
select
  month(replace(hiredate,'/','-')) as month,
  count(*) as cn
from
  employee
group by
  month(replace(hiredate,'/','-'))
```

2）查询每个人的年龄（年 + 月）

（1）期望结果

name	age
张无忌	42 年 8 月
赵敏	40 年 5 月
宋青书	41 年 7 月
周芷若	41 年 7 月
郭靖	37 年 7 月
黄蓉	39 年 10 月
杨过	34 年 9 月
小龙女	37 年 8 月

（2）需求实现

```
-- 转换日期
select
  name,
  replace(birthday,'/','-') birthday
from
  employee  t1

-- 求出年和月
select
  name,
  year(current_date())-year(t1.birthday) year,
  month(current_date())-month(t1.birthday) month
from
  (
    select
      name,
      replace(birthday,'/','-') birthday
    from
      employee
)t1 t2

-- 根据月份正负决定年龄

select
  name,
  concat(if(month>=0,year,year-1),'年
',if(month>=0,month,12+month),'月') age
from
  (
    select
      name,
      year(current_date())-year(t1.birthday) year,
```

```
      month(current_date())-month(t1.birthday) month
    from
      (
        select
          name,
          replace(birthday,'/','-') birthday
        from
          employee
    )t1
)t2
```

3）按照薪资，奖金的和进行倒序排序，如果奖金为 null，置位 0

（1）期望结果

name	sal
周芷若	19000
宋青书	19000
郭靖	15000
张无忌	15000
黄蓉	12000
赵敏	11000
小龙女	6000
杨过	5000

（2）需求实现

```
select
  name,
  salary + nvl(bonus,0) sal
from
  employee
order by
   sal desc
```

4）查询每个人有多少个朋友

（1）期望结果

name	cnt
张无忌	2
赵敏	2
宋青书	2
周芷若	2
郭靖	2
黄蓉	2

杨过	2
小龙女	2

（2）需求实现

```
select
    name,
    size(friends) cnt
from
    employee;
```

5）查询每个人的孩子的姓名

（1）期望结果

name	ch_name
张无忌	["张小无","张小忌"]
赵敏	["赵小敏"]
宋青书	["宋小青","宋小书"]
周芷若	["宋小青","宋小书"]
郭靖	["郭芙","郭襄"]
黄蓉	["郭芙","郭襄"]
杨过	["杨小过"]
小龙女	["杨小过"]

（2）需求实现

```
hive>
select
    name,
    map_keys(children) ch_name
from
    employee;
```

6）查询每个岗位男女各多少人

（1）期望结果

job	male	female
前台	1	1
研发	1	1
行政	0	2
销售	2	0

（2）需求实现

```
select
  job,
  sum(if(sex='男',1,0)) male,
```

```
  sum(if(sex='女',1,0)) female
from
  employee
group by
  job
```

6.3 高级聚合函数

多进一出（多行传入，一个行输出）。

1）普通聚合 count/sum.... 见第 6 章 6.2.4

2）collect_list 收集并形成 list 集合，结果不去重

```
hive>
select
  sex,
  collect_list(job)
from
  employee
group by
  sex
```

结果：

```
女  ["行政","研发","行政","前台"]
男  ["销售","研发","销售","前台"]
```

3）collect_set 收集并形成 set 集合，结果去重

```
hive>
select
  sex,
  collect_set(job)
from
  employee
group by
  sex
```

结果：

```
女  ["行政","研发","前台"]
男  ["销售","研发","前台"]
```

4）案例演示

（1）每个月的入职人数以及姓名

```
hive>
select
  month(replace(hiredate,'/','-')) as month,
  count(*) as cn,
  Collect_list(name) as name_list
from
  employee
group by
  month(replace(hiredate,'/','-'))
```

结果：

```
month   cn   name_list
4        2   ["宋青书","周芷若"]
6        1   ["黄蓉"]
7        1   ["郭靖"]
8        2   ["张无忌","杨过"]
9        2   ["赵敏","小龙女"]
```

6.4 综合案例练习——基础函数（6 道题目）

尚硅谷大数据之Hive SQL题库-中级.c

第 7 章 函数高级

7.1 炸裂函数

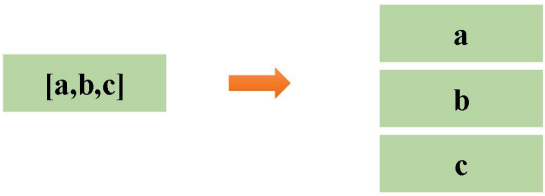
7.1.1 概述

UDTF

尚硅谷

定义

UDTF（Table-Generating Functions），接收一行数据，输出一行或多行数据。



7.1.2 案例演示

1）数据准备

（1）表结构

movie	category
《疑犯追踪》	悬疑，动作，科幻，剧情
《Lie to me》	悬疑，警匪，动作，心理，剧情
《战狼 2》	战争，动作，灾难

（2）建表语句

```
hive (default)>
create table movie_info(
    movie string,     --电影名称
    category string   --电影分类
)
row format delimited fields terminated by "\t";
```

（3）装载语句

```
insert overwrite table movie_info
values ("《疑犯追踪》", "悬疑,动作,科幻,剧情"),
       ("《Lie to me》", "悬疑,警匪,动作,心理,剧情"),
       ("《战狼 2》", "战争,动作,灾难");
```

2）需求

（1）需求说明：根据上述电影信息表，统计各分类的电影数量，期望结果如下：

剧情	2
动作	3
心理	1
悬疑	2
战争	1
灾难	1
科幻	1
警匪	1

（2）答案

```
select
    cate,
    count(*)
from
(
    select
        movie,
        cate
    from
    (
        select
            movie,
            split(category,',') cates
        from movie_info
    )t1 lateral view explode(cates) tmp as cate
)t2
group by cate;
```

7.2 窗口函数（开窗函数）

7.2.1 概述

窗口函数(window functions)

尚硅谷

定义

窗口函数，能为每行数据划分一个窗口，然后对窗口范围内的数据进行计算，最后将计算结果返回给该行数据。

7.2.2 常用窗口函数

按照功能，常用窗口可划分为如下几类：聚合函数、跨行取值函数、排名函数。

1）聚合函数

max：最大值。

min：最小值。

sum：求和。

avg：平均值。

count：计数。

2）跨行取值函数

（1）lead 和 lag

窗口函数(window functions)

常用窗口函数——lead和lag

功能：获取当前行的上/下边某行、某个字段的值。

语法：

```
select
  order_id,
  user_id,
  order_date,
  amount,
  lag(order_date,1, '1970-01-01') over (partition by user_id order by order_date) last_date,
  lead(order_date,1, '9999-12-31') over (partition by user_id order by order_date) next_date
from order_info;
```

字段名　偏移量　默认值

思考：该查询语句的结果是什么？

order_id	user_id	order_date	amount
1	1001	2022-01-01	10
2	1001	2022-01-02	20
3	1001	2022-01-03	10
4	1002	2022-01-04	30
5	1002	2022-01-05	40
6	1002	2022-01-06	20

order_id	user_id	order_date	amount	last_date	next_date
1	1001	2022-01-01	10	1970-01-01	2022-01-02
2	1001	2022-01-02	20	2022-01-01	2022-01-03
3	1001	2022-01-03	10	2022-01-02	9999-12-31
4	1002	2022-01-04	30	1970-01-01	2022-01-05
5	1002	2022-01-05	40	2022-01-04	2022-01-06
6	1002	2022-01-06	20	2022-01-05	9999-12-31

注：lag 和 lead 函数不支持自定义窗口。

（2）first_value 和 last_value

窗口函数(window functions)

常用窗口函数——first_value和last_value

功能：获取窗口内某一列的第一个值/最后一个值

语法：

```
select
  order_id,
  user_id,
  order_date,
  amount,
  first_value(order_date,false) over (partition by user_id order by order_date) first_date,
  last_value(order_date,false) over (partition by user_id order by order_date) last_date
from order_info;
```

字段名　是否跳过null

思考：该查询语句的结果是什么？

order_id	user_id	order_date	amount
1	1001	2022-01-01	10
2	1001	2022-01-02	20
3	1001	2022-01-03	10
4	1002	2022-01-04	30
5	1002	2022-01-05	40
6	1002	2022-01-06	20

order_id	user_id	order_date	amount	first_date	last_date
1	1001	2022-01-01	10	2022-01-01	2022-01-01
2	1001	2022-01-02	20	2022-01-01	2022-01-02
3	1001	2022-01-03	10	2022-01-01	2022-01-03
4	1002	2022-01-04	30	2022-01-04	2022-01-04
5	1002	2022-01-05	40	2022-01-04	2022-01-05
6	1002	2022-01-06	20	2022-01-04	2022-01-06

3）排名函数

窗口函数(window functions)

尚硅谷

常用窗口函数——rank、dense_rank、row_number

功能：计算排名

语法：

```
select
   stu_id,
   course,
   score,
   rank() over(partition by course order by score desc) rk,
   dense_rank() over(partition by course order by score desc) dense_rk,
   row_number() over(partition by course order by score desc) rn
from score_info;
```

stu_id	course	score
1	语文	99
2	语文	98
3	语文	95
4	数学	100
5	数学	100
6	数学	99

stu_id	course	score	rk	dense_rk	rn
1	语文	99	1	1	1
2	语文	98	2	2	2
3	语文	95	3	3	3
4	数学	100	1	1	1
5	数学	100	1	1	2
6	数学	99	3	2	3

注：**rank 、dense_rank、row_number 不支持自定义窗口。**

7.2.3 案例演示

1）数据准备

（1）表结构

order_id	user_id	user_name	order_date	order_amount
1	1001	小元	2022-01-01	10
2	1002	小海	2022-01-02	15
3	1001	小元	2022-02-03	23
4	1002	小海	2022-01-04	29
5	1001	小元	2022-01-05	46

（2）建表语句

```
create table order_info
(
    order_id     string, --订单 id
    user_id      string, -- 用户 id
    user_name    string, -- 用户姓名
    order_date   string, -- 下单日期
    order_amount int     -- 订单金额
);
```

（3）装载语句

```
insert overwrite table order_info
values ('1', '1001', '小元', '2022-01-01', '10'),
       ('2', '1002', '小海', '2022-01-02', '15'),
```

```
        ('3', '1001', '小元', '2022-02-03', '23'),
        ('4', '1002', '小海', '2022-01-04', '29'),
        ('5', '1001', '小元', '2022-01-05', '46'),
        ('6', '1001', '小元', '2022-04-06', '42'),
        ('7', '1002', '小海', '2022-01-07', '50'),
        ('8', '1001', '小元', '2022-01-08', '50'),
        ('9', '1003', '小辉', '2022-04-08', '62'),
        ('10', '1003', '小辉', '2022-04-09', '62'),
        ('11', '1004', '小猛', '2022-05-10', '12'),
        ('12', '1003', '小辉', '2022-04-11', '75'),
        ('13', '1004', '小猛', '2022-06-12', '80'),
        ('14', '1003', '小辉', '2022-04-13', '94');
```

2）需求

（1）统计每个用户截至每次下单的累积下单总额

①期望结果

order_id	user_id	user_name	order_date	order_amount	sum_so_far
1	1001	小元	2022-01-01	10	10
5	1001	小元	2022-01-05	46	56
8	1001	小元	2022-01-08	50	106
3	1001	小元	2022-02-03	23	129
6	1001	小元	2022-04-06	42	171
2	1002	小海	2022-01-02	15	15
4	1002	小海	2022-01-04	29	44
7	1002	小海	2022-01-07	50	94
9	1003	小辉	2022-04-08	62	62
10	1003	小辉	2022-04-09	62	124
12	1003	小辉	2022-04-11	75	199
14	1003	小辉	2022-04-13	94	293
11	1004	小猛	2022-05-10	12	12
13	1004	小猛	2022-06-12	80	92

②需求实现

```
select
    order_id,
    user_id,
    user_name,
    order_date,
    order_amount,
    sum(order_amount) over(partition by user_id order by
```

```
order_date rows between unbounded preceding and current row)
sum_so_far
from order_info;
```

（3）统计每个用户截至每次下单的当月累积下单总额

①期望结果

order_id	user_id	user_name	order_date	order_amount	sum_so_far
1	1001	小元	2022-01-01	10	10
5	1001	小元	2022-01-05	46	56
8	1001	小元	2022-01-08	50	106
3	1001	小元	2022-02-03	23	23
6	1001	小元	2022-04-06	42	42
2	1002	小海	2022-01-02	15	15
4	1002	小海	2022-01-04	29	44
7	1002	小海	2022-01-07	50	94
9	1003	小辉	2022-04-08	62	62
10	1003	小辉	2022-04-09	62	124
12	1003	小辉	2022-04-11	75	199
14	1003	小辉	2022-04-13	94	293
11	1004	小猛	2022-05-10	12	12
13	1004	小猛	2022-06-12	80	80

②需求实现

```
select
    order_id,
    user_id,
    user_name,
    order_date,
    order_amount,
    sum(order_amount) over(partition by
user_id,substring(order_date,1,7) order by order_date rows
between unbounded preceding and current row) sum_so_far
from order_info;
```

（3）统计每个用户每次下单距离上次下单相隔的天数（首次下单按 0 天算）

①期望结果

order_id	user_id	user_name	order_date	order_amount	diff
1	1001	小元	2022-01-01	10	0
5	1001	小元	2022-01-05	46	4

8	1001	小元	2022-01-08	50	3
3	1001	小元	2022-02-03	23	26
6	1001	小元	2022-04-06	42	62
2	1002	小海	2022-01-02	15	0
4	1002	小海	2022-01-04	29	2
7	1002	小海	2022-01-07	50	3
9	1003	小辉	2022-04-08	62	0
10	1003	小辉	2022-04-09	62	1
12	1003	小辉	2022-04-11	75	2
14	1003	小辉	2022-04-13	94	2
11	1004	小猛	2022-05-10	12	0
13	1004	小猛	2022-06-12	80	33

②需求实现

```
select
    order_id,
    user_id,
    user_name,
    order_date,
    order_amount,
    nvl(datediff(order_date,last_order_date),0) diff
from
(
    select
        order_id,
        user_id,
        user_name,
        order_date,
        order_amount,
        lag(order_date,1,null) over(partition by user_id order by
order_date) last_order_date
    from order_info
)t1
```

（4）查询所有下单记录以及每个用户的每个下单记录所在月份的首/末次下单日期

①期望结果

order_id	user_id	user_name	order_date	order_amount	first_date	last_date
1	1001	小元	2022-01-01	10	2022-01-01	2022-01-08
5	1001	小元	2022-01-05	46	2022-01-01	2022-01-08
8	1001	小元	2022-01-08	50	2022-01-01	2022-01-08
3	1001	小元	2022-02-03	23	2022-02-03	2022-02-03

6	1001	小元	2022-04-06	42	2022-04-06	2022-04-06
2	1002	小海	2022-01-02	15	2022-01-02	2022-01-07
4	1002	小海	2022-01-04	29	2022-01-02	2022-01-07
7	1002	小海	2022-01-07	50	2022-01-02	2022-01-07
9	1003	小辉	2022-04-08	62	2022-04-08	2022-04-13
10	1003	小辉	2022-04-09	62	2022-04-08	2022-04-13
12	1003	小辉	2022-04-11	75	2022-04-08	2022-04-13
14	1003	小辉	2022-04-13	94	2022-04-08	2022-04-13
11	1004	小猛	2022-05-10	12	2022-05-10	2022-05-10
13	1004	小猛	2022-06-12	80	2022-06-12	2022-06-12

②需求实现

```
select
    order_id,
    user_id,
    user_name,
    order_date,
    order_amount,
    first_value(order_date) over(partition by
user_id,substring(order_date,1,7) order by order_date) first_date,
    last_value(order_date) over(partition by
user_id,substring(order_date,1,7) order by order_date rows
between unbounded preceding and unbounded following) last_date
from order_info;
```

（5）为每个用户的所有下单记录按照订单金额进行排名

①期望结果

order_id	user_id	user_name	order_date	order_amount	rk	drk	rn
8	1001	小元	2022-01-08	50	1	1	1
5	1001	小元	2022-01-05	46	2	2	2
6	1001	小元	2022-04-06	42	3	3	3
3	1001	小元	2022-02-03	23	4	4	4
1	1001	小元	2022-01-01	10	5	5	5
7	1002	小海	2022-01-07	50	1	1	1
4	1002	小海	2022-01-04	29	2	2	2
2	1002	小海	2022-01-02	15	3	3	3
14	1003	小辉	2022-04-13	94	1	1	1
12	1003	小辉	2022-04-11	75	2	2	2

9	1003	小辉	2022-04-08	62	3	3	3
10	1003	小辉	2022-04-09	62	3	3	4
13	1004	小猛	2022-06-12	80	1	1	1
11	1004	小猛	2022-05-10	12	2	2	2

②需求实现

```
select
    order_id,
    user_id,
    user_name,
    order_date,
    order_amount,
    rank() over(partition by user_id order by order_amount desc)
rk,
    dense_rank() over(partition by user_id order by order_amount
desc) drk,
    row_number() over(partition by user_id order by order_amount
desc) rn
from order_info;
```

7.3 自定义函数

1）Hive 自带了一些函数，比如：max/min 等，但是数量有限，自己可以通过自定义 UDF 来方便的扩展。

2）当 Hive 提供的内置函数无法满足你的业务处理需要时，此时就可以考虑使用用户自定义函数（UDF：user-defined function）。

3）根据用户自定义函数类别分为以下三种：

（1）UDF（User-Defined-Function）

一进一出。

（2）UDAF（User-Defined Aggregation Function）

用户自定义聚合函数，多进一出。

类似于：count/max/min

（3）UDTF（User-Defined Table-Generating Functions）

用户自定义表生成函数，一进多出。

如 lateral view explode()

4）官方文档地址

https://cwiki.apache.org/confluence/display/Hive/HivePlugins

5）编程步骤

——

（1）继承 Hive 提供的类

org.apache.hadoop.hive.ql.udf.generic.GenericUDF

org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;

（2）实现类中的抽象方法

（3）在 hive 的命令行窗口创建函数

添加 jar。

```
add jar linux_jar_path
```

创建 function。

```
create [temporary] function [dbname.]function_name AS class_name;
```

（4）在 hive 的命令行窗口删除函数

```
drop [temporary] function [if exists] [dbname.]function_name;
```

7.4 自定义 UDF 函数

0）需求

自定义一个 UDF 实现计算给定基本数据类型的长度，例如：

```
hive(default)> select my_len("abcd");
4
```

1）创建一个 Maven 工程 Hive

2）导入依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.hive</groupId>
        <artifactId>hive-exec</artifactId>
        <version>3.1.3</version>
    </dependency>
</dependencies>
```

3）创建一个类

```
package com.atguigu.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentLengthException;
import org.apache.hadoop.hive.ql.exec.UDFArgumentTypeException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDF;
import
org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import
org.apache.hadoop.hive.serde2.objectinspector.primitive.Primitive
ObjectInspectorFactory;

/**
 * 我们需计算一个要给定基本数据类型的长度
 */
```

```
public class MyUDF extends GenericUDF {
    /**
     * 判断传进来的参数的类型和长度
     * 约定返回的数据类型
     */
    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments)
throws UDFArgumentException {

        if (arguments.length !=1) {
            throw  new UDFArgumentLengthException("please give me
only one arg");
        }

        if
(!arguments[0].getCategory().equals(ObjectInspector.Category.PRIM
ITIVE)){
            throw  new UDFArgumentTypeException(1, "i need
primitive type arg");
        }

        return
PrimitiveObjectInspectorFactory.javaIntObjectInspector;
    }

    /**
     * 解决具体逻辑的
     */
    @Override
    public Object evaluate(DeferredObject[] arguments) throws
HiveException {

        Object o = arguments[0].get();
        if(o==null){
            return 0;
        }

        return o.toString().length();
    }

    @Override
    // 用于获取解释的字符串
    public String getDisplayString(String[] children) {
        return "";
    }
}
```

4）创建临时函数

（1）打成 jar 包上传到服务器/opt/module/hive/datas/myudf.jar

（2）将 jar 包添加到 hive 的 classpath，临时生效

```
hive (default)> add jar /opt/module/hive/datas/myudf.jar;
```

（3）创建临时函数与开发好的 java class 关联

```
hive (default)>
create temporary function my_len
as "com.atguigu.hive.udf.MyUDF";
```

（4）即可在 hql 中使用自定义的临时函数

```
hive (default)>
select
    ename,
    my_len(ename) ename_len
from emp;
```

（5）删除临时函数

```
hive (default)> drop temporary function my_len;
```

注意：临时函数只跟会话有关系，跟库没有关系。只要创建临时函数的会话不断，在当前会话下，任意一个库都可以使用，其他会话全都不能使用。

5）创建永久函数

（1）创建永久函数

注意：因为 add jar 本身也是临时生效，所以在创建永久函数的时候，需要制定路径（并且因为元数据的原因，这个路径还得是 HDFS 上的路径）。

```
hive (default)>
create function my_len2
as "com.atguigu.hive.udf.MyUDF"
using jar "hdfs://hadoop102:8020/udf/myudf.jar";
```

（2）即可在 hql 中使用自定义的永久函数

```
hive (default)>
select
    ename,
    my_len2(ename) ename_len
from emp;
```

（3）删除永久函数

```
hive (default)> drop function my_len2;
```

注意：永久函数跟会话没有关系，创建函数的会话断了以后，其他会话也可以使用。

永久函数创建的时候，在函数名之前需要自己加上库名，如果不指定库名的话，会默认把当前库的库名给加上。

永久函数使用的时候，需要在指定的库里面操作，或者在其他库里面使用的话加上，**库名.函数名。**

7.5 常用函数大全（附录）

尚硅谷大数据技术
之Hive常用函数大全

7.6 综合案例练习——高级函数（6 道题目）

尚硅谷大数据技术
之Hive SQL题库（

7.7 综合案例练习——大厂真题（4 道题目）

尚硅谷大数据技术
之Hive SQL题库（

第 8 章 分区表和分桶表

8.1 分区表

Hive 中的分区就是把一张大表的数据按照业务需要分散的存储到多个目录，每个目录就称为该表的一个分区。在查询时通过 where 子句中的表达式选择查询所需要的分区，这样的查询效率会提高很多。

8.1.1 分区表基本语法

1）创建分区表

```
hive (default)>
create table dept_partition(
    deptno int,    --部门编号
    dname  string, --部门名称
    loc    string  --部门位置
)
partitioned by (day string)
row format delimited fields terminated by '\t';
```

2）分区表写数据

（1）load

①数据准备

在/opt/module/hive/datas/路径上创建文件 dept_20220401.log，并输入如下内容。

```
[atguigu@hadoop102 datas]$ vim dept_20220401.log

10  行政部  1700
20  财务部  1800
```

②装载语句

```
hive (default)>
load data local inpath '/opt/module/hive/datas/dept_20220401.log'
```

```
into table dept_partition
partition(day='20220401');
```

（2）insert

将 **day='20220401'**分区的数据插入到 **day='20220402'**分区，可执行如下装载语句

```
hive (default)>
insert overwrite table dept_partition partition (day = '20220402')
select deptno, dname, loc
from dept_partition
where day = '2020-04-01';
```

3）分区表读数据

查询分区表数据时，可以将分区字段看作表的**伪列**，可像使用其他字段一样使用分区字段。

```
select deptno, dname, loc ,day
from dept_partition
where day = '2020-04-01';
```

4）分区表基本操作

（1）查看所有分区信息

```
hive> show partitions dept_partition;
```

（2）增加分区

①创建单个分区

```
hive (default)>
alter table dept_partition
add partition(day='20220403');
```

②同时创建多个分区（分区之间不能有逗号）

```
hive (default)>
alter table dept_partition
add partition(day='20220404') partition(day='20220405');
```

（3）删除分区

①删除单个分区

```
hive (default)>
alter table dept_partition
drop partition (day='20220403');
```

②同时删除多个分区（分区之间必须有逗号）

```
hive (default)>
alter table dept_partition
drop partition (day='20220404'), partition(day='20220405');
```

（4）修复分区

Hive 将分区表的所有分区信息都保存在了元数据中，只有元数据与 HDFS 上的分区路径一致时，分区表才能正常读写数据。若用户手动创建/删除分区路径，Hive 都是感知不到的，这样就会导致 Hive 的元数据和 HDFS 的分区路径不一致。再比如，若分区表为外部表，

用户执行 drop partition 命令后，分区元数据会被删除，而 HDFS 的分区路径不会被删除，同样会导致 Hive 的元数据和 HDFS 的分区路径不一致。

若出现元数据和 HDFS 路径不一致的情况，可通过如下几种手段进行修复。

①add partition

若手动创建 HDFS 的分区路径，Hive 无法识别，可通过 add partition 命令增加分区元数据信息，从而使元数据和分区路径保持一致。

②drop partition

若手动删除 HDFS 的分区路径，Hive 无法识别，可通过 drop partition 命令删除分区元数据信息，从而使元数据和分区路径保持一致。

③msck

若分区元数据和 HDFS 的分区路径不一致，还可使用 msck 命令进行修复，以下是该命令的用法说明。

```
hive (default)>
msck repair table table_name [add/drop/sync partitions];
```

说明：

msck repair table table_name add partitions：该命令会增加 HDFS 路径存在但元数据缺失的分区信息。

msck repair table table_name drop partitions：该命令会删除 HDFS 路径已经删除但元数据仍然存在的分区信息。

msck repair table table_name sync partitions：该命令会同步 HDFS 路径和元数据分区信息，相当于同时执行上述的两个命令。

msck repair table table_name：等价于 msck repair table table_name **add** partitions 命令。

8.1.2 二级分区表

思考：如果一天内的日志数据量也很大，如何再将数据拆分?答案是二级分区表，例如可以在按天分区的基础上，再对每天的数据按小时进行分区。

1）二级分区表建表语句

```
hive (default)>
create table dept_partition2(
    deptno int,     -- 部门编号
    dname string, -- 部门名称
    loc string      -- 部门位置
)
partitioned by (day string, hour string)
```

```
row format delimited fields terminated by '\t';
```

2）数据装载语句

```
hive (default)>
load data local inpath '/opt/module/hive/datas/dept_20220401.log'
into table dept_partition2
partition(day='20220401', hour='12');
```

3）查询分区数据

```
hive (default)>
select
    *
from dept_partition2
where day='20220401' and hour='12';
```

8.1.3 动态分区

动态分区是指向分区表 insert 数据时，被写往的分区不由用户指定，而是由每行数据的最后一个字段的值来动态的决定。使用动态分区，可只用一个 insert 语句将数据写入多个分区。

1）动态分区相关参数

（1）动态分区功能总开关（默认 true，开启）

```
set hive.exec.dynamic.partition=true
```

（2）严格模式和非严格模式

动态分区的模式，默认 strict（严格模式），要求必须指定至少一个分区为静态分区，nonstrict（非严格模式）允许所有的分区字段都使用动态分区。

```
set hive.exec.dynamic.partition.mode=nonstrict
```

（3）一条 insert 语句可同时创建的最大的分区个数，默认为 1000。

```
set hive.exec.max.dynamic.partitions=1000
```

（4）单个 Mapper 或者 Reducer 可同时创建的最大的分区个数，默认为 100。

```
set hive.exec.max.dynamic.partitions.pernode=100
```

（5）一条 insert 语句可以创建的最大的文件个数，默认 100000。

```
hive.exec.max.created.files=100000
```

（6）当查询结果为空时且进行动态分区时，是否抛出异常，默认 false。

```
hive.error.on.empty.partition=false
```

2）案例实操

需求：将 dept 表中的数据按照地区（loc 字段），插入到目标表 dept_partition_dynamic 的相应分区中。

（1）创建目标分区表

```
hive (default)>
create table dept_partition_dynamic(
    id int,
```

```
    name string
)
partitioned by (loc int)
row format delimited fields terminated by '\t';
```

（2）设置动态分区

```
set hive.exec.dynamic.partition.mode = nonstrict;
hive (default)>
insert into table dept_partition_dynamic
partition(loc)
select
    deptno,
    dname,
    loc
from dept;
```

（3）查看目标分区表的分区情况

```
hive (default)> show partitions dept_partition_dynamic;
```

8.2 分桶表

分区提供一个隔离数据和优化查询的便利方式。不过，并非所有的数据集都可形成合理的分区。对于一张表或者分区，Hive 可以进一步组织成桶，也就是更为细粒度的数据范围划分，分区针对的是数据的存储路径，分桶针对的是数据文件。

分桶表的基本原理是，首先为每行数据计算一个指定字段的数据的 hash 值，然后模以一个指定的分桶数，最后将取模运算结果相同的行，写入同一个文件中，这个文件就称为一个分桶（bucket）。

8.2.1 分桶表基本语法

1）建表语句

```
hive (default)>
create table stu_buck(
    id int,
    name string
)
clustered by(id)
into 4 buckets
row format delimited fields terminated by '\t';
```

2）数据装载

（1）数据准备

在/opt/module/hive/datas/路径上创建 student.txt 文件，并输入如下内容。

```
1001   student1
1002   student2
1003   student3
1004   student4
1005   student5
```

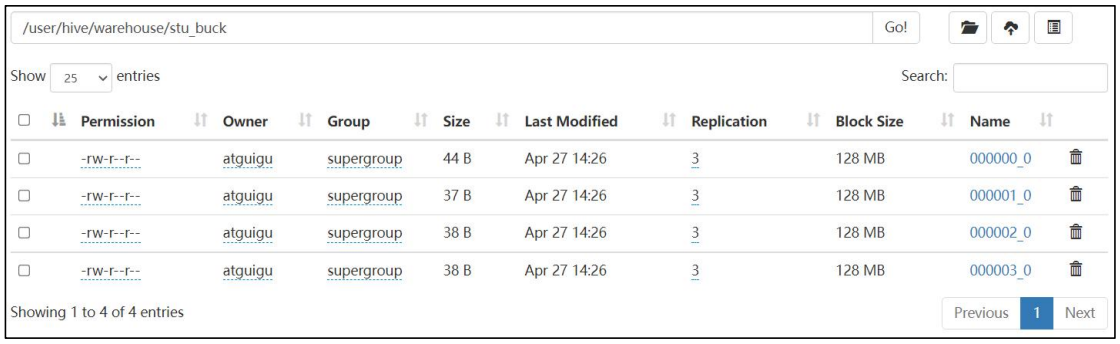
```
1006   student6
1007   student7
1008   student8
1009   student9
1010   student10
1011   student11
1012   student12
1013   student13
1014   student14
1015   student15
1016   student16
```

（2）导入数据到分桶表中

说明：Hive 新版本 load 数据可以直接跑 MapReduce，老版的 Hive 需要将数据传到一张表里，再通过查询的方式导入到分桶表里面。

```
hive (default)>
load data local inpath '/opt/module/hive/datas/student.txt'
into table stu_buck;
```

（3）查看创建的分桶表中是否分成 4 个桶



（4）观察每个分桶中的数据

8.2.2 分桶排序表

1）建表语句

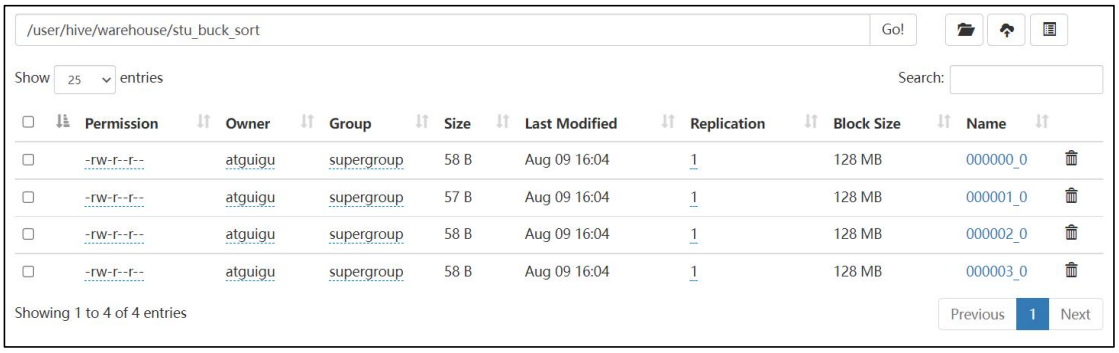
```
hive (default)>
create table stu_buck_sort(
    id int,
    name string
)
clustered by(id) sorted by(id)
into 4 buckets
row format delimited fields terminated by '\t';
```

2）数据装载

（1）导入数据到分桶表中

```
hive (default)>
load data local inpath '/opt/module/hive/datas/student.txt'
into table stu_buck_sort;
```

（2）查看创建的分桶表中是否分成 4 个桶



（3）观察每个分桶中的数据

第 9 章 文件格式和压缩

9.1 文件格式

为 Hive 表中的数据选择一个合适的文件格式，对提高查询性能的提高是十分有益的。Hive 表数据的存储格式，可以选择 text file、orc、parquet、sequence file 等。

9.1.1 Text File

文本文件是 Hive 默认使用的文件格式，文本文件中的一行内容，就对应 Hive 表中的一行记录。

可通过以下建表语句指定文件格式为文本文件：

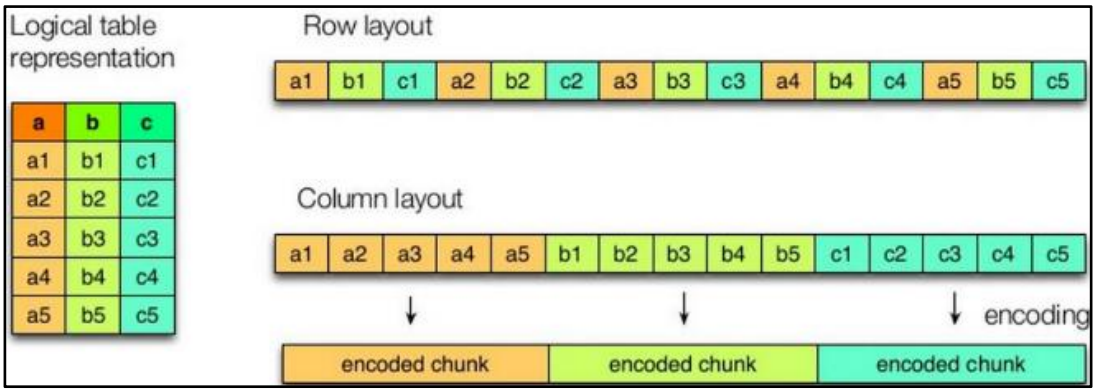
```
create table textfile_table
(column_specs)
stored as textfile;
```

9.1.2 ORC

1）文件格式

ORC（Optimized Row Columnar）file format 是 Hive 0.11 版里引入的一种**列式存储**的文件格式。ORC 文件能够提高 Hive 读写数据和处理数据的性能。

与列式存储相对的是行式存储，下图是两者的对比：



如图所示左边为逻辑表，右边第一个为行式存储，第二个为列式存储。

（1）行存储的特点

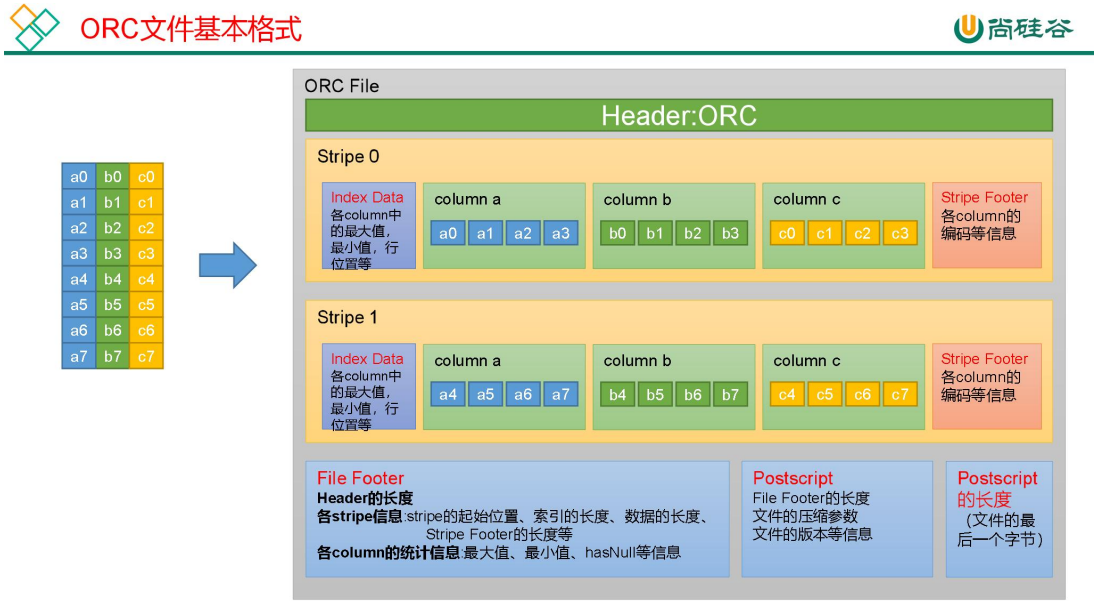
查询满足条件的一整行数据的时候，列存储则需要去每个聚集的字段找到对应的每个列的值，行存储只需要找到其中一个值，其余的值都在相邻地方，所以此时行存储查询的速度更快。

（2）列存储的特点

因为每个字段的数据聚集存储，在查询只需要少数几个字段的时候，能大大减少读取的数据量；每个字段的数据类型一定是相同的，列式存储可以针对性的设计更好的设计压缩算法。

前文提到的 text file 和 sequence file 都是基于行存储的，orc 和 parquet 是基于列式存储的。

orc 文件的具体结构如下图所示：



每个 Orc 文件由 Header、Body 和 Tail 三部分组成。

其中 Header 内容为 ORC，用于表示文件类型。

Body 由 1 个或多个 stripe 组成，每个 stripe 一般为 HDFS 的块大小，每一个 stripe 包含多条记录，这些记录按照列进行独立存储，每个 stripe 里有三部分组成，分别是 Index Data，Row Data，Stripe Footer。

Index Data： 一个轻量级的 index，默认是为各列每隔 1W 行做一个索引。每个索引会记录第 n 万行的位置，和最近一万行的最大值和最小值等信息。

Row Data： 存的是具体的数据，按列进行存储，并对每个列进行编码，分成多个 Stream 来存储。

Stripe Footer： 存放的是各个 Stream 的位置以及各 column 的编码信息。

Tail 由 File Footer 和 PostScript 组成。File Footer 中保存了各 Stripe 的其实位置、索引长度、数据长度等信息，各 Column 的统计信息等；PostScript 记录了整个文件的压缩类型以及 File Footer 的长度信息等。

在读取 ORC 文件时，会先从最后一个字节读取 PostScript 长度，进而读取到 PostScript，从里面解析到 File Footer 长度，进而读取 FileFooter，从中解析到各个 Stripe 信息，再读各个 Stripe，即从后往前读。

3）建表语句

```
create table orc_table
(column_specs)
stored as orc
tblproperties (property_name=property_value, ...);
```

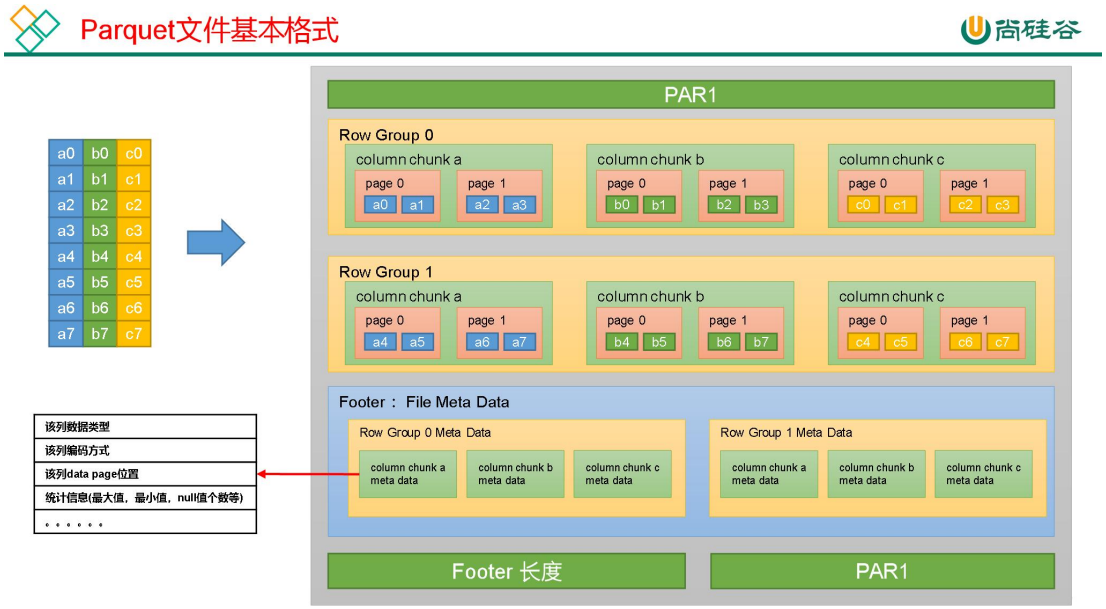
ORC 文件格式支持的参数如下：

参数	默认值	说明
orc.compress	ZLIB	压缩格式，可选项：NONE、ZLIB,、SNAPPY
orc.compress.size	262,144	每个压缩块的大小（ORC 文件是分块压缩的）
orc.stripe.size	67,108,864	每个 stripe 的大小
orc.row.index.stride	10,000	索引步长（每隔多少行数据建一条索引）

9.1.3 Parquet

Parquet 文件是 Hadoop 生态中的一个通用的文件格式，它也是一个列式存储的文件格式。

Parquet 文件的格式如下图所示：



上图展示了一个 Parquet 文件的基本结构，文件的首尾都是该文件的 Magic Code，用于校验它是否是一个 Parquet 文件。

首尾中间由若干个 Row Group 和一个 Footer（File Meta Data）组成。

每个 Row Group 包含多个 Column Chunk，每个 Column Chunk 包含多个 Page。以下是 Row Group、Column Chunk 和 Page 三个概念的说明：

行组（Row Group）：一个行组对应逻辑表中的若干行。

列块（Column Chunk）：一个行组中的一列保存在一个列块中。

页（Page）：一个列块的数据会划分为若干个页。

Footer（File Meta Data）中存储了每个行组（Row Group）中的每个列快（Column Chunk）的元数据信息，元数据信息包含了该列的数据类型、该列的编码方式、该类的 Data Page 位置等信息。

3）建表语句

```
Create table parquet_table
(column_specs)
stored as parquet
tblproperties (property_name=property_value, ...);
```

支持的参数如下：

参数	默认值	说明
parquet.compression	**uncompressed**	**压缩格式，可选项：uncompressed，snappy，gzip，lzo，lz4**
parquet.block.size	**134217728**	**行组大小，通常与 HDFS 块大小保持一致**

parquet.page.size	**1048576**	页大小

9.1.4 企业中选择

在数据分析的场景下，使用列式存储格式（orc 和 parquet）的查询性能和存储效率都要优于默认的文本文件格式（textfile），其中 orc 的性能略微优于 parquet。故优先选用 orc 格式。

参考资料：

- 以下是一个使用 TPC-DS 数据集（TPC-DS 是大数据领域最为知名的 Benchmark 标准），对 textfile、orc、parquet 三种文件格式的压力测试报告，可作为参考 https://codeantenna.com/a/QNO99FqYme
- 以下是 Hortonworks 公司开发的业界最常用的 TPC-DS 测试工具：https://github.com/hortonworks/hive-testbench

9.2 压缩

在 Hive 表中和计算过程中，保持数据的压缩，对磁盘空间的有效利用和提高查询性能都是十分有益的。

9.2.1 Hadoop 压缩概述

压缩格式	算法	文件扩展名	是否可切分
DEFLATE	DEFLATE	.deflate	否
Gzip	DEFLATE	.gz	否
bzip2	bzip2	.bz2	**是**
LZO	LZO	.lzo	**是**
Snappy	Snappy	.snappy	否

为了支持多种压缩/解压缩算法，Hadoop 引入了编码/解码器，如下表所示：

Hadoop 查看支持压缩的方式 hadoop checknative。

Hadoop 在 driver 端设置压缩。

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较：

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度

gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

http://google.github.io/snappy/

On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

9.2.2 Hive 表数据进行压缩

在 Hive 中，不同文件类型的表，声明数据压缩的方式是不同的。

1）TextFile

若一张表的文件类型为 TextFile，若需要对该表中的数据进行压缩，多数情况下，无需在建表语句做出声明。直接将压缩后的文件导入到该表即可，Hive 在查询表中数据时，可自动识别其压缩格式，进行解压。

需要注意的是，在执行往表中导入数据的 SQL 语句时，用户需设置以下参数，来保证写入表中的数据是被压缩的。

```
--SQL 语句的最终输出结果是否压缩
set hive.exec.compress.output=true;
--输出结果的压缩格式（以下示例为 snappy）
set mapreduce.output.fileoutputformat.compress.codec
=org.apache.hadoop.io.compress.SnappyCodec;
```

2）ORC

若一张表的文件类型为 ORC，若需要对该表数据进行压缩，需在建表语句中声明压缩格式如下：

```
create table orc_table
(column_specs)
stored as orc
tblproperties ("orc.compress"="snappy");
```

3）Parquet

若一张表的文件类型为 Parquet，若需要对该表数据进行压缩，需在建表语句中声明压缩格式如下：

```
create table orc_table
(column_specs)
stored as parquet
tblproperties ("parquet.compression"="snappy");
```

9.2.3 计算过程中使用压缩

1）单个 MR 的中间结果进行压缩

单个 MR 的中间结果是指 Mapper 输出的数据，对其进行压缩可降低 shuffle 阶段的网络 IO，可通过以下参数进行配置：

```
--开启 MapReduce 中间数据压缩功能
set mapreduce.map.output.compress=true;
--设置 MapReduce 中间数据数据的压缩方式（以下示例为 snappy）
set
mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.
SnappyCodec;
```

2）单条 SQL 语句的中间结果进行压缩

单条 SQL 语句的中间结果是指，两个 MR（一条 SQL 语句可能需要通过 MR 进行计算）之间的临时数据，可通过以下参数进行配置：

```
--是否对两个 MR 之间的临时数据进行压缩
set hive.exec.compress.intermediate=true;
--压缩格式（以下示例为 snappy）
set hive.intermediate.compression.codec=
org.apache.hadoop.io.compress.SnappyCodec;
```

第 10 章 企业级调优

10.1 计算资源配置

本教程的计算环境为 Hive on MR。计算资源的调整主要包括 Yarn 和 MR。

10.1.1 Yarn 资源配置

1）Yarn 配置说明

需要调整的 Yarn 参数均与 CPU、内存等资源有关，核心配置参数如下

（1）yarn.nodemanager.resource.memory-mb

该参数的含义是，一个 NodeManager 节点分配给 Container 使用的内存。该参数的配置，取决于 NodeManager 所在节点的总内存容量和该节点运行的其他服务的数量。

考虑上述因素，此处可将该参数设置为 64G，如下：

```
<property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>65536</value>
</property>
```

（2）yarn.nodemanager.resource.cpu-vcores

该参数的含义是，一个 NodeManager 节点分配给 Container 使用的 CPU 核数。该参数的配置，同样取决于 NodeManager 所在节点的总 CPU 核数和该节点运行的其他服务。

考虑上述因素，此处可将该参数设置为 16。

```
<property>
```

```
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>16</value>
</property>
```

（3）yarn.scheduler.maximum-allocation-mb

该参数的含义是，单个 Container 能够使用的最大内存。推荐配置如下：

```
<property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>16384</value>
</property>
```

（4）yarn.scheduler.minimum-allocation-mb

该参数的含义是，单个 Container 能够使用的最小内存，推荐配置如下：

```
<property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>512</value>
</property>
```

2）Yarn 配置实操

（1）修改$HADOOP_HOME/etc/hadoop/yarn-site.xml 文件

（2）修改如下参数

```
<property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>65536</value>
</property>
<property>
    <name>yarn.nodemanager.resource.cpu-vcores</name>
    <value>16</value>
</property>
<property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>16384</value>
</property>
<property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>512</value>
</property>
```

（3）分发该配置文件

（4）重启 Yarn。

10.1.2 MapReduce 资源配置

MapReduce 资源配置主要包括 Map Task 的内存和 CPU 核数，以及 Reduce Task 的内存和 CPU 核数。核心配置参数如下：

1）mapreduce.map.memory.mb

该参数的含义是，单个 Map Task 申请的 container 容器内存大小，其默认值为 1024。

该值不能超出 yarn.scheduler.maximum-allocation-mb 和 yarn.scheduler.minimum-allocation-mb 规定的范围。

该参数需要根据不同的计算任务单独进行配置，在 hive 中，可直接使用如下方式为每个 SQL 语句单独进行配置：

```
set  mapreduce.map.memory.mb=2048;
```

2）mapreduce.map.cpu.vcores

该参数的含义是，单个 Map Task 申请的 container 容器 cpu 核数，其默认值为 1。该值一般无需调整。

3）mapreduce.reduce.memory.mb

该参数的含义是，单个 Reduce Task 申请的 container 容器内存大小，其默认值为 1024。该值同样不能超出 yarn.scheduler.maximum-allocation-mb 和 yarn.scheduler.minimum-allocation-mb 规定的范围。

该参数需要根据不同的计算任务单独进行配置，在 hive 中，可直接使用如下方式为每个 SQL 语句单独进行配置：

```
set  mapreduce.reduce.memory.mb=2048;
```

4）mapreduce.reduce.cpu.vcores

该参数的含义是，单个 Reduce Task 申请的 container 容器 cpu 核数，其默认值为 1。该值一般无需调整。

10.2 测试用表

10.2.1 订单表(2000w 条数据)

1）表结构

id （订单 id）	user_id （用户 id）	product_id （商品 id）	province_id （省份 id）	create_time （下单时间）	product_num （商品 id）	total_amount （订单金额）
10000001	125442354	15003199	1	2020-06-14 03:54:29	3	100.58
10000002	192758405	17210367	1	2020-06-14 01:19:47	8	677.18

2）建表语句

```
hive (default)>
drop table if exists order_detail;
create table order_detail(
    id           string comment '订单 id',
    user_id      string comment '用户 id',
    product_id   string comment '商品 id',
```

```
    province_id  string comment '省份 id',
    create_time  string comment '下单时间',
    product_num  int comment '商品件数',
    total_amount decimal(16, 2) comment '下单金额'
)
partitioned by (dt string)
row format delimited fields terminated by '\t';
```

3）数据装载

将 order_detail.txt 文件上传到 hadoop102 节点的/opt/module/hive/datas/目录，并执行以下导入语句。

注：文件较大，请耐心等待。

```
hive (default)>
load data local inpath '/opt/module/hive/datas/order_detail.txt'
overwrite into table order_detail partition(dt='2020-06-14');
```

10.2.2 支付表(600w 条数据)

1）表结构

id （支付 id）	order_detail_id （订单 id）	user_id （用户 id）	payment_time （支付时间）	total_amount （订单金额）
10000001	17403042	131508758	2020-06-14 13:55:44	391.72
10000002	19198884	133018075	2020-06-14 08:46:23	657.10

2）建表语句

```
hive (default)>
drop table if exists payment_detail;
create table payment_detail(
    id              string comment '支付 id',
    order_detail_id string comment '订单明细 id',
    user_id         string comment '用户 id',
    payment_time    string comment '支付时间',
    total_amount    decimal(16, 2) comment '支付金额'
)
partitioned by (dt string)
row format delimited fields terminated by '\t';
```

3）数据装载

将 payment_detail.txt 文件上传到 hadoop102 节点的/opt/module/hive/datas/目录，并执行以下导入语句。

注：文件较大，请耐心等待。

```
hive (default)>
load data local inpath
'/opt/module/hive/datas/payment_detail.txt' overwrite into table
payment_detail partition(dt='2020-06-14');
```

10.2.3 商品信息表（100w 条数据）

1）表结构

id （商品 id）	product_name （商品名称）	price （价格）	category_id （分类 id）
1000001	**CuisW**	**4517.00**	**219**
1000002	**TBtbp**	**9357.00**	**208**

2）建表语句

```
hive (default)>
drop table if exists product_info;
create table product_info(
    id           string comment '商品 id',
    product_name string comment '商品名称',
    price        decimal(16, 2) comment '价格',
    category_id  string comment '分类 id'
)
row format delimited fields terminated by '\t';
```

3）数据装载

将 product_info.txt 文件上传到 hadoop102 节点的/opt/module/hive/datas/目录，并执行以下导入语句。

```
hive (default)>
load data local inpath '/opt/module/hive/datas/product_info.txt'
overwrite into table product_info;
```

10.2.4 省份信息表（34 条数据）

1）表结构

id（省份 id）	province_name（省份名称）
1	**北京**
2	**天津**

2）建表语句

```
hive (default)>
drop table if exists province_info;
create table province_info(
    id            string comment '省份 id',
    province_name string comment '省份名称'
)
row format delimited fields terminated by '\t';
```

3）数据装载

将 province_info.txt 文件上传到 hadoop102 节点的/opt/module/hive/datas/目录，并执行以下导入语句。

```
hive (default)>
load data local inpath '/opt/module/hive/datas/province_info.txt'
overwrite into table province_info;
```

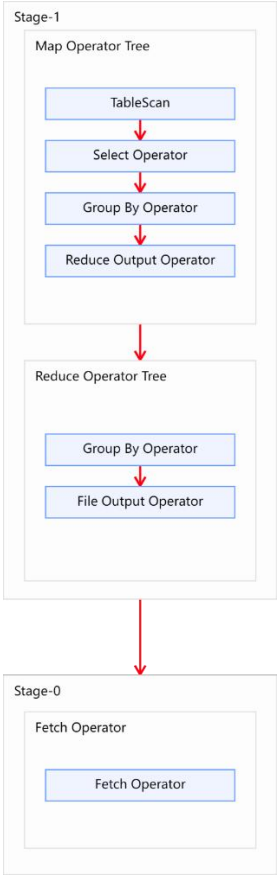
10.3 Explain 查看执行计划（重点）

10.3.1 Explain 执行计划概述

Explain 呈现的执行计划，由一系列 Stage 组成，这一系列 Stage 具有依赖关系，每个 Stage 对应一个 MapReduce Job，或者一个文件系统操作等。

若某个 Stage 对应的一个 MapReduce Job，其 Map 端和 Reduce 端的计算逻辑分别由 Map Operator Tree 和 Reduce Operator Tree 进行描述，Operator Tree 由一系列的 Operator 组成，一个 Operator 代表在 Map 或 Reduce 阶段的一个单一的逻辑操作，例如 TableScan Operator，Select Operator，Join Operator 等。

下图是由一个执行计划绘制而成：



常见的 Operator 及其作用如下：

- TableScan：表扫描操作，通常 map 端第一个操作肯定是表扫描操作
- Select Operator：选取操作
- Group By Operator：分组聚合操作
- Reduce Output Operator：输出到 reduce 操作

- Filter Operator：过滤操作
- Join Operator：join 操作
- File Output Operator：文件输出操作
- Fetch Operator 客户端获取数据操作

10.3.2 基本语法

```
EXPLAIN [FORMATTED | EXTENDED | DEPENDENCY] query-sql
```

注：FORMATTED、EXTENDED、DEPENDENCY 关键字为可选项，各自作用如下。

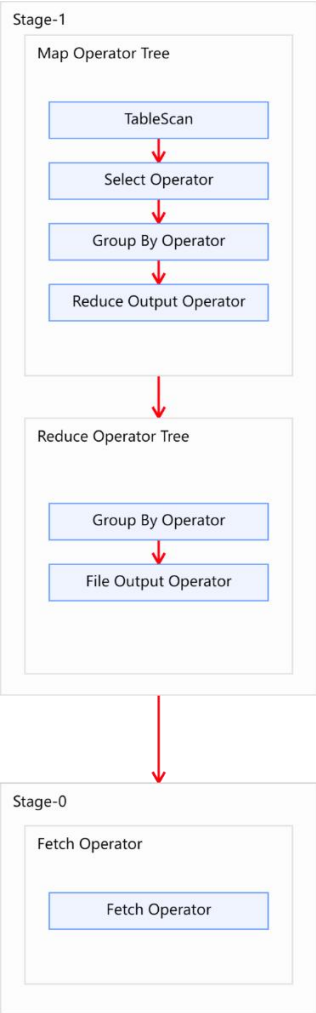
- FORMATTED：将执行计划以 JSON 字符串的形式输出
- EXTENDED：输出执行计划中的额外信息，通常是读写的文件名等信息
- DEPENDENCY：输出执行计划读取的表及分区

10.3.3 案例实操

1）查看下面这条语句的执行计划

```
hive (default)>
explain
select
    user_id,
    count(*)
from order_detail
group by user_id;
```

2）执行计划如下图



10.4 HQL 语法优化之分组聚合优化

10.4.1 优化说明

Hive 中未经优化的分组聚合，是通过一个 MapReduce Job 实现的。Map 端负责读取数据，并按照分组字段分区，通过 Shuffle，将数据发往 Reduce 端，各组数据在 Reduce 端完成最终的聚合运算。

Hive 对分组聚合的优化主要围绕着减少 Shuffle 数据量进行，具体做法是 map-side 聚合。所谓 map-side 聚合，就是在 map 端维护一个 hash table，利用其完成部分的聚合，然后将部分聚合的结果，按照分组字段分区，发送至 reduce 端，完成最终的聚合。map-side 聚合能有效减少 shuffle 的数据量，提高分组聚合运算的效率。

map-side 聚合相关的参数如下：

```
--启用 map-side 聚合
set hive.map.aggr=true;

--用于检测源表数据是否适合进行 map-side 聚合。检测的方法是：先对若干条数据进行
```

```
map-side 聚合，若聚合后的条数和聚合前的条数比值小于该值，则认为该表适合进行map-side 聚合；否则，认为该表数据不适合进行 map-side 聚合，后续数据便不再进行map-side 聚合。
set hive.map.aggr.hash.min.reduction=0.5;

--用于检测源表是否适合 map-side 聚合的条数。
set hive.groupby.mapaggr.checkinterval=100000;

--map-side 聚合所用的 hash table，占用 map task 堆内存的最大比例，若超出该值，则会对 hash table 进行一次 flush。
set hive.map.aggr.hash.force.flush.memory.threshold=0.9;
```

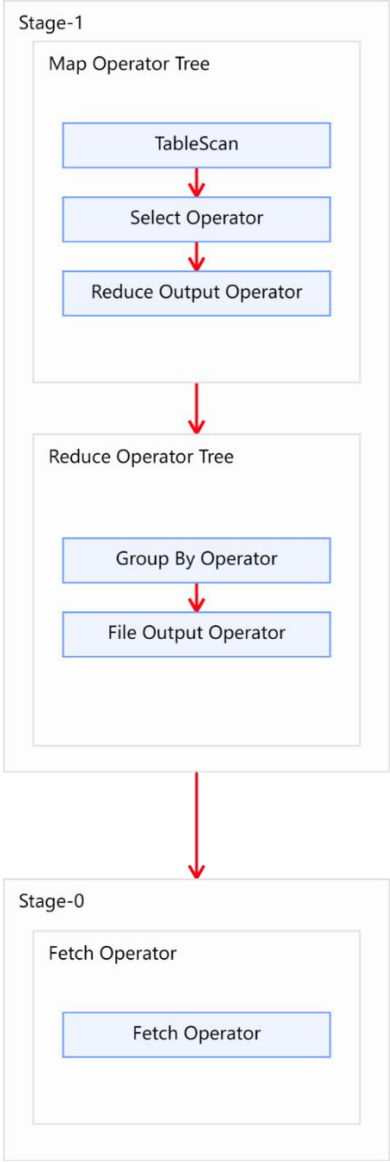
10.4.2 优化案例

1）示例 SQL

```
hive (default)>
select
    product_id,
    count(*)
from order_detail
group by product_id;
```

2）优化前

未经优化的分组聚合，执行计划如下图所示：



3）优化思路

可以考虑开启 map-side 聚合，配置以下参数：

```
--启用 map-side 聚合，默认是 true
set hive.map.aggr=true;

--用于检测源表数据是否适合进行 map-side 聚合。检测的方法是：先对若干条数据进行 map-side 聚合，若聚合后的条数和聚合前的条数比值小于该值，则认为该表适合进行 map-side 聚合；否则，认为该表数据不适合进行 map-side 聚合，后续数据便不再进行 map-side 聚合。
set hive.map.aggr.hash.min.reduction=0.5;

--用于检测源表是否适合 map-side 聚合的条数。
set hive.groupby.mapaggr.checkinterval=100000;

--map-side 聚合所用的 hash table，占用 map task 堆内存的最大比例，若超出该值，则会对 hash table 进行一次 flush。
```

```
set hive.map.aggr.hash.force.flush.memory.threshold=0.9;
```

优化后的执行计划如图所示：

10.5 HQL 语法优化之 Join 优化

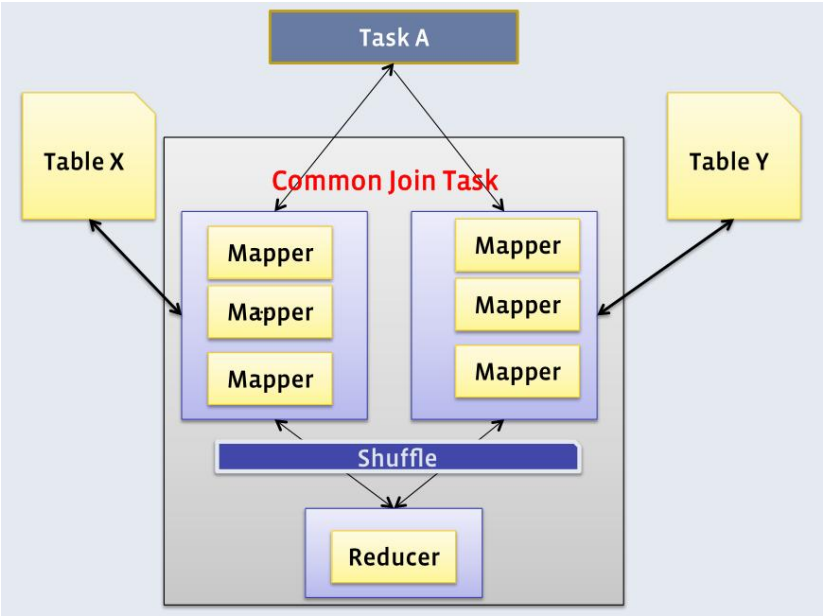
10.5.1 Join 算法概述

Hive 拥有多种 join 算法，包括 Common Join，Map Join，Bucket Map Join，Sort Merge Buckt Map Join 等，下面对每种 join 算法做简要说明：

1）Common Join

Common Join 是 Hive 中最稳定的 join 算法，其通过一个 MapReduce Job 完成一个 join 操作。Map 端负责读取 join 操作所需表的数据，并按照关联字段进行分区，通过

Shuffle，将其发送到 Reduce 端，相同 key 的数据在 Reduce 端完成最终的 Join 操作。如下图所示：



需要**注意**的是，sql 语句中的 join 操作和执行计划中的 Common Join 任务并非一对一的关系，一个 sql 语句中的**相邻**的且**关联字段相同**的多个 join 操作可以合并为一个 Common Join 任务。

例如：

```
hive (default)>
select
    a.val,
    b.val,
    c.val
from a
join b on (a.key = b.key1)
join c on (c.key = b.key1)
```

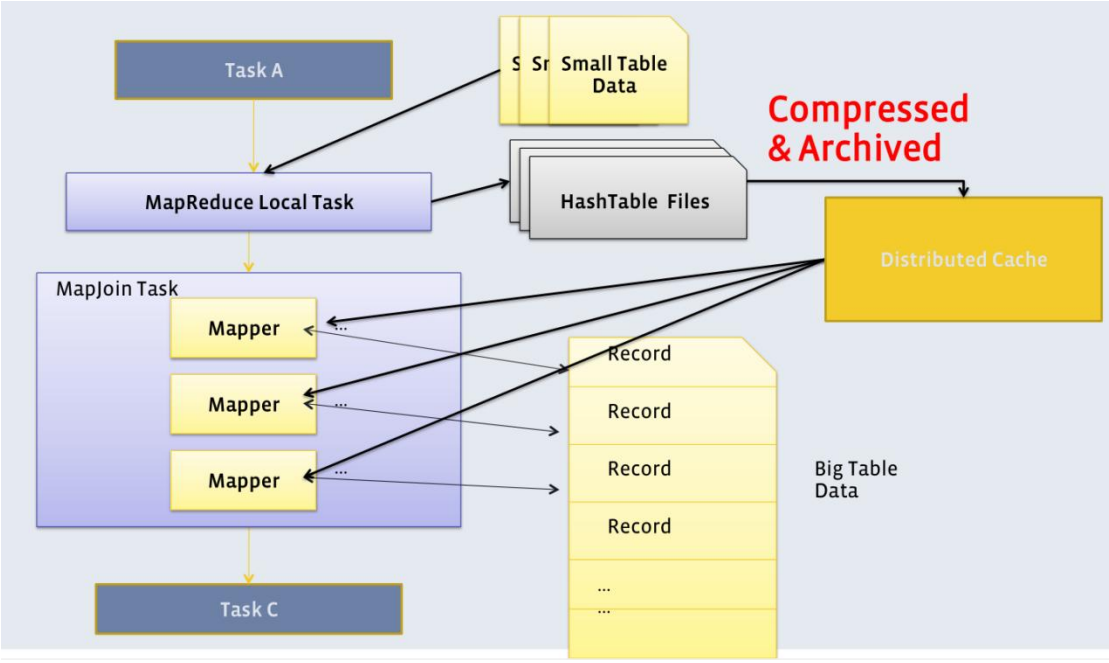
上述 sql 语句中两个 join 操作的关联字段均为 b 表的 key1 字段，则该语句中的两个 join 操作可由一个 Common Join 任务实现，也就是可通过一个 Map Reduce 任务实现。

```
hive (default)>
select
    a.val,
    b.val,
    c.val
from a
join b on (a.key = b.key1)
join c on (c.key = b.key2)
```

上述 sql 语句中的两个 join 操作关联字段各不相同，则该语句的两个 join 操作需要各自通过一个 Common Join 任务实现，也就是通过两个 Map Reduce 任务实现。

2）Map Join

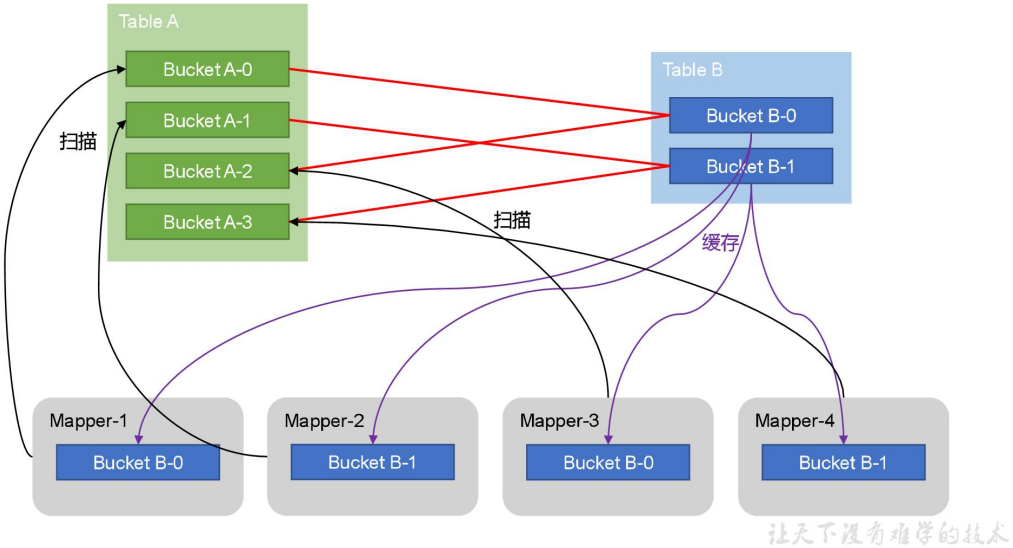
Map Join 算法可以通过两个只有 map 阶段的 Job 完成一个 join 操作。其适用场景为大表 join 小表。若某 join 操作满足要求，则第一个 Job 会读取小表数据，将其制作为 hash table，并上传至 Hadoop 分布式缓存（本质上是上传至 HDFS）。第二个 Job 会先从分布式缓存中读取小表数据，并缓存在 Map Task 的内存中，然后扫描大表数据，这样在 map 端即可完成关联操作。如下图所示：



3）Bucket Map Join

Bucket Map Join 是对 Map Join 算法的改进，其打破了 Map Join 只适用于大表 join 小表的限制，可用于大表 join 大表的场景。

Bucket Map Join 的核心思想是：若能保证参与 join 的表均为分桶表，且关联字段为分桶字段，且其中一张表的分桶数量是另外一张表分桶数量的整数倍，就能保证参与 join 的两张表的分桶之间具有明确的关联关系，所以就可以在两表的分桶间进行 Map Join 操作了。这样一来，第二个 Job 的 Map 端就无需再缓存小表的全表数据了，而只需缓存其所需的分桶即可。其原理如图所示：

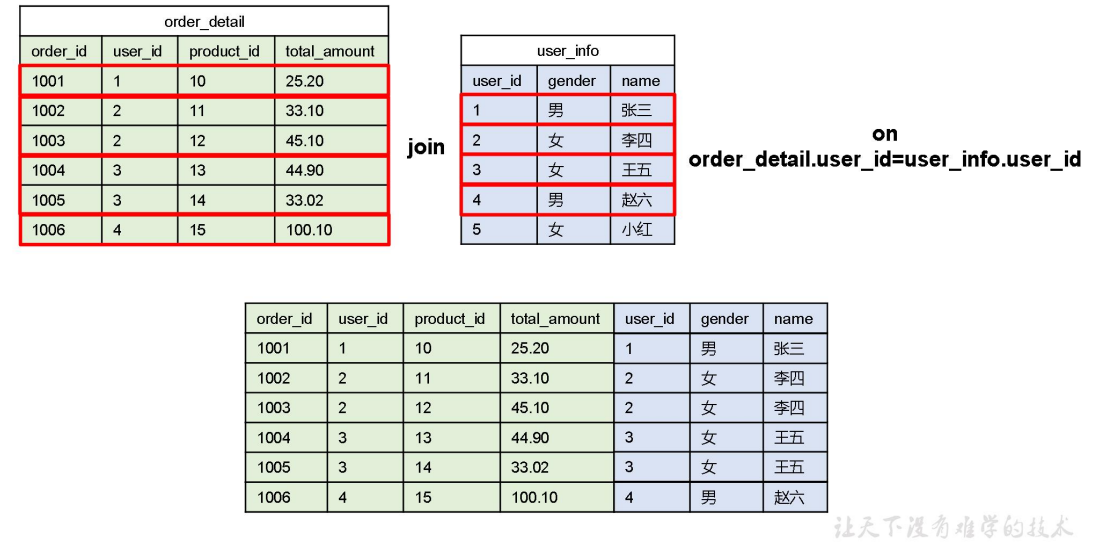


4）Sort Merge Bucket Map Join

Sort Merge Bucket Map Join（简称 SMB Map Join）基于 Bucket Map Join。SMB Map Join 要求，参与 join 的表均为分桶表，且需保证分桶内的数据是有序的，且分桶字段、排序字段和关联字段为相同字段，且其中一张表的分桶数量是另外一张表分桶数量的整数倍。

SMB Map Join 同 Bucket Join 一样，同样是利用两表各分桶之间的关联关系，在分桶之间进行 join 操作，不同的是，分桶之间的 join 操作的实现原理。Bucket Map Join，两个分桶之间的 join 实现原理为 Hash Join 算法；而 SMB Map Join，两个分桶之间的 join 实现原理为 Sort Merge Join 算法。

Hash Join 和 Sort Merge Join 均为关系型数据库中常见的 Join 实现算法。Hash Join 的原理相对简单，就是对参与 join 的一张表构建 hash table，然后扫描另外一张表，然后进行逐行匹配。Sort Merge Join 需要在两张按照关联字段排好序的表中进行，其原理如图所示：



Hive 中的 SMB Map Join 就是对两个分桶的数据按照上述思路进行 Join 操作。可以看出，SMB Map Join 与 Bucket Map Join 相比，在进行 Join 操作时，Map 端是无需对整个 Bucket 构建 hash table，也无需在 Map 端缓存整个 Bucket 数据的，每个 Mapper 只需按顺序逐个 key 读取两个分桶的数据进行 join 即可。

10.5.2 Map Join

10.5.2.1 优化说明

Map Join 有两种触发方式，一种是用户在 SQL 语句中增加 hint 提示，另外一种是 Hive 优化器根据参与 join 表的数据量大小，自动触发。

1）Hint 提示

用户可通过如下方式，指定通过 map join 算法，并且 ta 将作为 map join 中的小表。这种方式已经过时，不推荐使用。

```
hive (default)>
select /*+ mapjoin(ta) */
    ta.id,
    tb.id
from table_a ta
join table_b tb
on ta.id=tb.id;
```

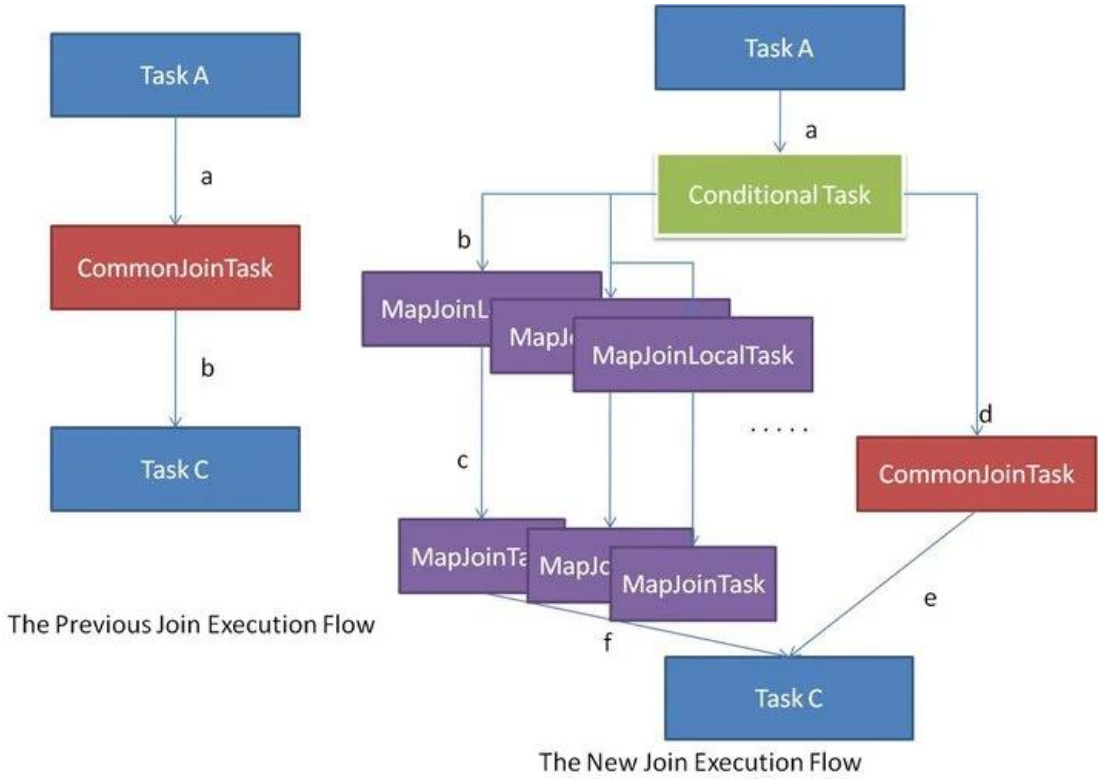
2）自动触发

Hive 在编译 SQL 语句阶段，起初所有的 join 操作均采用 Common Join 算法实现。

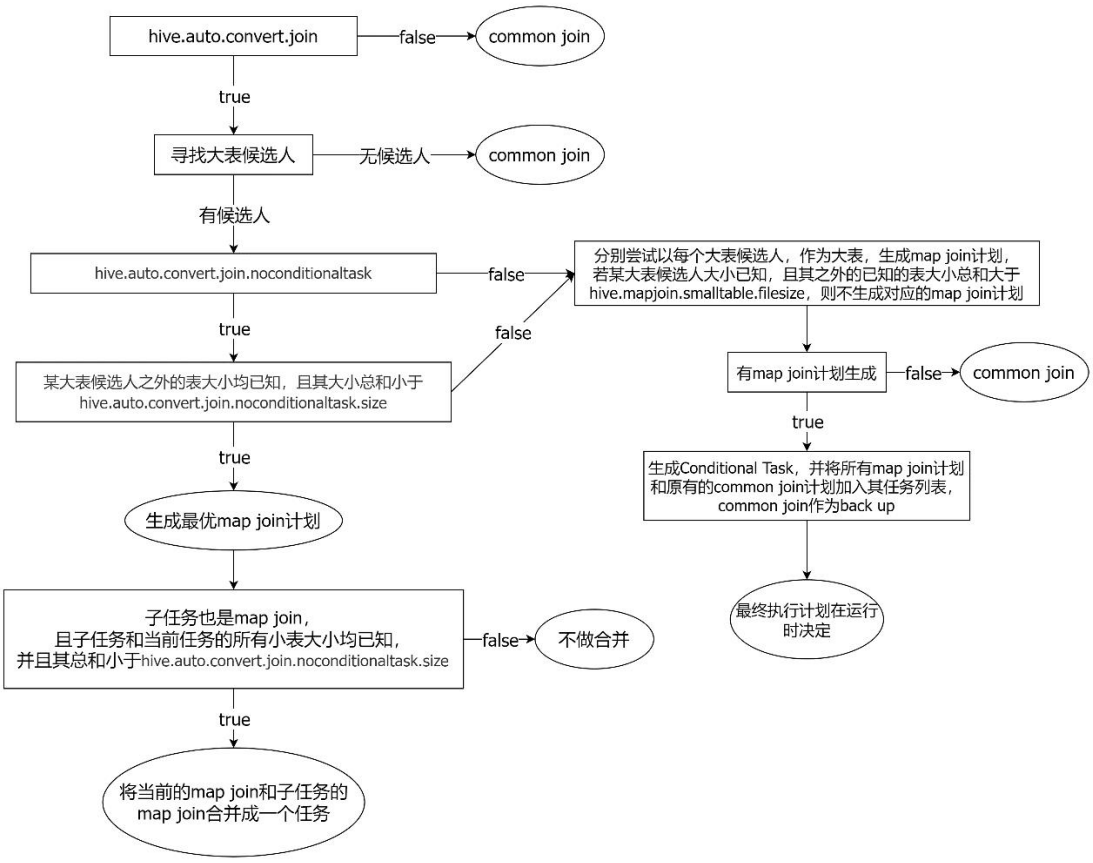
之后在物理优化阶段，Hive 会根据每个 Common Join 任务所需表的大小判断该 Common Join 任务是否能够转换为 Map Join 任务，若满足要求，便将 Common Join 任务自动转换为 Map Join 任务。

但有些 Common Join 任务所需的表大小，在 SQL 的编译阶段是未知的（例如对子查询进行 join 操作），所以这种 Common Join 任务是否能转换成 Map Join 任务在编译阶是无法确定的。

针对这种情况，Hive 会在编译阶段生成一个条件任务（Conditional Task），其下会包含一个计划列表，计划列表中包含转换后的 Map Join 任务以及原有的 Common Join 任务。最终具体采用哪个计划，是在运行时决定的。大致思路如下图所示：



Map join 自动转换的具体判断逻辑如下图所示：



图中涉及到的参数如下：

```
--启动 Map Join 自动转换
set hive.auto.convert.join=true;

--一个 Common Join operator 转为 Map Join operator 的判断条件,若该 Common Join 相关的表中,存在 n-1 张表的已知大小总和<=该值,则生成一个 Map Join 计划,此时可能存在多种 n-1 张表的组合均满足该条件,则 hive 会为每种满足条件的组合均生成一个 Map Join 计划,同时还会保留原有的 Common Join 计划作为后备(back up)计划,实际运行时,优先执行 Map Join 计划，若不能执行成功，则启动 Common Join 后备计划。
set hive.mapjoin.smalltable.filesize=250000;

--开启无条件转 Map Join
set hive.auto.convert.join.noconditionaltask=true;

--无条件转 Map Join 时的小表之和阈值,若一个 Common Join operator 相关的表中，存在 n-1 张表的大小总和<=该值,此时 hive 便不会再为每种 n-1 张表的组合均生成 Map Join 计划,同时也不会保留 Common Join 作为后备计划。而是只生成一个最优的 Map Join 计划。
set hive.auto.convert.join.noconditionaltask.size=10000000;
```

10.5.2.2 优化案例

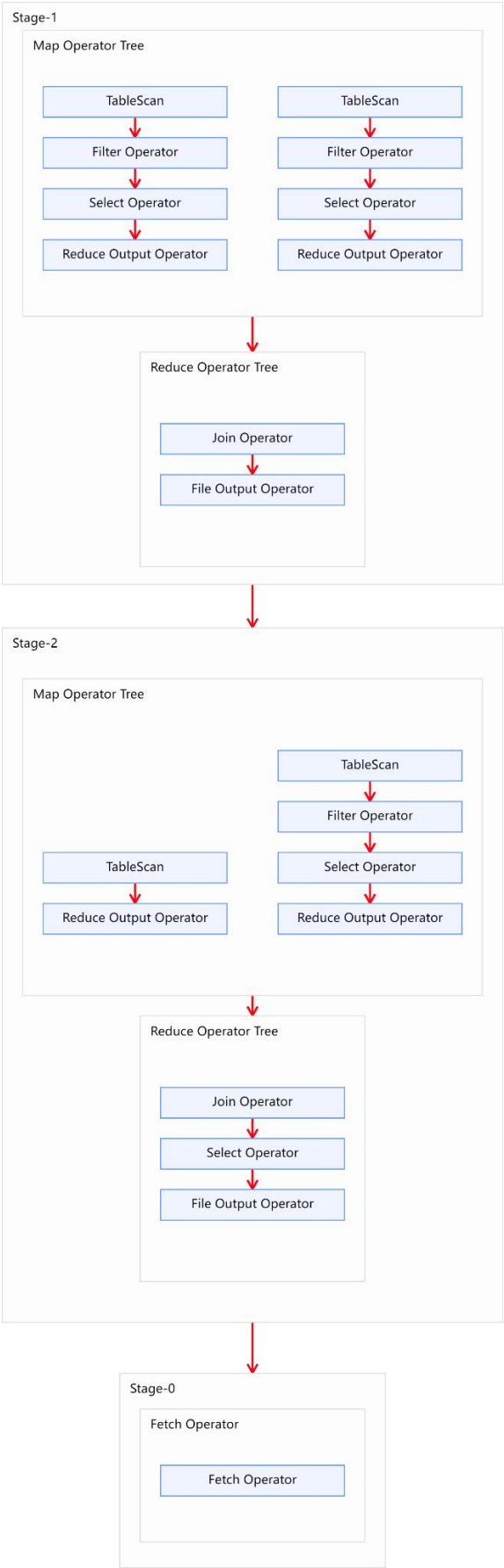
1）示例 SQL

```
hive (default)>
select
    *
```

```
from order_detail od
join product_info product on od.product_id = product.id
join province_info province on od.province_id = province.id;
```

2）优化前

上述 SQL 语句共有三张表进行两次 join 操作，且两次 join 操作的关联字段不同。故优化前的执行计划应该包含两个 Common Join operator，也就是由两个 MapReduce 任务实现。执行计划如下图所示：


Stage-1
Map Operator Tree
TableScan
Filter Operator
Select Operator
Reduce Output Operator
TableScan
Filter Operator
Select Operator
Reduce Output Operator
Reduce Operator Tree
Join Operator
File Output Operator
Stage-2
Map Operator Tree
TableScan
Reduce Output Operator
TableScan
Filter Operator
Select Operator
Reduce Output Operator
Reduce Operator Tree
Join Operator
Select Operator
File Output Operator
Stage-0
Fetch Operator
Fetch Operator

3）优化思路

经分析，参与 join 的三张表，数据量如下

表名	大小
order_detail	1176009934（约 1122M）
product_info	25285707（约 24M）
province_info	369（约 0.36K）

注：可使用如下语句获取表/分区的大小信息

```
hive (default)>
desc formatted table_name partition(partition_col='partition');
```

三张表中，product_info 和 province_info 数据量较小，可考虑将其作为小表，进行 Map Join 优化。

根据前文 Common Join 任务转 Map Join 任务的判断逻辑图，可得出以下优化方案：

方案一：

启用 Map Join 自动转换。

```
hive (default)>
set hive.auto.convert.join=true;
```

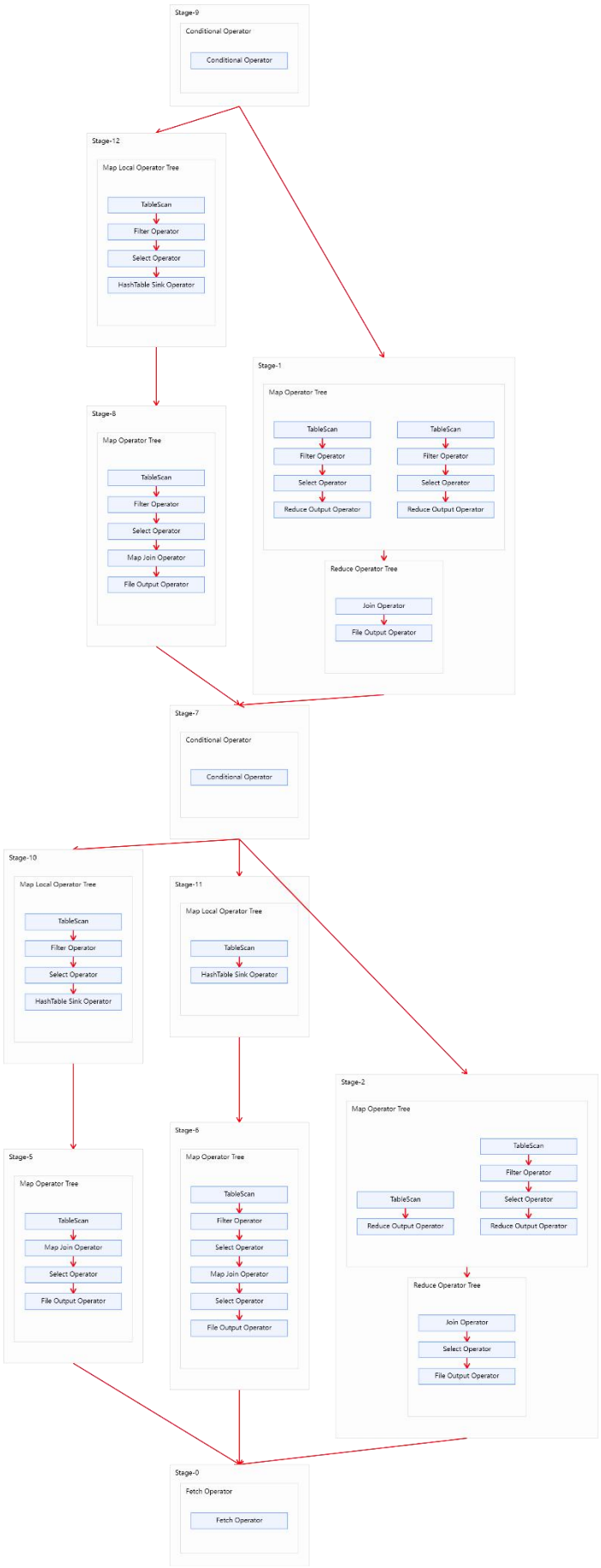
不使用无条件转 Map Join。

```
hive (default)>
set hive.auto.convert.join.noconditionaltask=false;
```

调整 hive.mapjoin.smalltable.filesize 参数，使其大于等于 product_info。

```
hive (default)>
set hive.mapjoin.smalltable.filesize=25285707;
```

这样可保证将两个 Common Join operator 均可转为 Map Join operator，并保留 Common Join 作为后备计划，保证计算任务的稳定。调整完的执行计划如下图：


Stage-9
Conditional Operator
Conditional Operator
Stage-12
Map Local Operator Tree
TableScan
Filter Operator
Select Operator
HashTable Sink Operator
Stage-1
Map Operator Tree
TableScan
Filter Operator
Select Operator
Reduce Output Operator
TableScan
Filter Operator
Select Operator
Reduce Output Operator
Reduce Operator Tree
Join Operator
File Output Operator
Stage-8
Map Operator Tree
TableScan
Filter Operator
Select Operator
Map Join Operator
File Output Operator
Stage-7
Conditional Operator
Conditional Operator
Stage-10
Map Local Operator Tree
TableScan
Filter Operator
Select Operator
HashTable Sink Operator
Stage-11
Map Local Operator Tree
TableScan
HashTable Sink Operator
Stage-2
Map Operator Tree
TableScan
Reduce Output Operator
TableScan
Filter Operator
Select Operator
Reduce Output Operator
Reduce Operator Tree
Join Operator
Select Operator
File Output Operator
Stage-6
Map Operator Tree
TableScan
Filter Operator
Select Operator
Map Join Operator
Select Operator
File Output Operator
Stage-5
Map Operator Tree
TableScan
Map Join Operator
Select Operator
File Output Operator
Stage-0
Fetch Operator
Fetch Operator

——

方案二：

启用 Map Join 自动转换。

```
hive (default)>
set hive.auto.convert.join=true;
```

使用无条件转 Map Join。

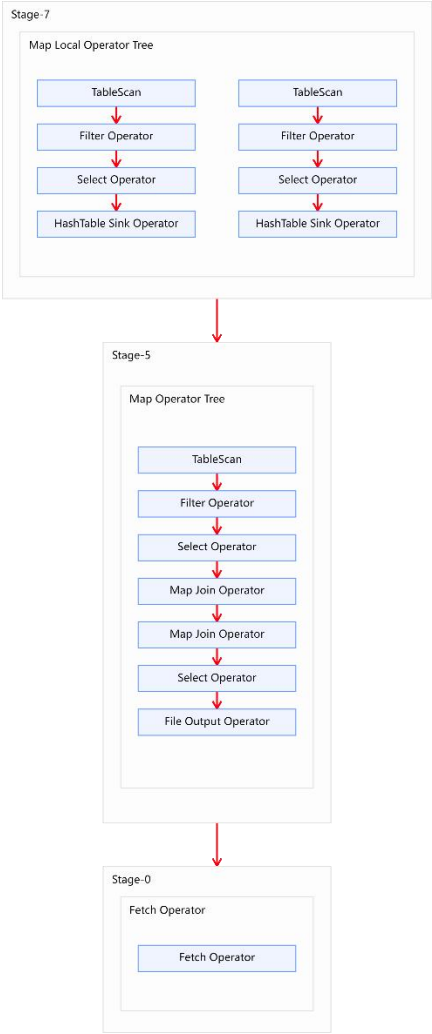
```
hive (default)>
set hive.auto.convert.join.noconditionaltask=true;
```

调整 hive.auto.convert.join.noconditionaltask.size 参数，使其大于等于 product_info 和 province_info 之和。

```
hive (default)>
set hive.auto.convert.join.noconditionaltask.size=25286076;
```

这样可直接将两个 Common Join operator 转为两个 Map Join operator，并且由于两个 Map Join operator 的小表大小之和小于等于 hive.auto.convert.join.noconditionaltask.size，故两个 Map Join operator 任务可合并为同一个。这个方案计算效率最高，但需要的内存也是最多的。

调整完的执行计划如下图：



方案三：

启用 Map Join 自动转换。

```
hive (default)>
set hive.auto.convert.join=true;
```

使用无条件转 Map Join。

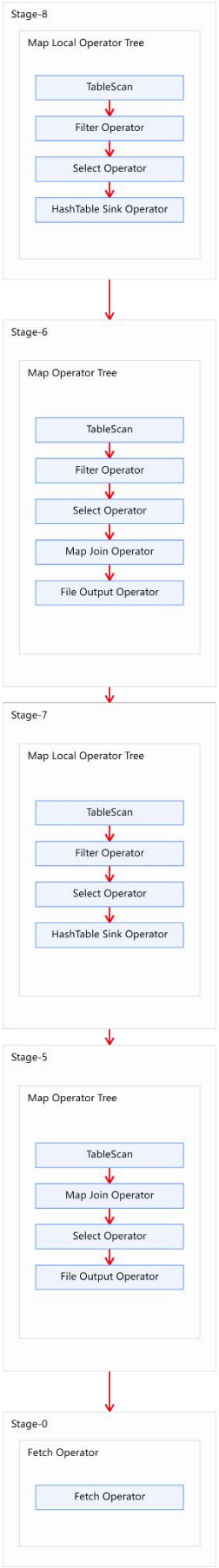
```
hive (default)>
set hive.auto.convert.join.noconditionaltask=true;
```

调整 hive.auto.convert.join.noconditionaltask.size 参数，使其等于 product_info。

```
hive (default)>
set hive.auto.convert.join.noconditionaltask.size=25285707;
```

这样可直接将两个 Common Join operator 转为 Map Join operator，但不会将两个 Map Join 的任务合并。该方案计算效率比方案二低，但需要的内存也更少。

调整完的执行计划如下图：


Stage-8
Map Local Operator Tree
TableScan
Filter Operator
Select Operator
HashTable Sink Operator
Stage-6
Map Operator Tree
TableScan
Filter Operator
Select Operator
Map Join Operator
File Output Operator
Stage-7
Map Local Operator Tree
TableScan
Filter Operator
Select Operator
HashTable Sink Operator
Stage-5
Map Operator Tree
TableScan
Map Join Operator
Select Operator
File Output Operator
Stage-0
Fetch Operator
Fetch Operator

10.5.3 Bucket Map Join

10.5.3.1 优化说明

Bucket Map Join 不支持自动转换，发须通过用户在 SQL 语句中提供如下 Hint 提示，并配置如下相关参数，方可使用。

1）Hint 提示

```
hive (default)>
select /*+ mapjoin(ta) */
    ta.id,
    tb.id
from table_a ta
join table_b tb on ta.id=tb.id;
```

2）相关参数

```
--关闭 cbo 优化，cbo 会导致 hint 信息被忽略
set hive.cbo.enable=false;
--map join hint 默认会被忽略(因为已经过时)，需将如下参数设置为 false
set hive.ignore.mapjoin.hint=false;
--启用 bucket map join 优化功能
set hive.optimize.bucketmapjoin = true;
```

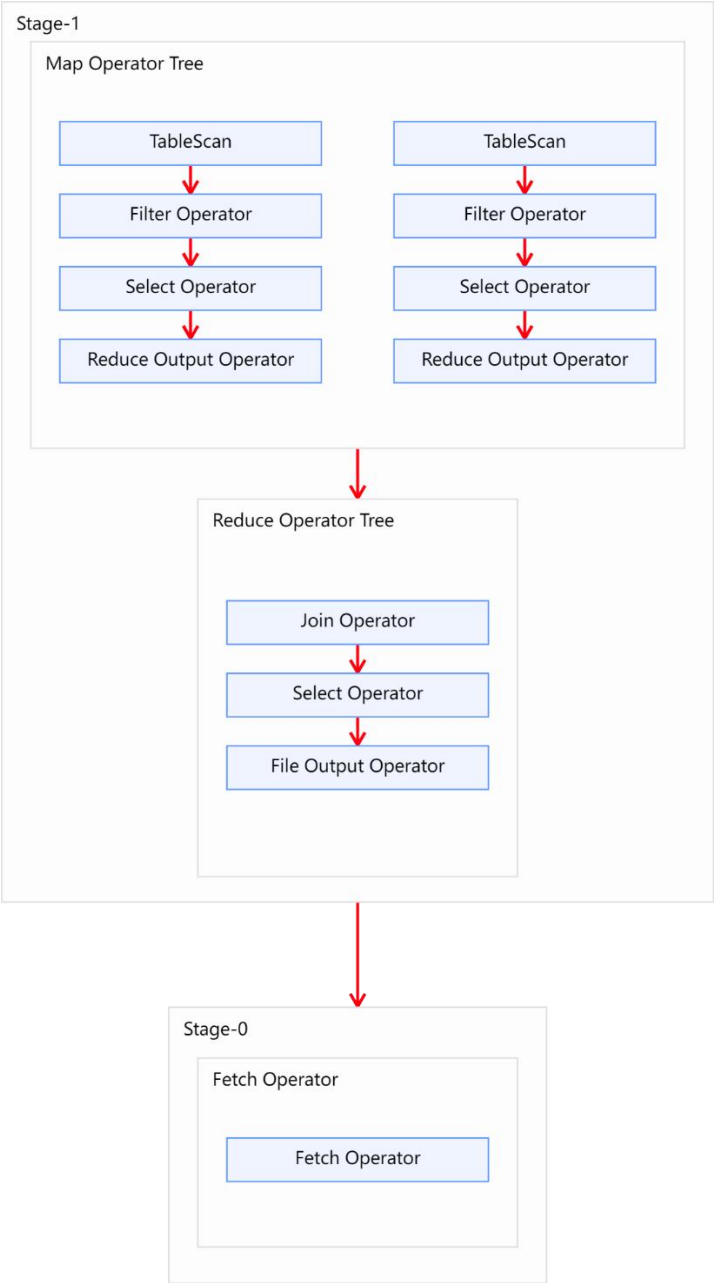
10.5.3.2 优化案例

1）示例 SQL

```
hive (default)>
select
    *
from(
    select
        *
    from order_detail
    where dt='2020-06-14'
)od
join(
    select
        *
    from payment_detail
    where dt='2020-06-14'
)pd
on od.id=pd.order_detail_id;
```

2）优化前

上述 SQL 语句共有两张表一次 join 操作，故优化前的执行计划应包含一个 Common Join 任务，通过一个 MapReduce Job 实现。执行计划如下图所示：



3）优化思路

经分析，参与 join 的两张表，数据量如下。

表名	大小
order_detail	1176009934（约 1122M）
payment_detail	334198480（约 319M）

两张表都相对较大，若采用普通的 Map Join 算法，则 Map 端需要较多的内存来缓存数据，当然可以选择为 Map 段分配更多的内存，来保证任务运行成功。但是，Map 端的内存不可能无上限的分配，所以当参与 Join 的表数据量均过大时，就可以考虑采用 Bucket Map Join 算法。下面演示如何使用 Bucket Map Join。

首先需要依据源表创建两个分桶表，order_detail 建议分 16 个 bucket，payment_detail 建议分 8 个 bucket,注意**分桶个数**的倍数关系以及**分桶字段**。

```
--订单表
hive (default)>
drop table if exists order_detail_bucketed;
create table order_detail_bucketed(
    id            string comment '订单 id',
    user_id       string comment '用户 id',
    product_id    string comment '商品 id',
    province_id   string comment '省份 id',
    create_time   string comment '下单时间',
    product_num   int comment '商品件数',
    total_amount decimal(16, 2) comment '下单金额'
)
clustered by (id) into 16 buckets
row format delimited fields terminated by '\t';

--支付表
hive (default)>
drop table if exists payment_detail_bucketed;
create table payment_detail_bucketed(
    id               string comment '支付 id',
    order_detail_id string comment '订单明细 id',
    user_id          string comment '用户 id',
    payment_time     string comment '支付时间',
    total_amount     decimal(16, 2) comment '支付金额'
)
clustered by (order_detail_id) into 8 buckets
row format delimited fields terminated by '\t';
```

然后向两个分桶表导入数据。

```
--订单表
hive (default)>
insert overwrite table order_detail_bucketed
select
    id,
    user_id,
    product_id,
    province_id,
    create_time,
    product_num,
    total_amount
from order_detail
where dt='2020-06-14';

--分桶表
hive (default)>
insert overwrite table payment_detail_bucketed
select
    id,
```

```
    order_detail_id,
    user_id,
    payment_time,
    total_amount
from payment_detail
where dt='2020-06-14';
```

然后设置以下参数：

```
--关闭 cbo 优化，cbo 会导致 hint 信息被忽略，需将如下参数修改为 false
set hive.cbo.enable=false;
--map join hint 默认会被忽略(因为已经过时)，需将如下参数修改为 false
set hive.ignore.mapjoin.hint=false;
--启用 bucket map join 优化功能,默认不启用，需将如下参数修改为 true
set hive.optimize.bucketmapjoin = true;
```

最后在重写 SQL 语句，如下：

```
hive (default)>
select /*+ mapjoin(pd) */
    *
from order_detail_bucketed od
join payment_detail_bucketed pd on od.id = pd.order_detail_id;
```

优化后的执行计划如图所示：

需要注意的是，Bucket Map Join 的执行计划的基本信息和普通的 Map Join 无异，若想看到差异，可执行如下语句，查看执行计划的详细信息。详细执行计划中，如在 Map Join Operator 中看到 “**BucketMapJoin: true**”，则表明使用的 Join 算法为 Bucket Map Join。

```
hive (default)>
explain extended select /*+ mapjoin(pd) */
    *
from order_detail_bucketed od
join payment_detail_bucketed pd on od.id = pd.order_detail_id;
```

10.5.4 Sort Merge Bucket Map Join

10.5.4.1 优化说明

Sort Merge Bucket Map Join 有两种触发方式，包括 Hint 提示和自动转换。Hint 提示已过时，不推荐使用。下面是自动转换的相关参数：

```
--启动 Sort Merge Bucket Map Join 优化
set hive.optimize.bucketmapjoin.sortedmerge=true;
--使用自动转换 SMB Join
set hive.auto.convert.sortmerge.join=true;
```

10.5.4.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    *
from(
    select
        *
    from order_detail
    where dt='2020-06-14'
)od
join(
    select
        *
    from payment_detail
    where dt='2020-06-14'
)pd
on od.id=pd.order_detail_id;
```

2）优化前

上述 SQL 语句共有两张表一次 join 操作，故优化前的执行计划应包含一个 Common Join 任务，通过一个 MapReduce Job 实现。

3）优化思路

经分析，参与 join 的两张表，数据量如下。

表名	大小
order_detail	1176009934（约 1122M）
payment_detail	334198480（约 319M）

两张表都相对较大，除了可以考虑采用 Bucket Map Join 算法，还可以考虑 SMB Join。相较于 Bucket Map Join，SMB Map Join 对分桶大小是没有要求的。下面演示如何使用 SMB Map Join。

首先需要依据源表创建两个的有序的分桶表，order_detail 建议分 16 个 bucket，payment_detail 建议分 8 个 bucket,注意**分桶个数**的倍数关系以及**分桶字段和排序字段**。

```
--订单表
hive (default)>
drop table if exists order_detail_sorted_bucketed;
create table order_detail_sorted_bucketed(
    id            string comment '订单 id',
    user_id       string comment '用户 id',
    product_id    string comment '商品 id',
    province_id   string comment '省份 id',
    create_time   string comment '下单时间',
    product_num   int comment '商品件数',
    total_amount decimal(16, 2) comment '下单金额'
)
clustered by (id) sorted by(id) into 16 buckets
row format delimited fields terminated by '\t';

--支付表
hive (default)>
drop table if exists payment_detail_sorted_bucketed;
create table payment_detail_sorted_bucketed(
    id               string comment '支付 id',
    order_detail_id string comment '订单明细 id',
    user_id          string comment '用户 id',
    payment_time     string comment '支付时间',
    total_amount     decimal(16, 2) comment '支付金额'
)
clustered by (order_detail_id) sorted by(order_detail_id) into 8
buckets
row format delimited fields terminated by '\t';
```

然后向两个分桶表导入数据。

```
--订单表
hive (default)>
insert overwrite table order_detail_sorted_bucketed
select
    id,
    user_id,
    product_id,
    province_id,
    create_time,
    product_num,
    total_amount
from order_detail
where dt='2020-06-14';

--分桶表
hive (default)>
insert overwrite table payment_detail_sorted_bucketed
select
```

```
    id,
    order_detail_id,
    user_id,
    payment_time,
    total_amount
from payment_detail
where dt='2020-06-14';
```

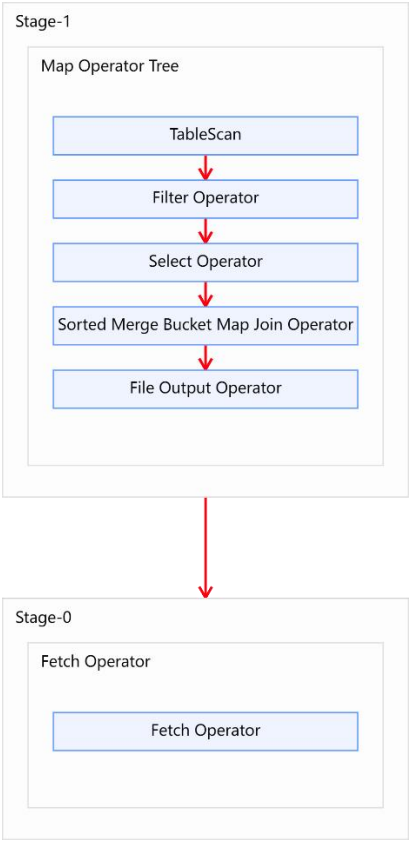
然后设置以下参数：

```
--启动 Sort Merge Bucket Map Join 优化
set hive.optimize.bucketmapjoin.sortedmerge=true;
--使用自动转换 SMB Join
set hive.auto.convert.sortmerge.join=true;
```

最后在重写 SQL 语句，如下：

```
hive (default)>
select
    *
from order_detail_sorted_bucketed od
join payment_detail_sorted_bucketed pd
on od.id = pd.order_detail_id;
```

优化后的执行计如图所示：



10.6 HQL 语法优化之数据倾斜

10.6.1 数据倾斜概述

数据倾斜问题，通常是指参与计算的数据分布不均，即某个 key 或者某些 key 的数据量远超其他 key，导致在 shuffle 阶段，大量相同 key 的数据被发往同一个 Reduce，进而导致该 Reduce 所需的时间远超其他 Reduce，成为整个任务的瓶颈。

Hive 中的数据倾斜常出现在分组聚合和 join 操作的场景中，下面分别介绍在上述两种场景下的优化思路。

10.6.2 分组聚合导致的数据倾斜

10.6.2.1 优化说明

前文提到过，Hive 中未经优化的分组聚合，是通过一个 MapReduce Job 实现的。Map 端负责读取数据，并按照分组字段分区，通过 Shuffle，将数据发往 Reduce 端，各组数据在 Reduce 端完成最终的聚合运算。

如果 group by 分组字段的值分布不均，就可能导致大量相同的 key 进入同一 Reduce，从而导致数据倾斜问题。

由分组聚合导致的数据倾斜问题，有以下两种解决思路：

1）Map-Side 聚合

开启 Map-Side 聚合后，数据会现在 Map 端完成部分聚合工作。这样一来即便原始数据是倾斜的，经过 Map 端的初步聚合后，发往 Reduce 的数据也就不再倾斜了。最佳状态下，Map-端聚合能完全屏蔽数据倾斜问题。

相关参数如下：

```
--启用 map-side 聚合
set hive.map.aggr=true;

--用于检测源表数据是否适合进行 map-side 聚合。检测的方法是：先对若干条数据进行 map-side 聚合，若聚合后的条数和聚合前的条数比值小于该值，则认为该表适合进行 map-side 聚合；否则，认为该表数据不适合进行 map-side 聚合，后续数据便不再进行 map-side 聚合。
set hive.map.aggr.hash.min.reduction=0.5;

--用于检测源表是否适合 map-side 聚合的条数。
set hive.groupby.mapaggr.checkinterval=100000;

--map-side 聚合所用的 hash table，占用 map task 堆内存的最大比例，若超出该值，则会对 hash table 进行一次 flush。
```

```
set hive.map.aggr.hash.force.flush.memory.threshold=0.9;
```

2）Skew-GroupBy 优化

Skew-GroupBy 的原理是启动两个 MR 任务，第一个 MR 按照随机数分区，将数据分散发送到 Reduce，完成部分聚合，第二个 MR 按照分组字段分区，完成最终聚合。

相关参数如下：

```
--启用分组聚合数据倾斜优化
set hive.groupby.skewindata=true;
```

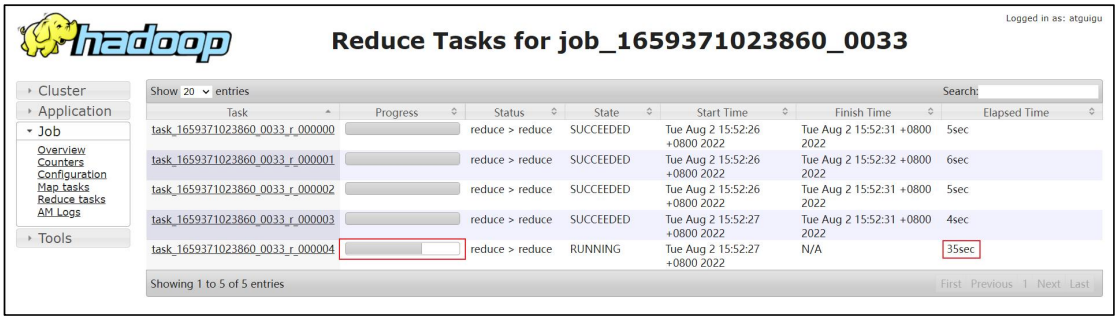
10.6.2.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    province_id,
    count(*)
from order_detail
group by province_id;
```

2）优化前

该表数据中的 province_id 字段是存在倾斜的，若不经过优化，通过观察任务的执行过程，是能够看出数据倾斜现象的。



需要注意的是，hive 中的 map-side 聚合是默认开启的，若想看到数据倾斜的现象，需要先将 hive.map.aggr 参数设置为 false。

3）优化思路

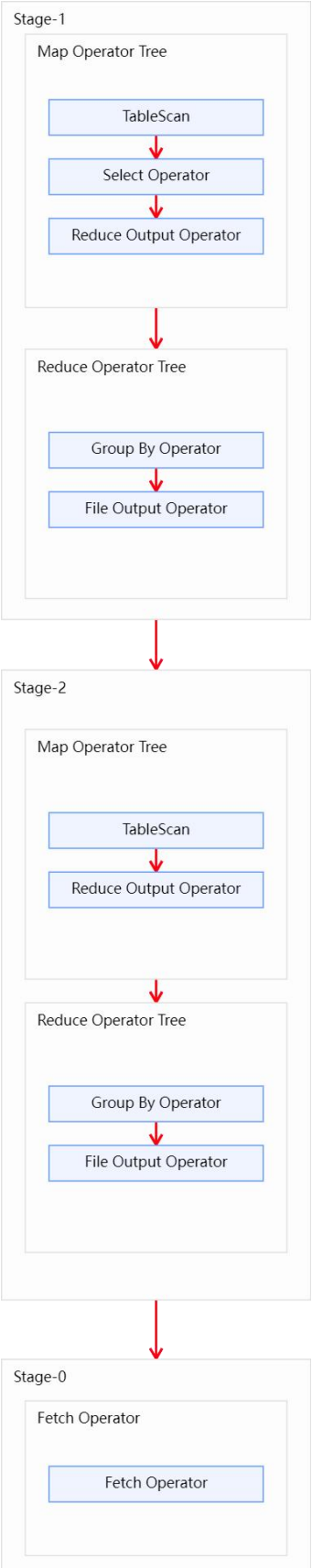
通过上述两种思路均可解决数据倾斜的问题。下面分别进行说明：

（1）Map-Side 聚合

设置如下参数

```
--启用 map-side 聚合
set hive.map.aggr=true;
--关闭 skew-groupby
set hive.groupby.skewindata=false;
```

开启 map-side 聚合后的执行计划如下图所示：



很明显可以看到开启 map-side 聚合后，reduce 数据不再倾斜。

Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Shuffle Finish Time	Merge Finish Time	Finish Time	Elapsed Time Shuffle	Elapsed Time Merge	Elapsed Time Reduce	Elapsed Time
task_1660095370483_0019_r_000000	SUCCEEDED	Wed Aug 10 22:52:04 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	5sec	Wed Aug 10 22:52:04 +0800 2022	Wed Aug 10 22:52:07 +0800 2022	Wed Aug 10 22:52:07 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	2sec	0sec	2sec	5sec
task_1660095370483_0019_r_000001	SUCCEEDED	Wed Aug 10 22:52:04 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	7sec	Wed Aug 10 22:52:04 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	5sec	0sec	1sec	7sec
task_1660095370483_0019_r_000002	SUCCEEDED	Wed Aug 10 22:52:05 +0800 2022	Wed Aug 10 22:52:10 +0800 2022	5sec	Wed Aug 10 22:52:05 +0800 2022	Wed Aug 10 22:52:08 +0800 2022	Wed Aug 10 22:52:08 +0800 2022	Wed Aug 10 22:52:10 +0800 2022	3sec	0sec	2sec	5sec
task_1660095370483_0019_r_000003	SUCCEEDED	Wed Aug 10 22:52:05 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	6sec	Wed Aug 10 22:52:05 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	Wed Aug 10 22:52:09 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	4sec	0sec	1sec	6sec
task_1660095370483_0019_r_000004	SUCCEEDED	Wed Aug 10 22:52:06 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	5sec	Wed Aug 10 22:52:06 +0800 2022	Wed Aug 10 22:52:10 +0800 2022	Wed Aug 10 22:52:10 +0800 2022	Wed Aug 10 22:52:11 +0800 2022	4sec	0sec	1sec	5sec

（2）Skew-GroupBy 优化

设置如下参数

```
--启用 skew-groupby
set hive.groupby.skewindata=true;
--关闭 map-side 聚合
set hive.map.aggr=false;
```

开启 Skew-GroupBy 优化后，可以很明显看到该 sql 执行在 yarn 上启动了两个 mr 任务，第一个 mr 打散数据，第二个 mr 按照打散后的数据进行分组聚合。

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocate CPU VCores
application_1660095370483_0021	atguigu	select province_id, ...province_id (Stage-2)	MAPREDUCE	default	0	Wed Aug 10 22:54:32 +0800 2022	Wed Aug 10 22:54:17 +0800 2022	Wed Aug 10 22:54:32 +0800 2022	FINISHED	SUCCEEDED	N/A	N/A
application_1660095370483_0020	atguigu	select province_id, ...province_id (Stage-1)	MAPREDUCE	default	0	Wed Aug 10 22:53:36 +0800 2022	Wed Aug 10 22:53:37 +0800 2022	Wed Aug 10 22:54:10 +0800 2022	FINISHED	SUCCEEDED	N/A	N/A

第二个mr对打散的数据进行聚合

第一个mr打散数据

10.6.3 Join 导致的数据倾斜

10.6.3.1 优化说明

前文提到过，未经优化的 join 操作，默认是使用 common join 算法，也就是通过一个 MapReduce Job 完成计算。Map 端负责读取 join 操作所需表的数据，并按照关联字段进行分区，通过 Shuffle，将其发送到 Reduce 端，相同 key 的数据在 Reduce 端完成最终的 Join 操作。

如果关联字段的值分布不均，就可能导致大量相同的 key 进入同一 Reduce，从而导致数据倾斜问题。

由 join 导致的数据倾斜问题，有如下三种解决方案：

1）map join

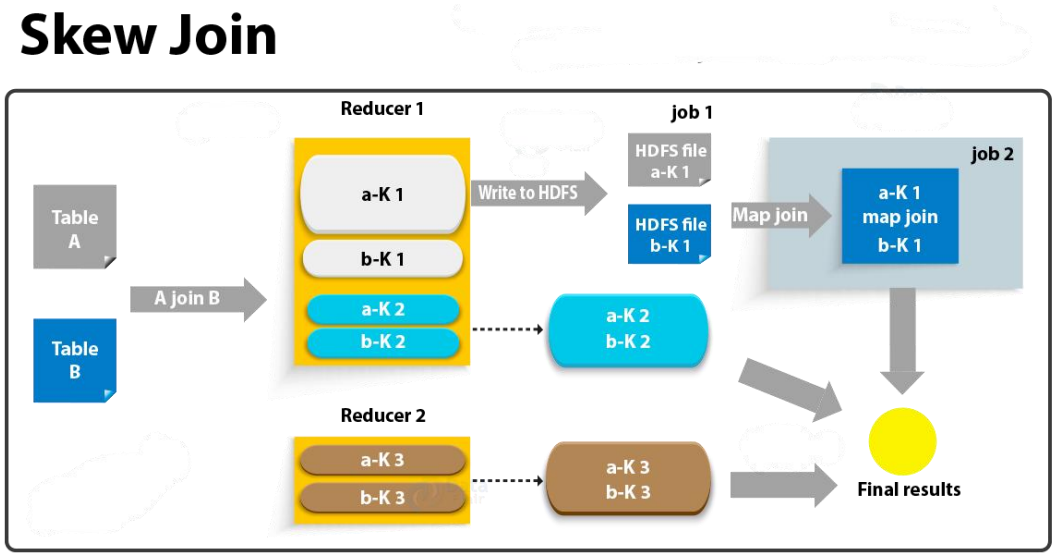
使用 map join 算法，join 操作仅在 map 端就能完成，没有 shuffle 操作，没有 reduce 阶段，自然不会产生 reduce 端的数据倾斜。该方案适用于大表 join 小表时发生数据倾斜的场景。

相关参数如下：

```
--启动 Map Join 自动转换
set hive.auto.convert.join=true;

--一个 Common Join operator 转为 Map Join operator 的判断条件,若该 Common Join 相关的表中,存在 n-1 张表的大小总和<=该值,则生成一个 Map Join 计划,此时可能存在多种 n-1 张表的组合均满足该条件,则 hive 会为每种满足条件的组合均生成一个 Map Join 计划,同时还会保留原有的 Common Join 计划作为后备(back up)计划,实际运行时,优先执行 Map Join 计划，若不能执行成功，则启动 Common Join 后备计划。
set hive.mapjoin.smalltable.filesize=250000;

--开启无条件转 Map Join
set hive.auto.convert.join.noconditionaltask=true;

--无条件转 Map Join 时的小表之和阈值,若一个 Common Join operator 相关的表中，存在 n-1 张表的大小总和<=该值,此时 hive 便不会再为每种 n-1 张表的组合均生成 Map Join 计划,同时也不会保留 Common Join 作为后备计划。而是只生成一个最优的 Map Join 计划。
set hive.auto.convert.join.noconditionaltask.size=10000000;
```

2）skew join

skew join 的原理是，为倾斜的大 key 单独启动一个 map join 任务进行计算，其余 key 进行正常的 common join。原理图如下：



相关参数如下：

```
--启用 skew join 优化
set hive.optimize.skewjoin=true;
```

```
--触发 skew join 的阈值，若某个 key 的行数超过该参数值，则触发
set hive.skewjoin.key=100000;
```

这种方案对参与 join 的源表大小没有要求，但是对两表中倾斜的 key 的数据量有要求，要求一张表中的倾斜 key 的数据量比较小（方便走 mapjoin）。

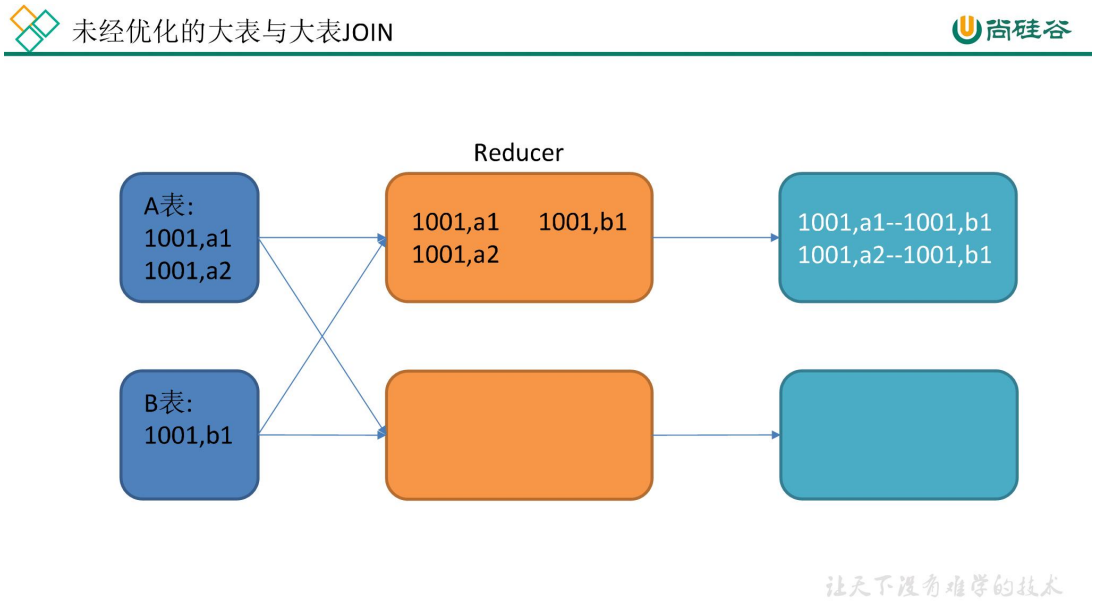
3）调整 SQL 语句

若参与 join 的两表均为大表，其中一张表的数据是倾斜的，此时也可通过以下方式对 SQL 语句进行相应的调整。

假设原始 SQL 语句如下：A，B 两表均为大表，且其中一张表的数据是倾斜的。

```
hive (default)>
select
    *
from A
join B
on A.id=B.id;
```

其 join 过程如下：



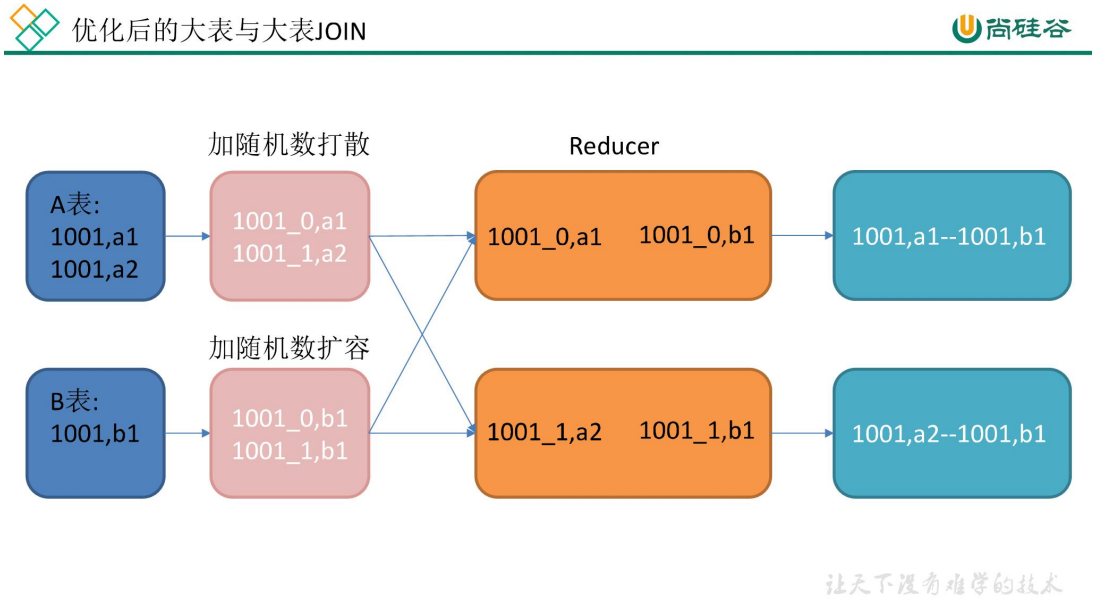
图中 1001 为倾斜的大 key，可以看到，其被发往了同一个 Reduce 进行处理。

调整 SQL 语句如下：

```
hive (default)>
select
    *
from(
    select --打散操作
        concat(id,'_',cast(rand()*2 as int)) id,
        value
    from A
)ta
join(
    select --扩容操作
```

```
        concat(id,'_',0) id,
        value
    from B
    union all
    select
        concat(id,'_',1) id,
        value
    from B
)tb
on ta.id=tb.id;
```

调整之后的 SQL 语句执行计划如下图所示：



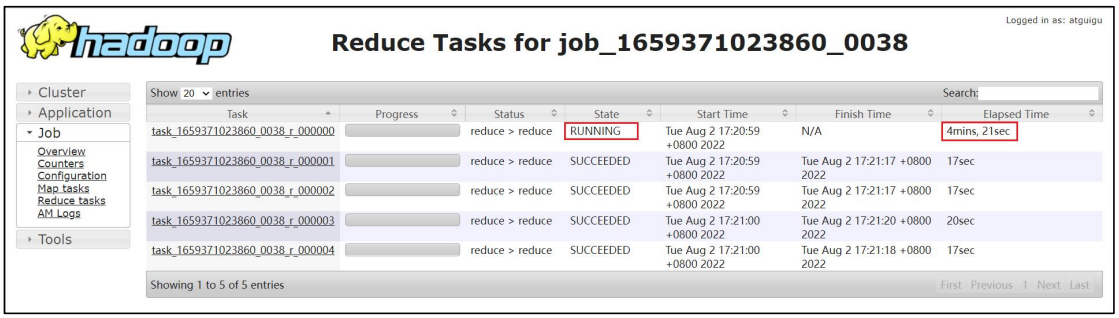
10.6.3.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    *
from order_detail od
join province_info pi
on od.province_id=pi.id;
```

2）优化前

order_detail 表中的 province_id 字段是存在倾斜的，若不经过优化，通过观察任务的执行过程，是能够看出数据倾斜现象的。



需要注意的是，hive 中的 map join 自动转换是默认开启的，若想看到数据倾斜的现象，需要先将 hive.auto.convert.join 参数设置为 false。

3）优化思路

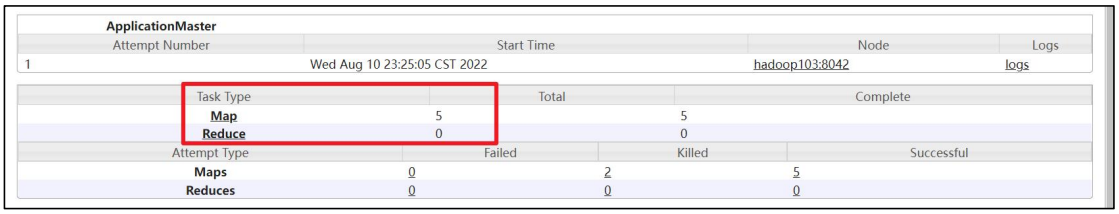
上述两种优化思路均可解决该数据倾斜问题，下面分别进行说明：

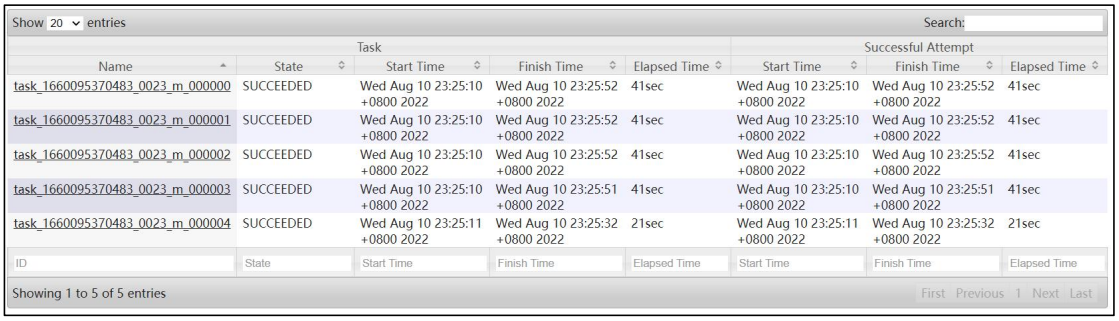
（1）map join

设置如下参数

```
--启用 map join
set hive.auto.convert.join=true;
--关闭 skew join
set hive.optimize.skewjoin=false;
```

可以很明显看到开启 map join 以后，mr 任务只有 map 阶段，没有 reduce 阶段，自然也就不会有数据倾斜发生。





（2）skew join

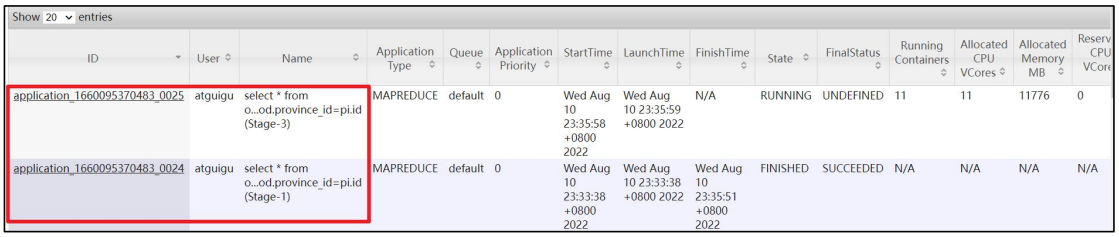
设置如下参数

```
--启动 skew join
set hive.optimize.skewjoin=true;
--关闭 map join
set hive.auto.convert.join=false;
```

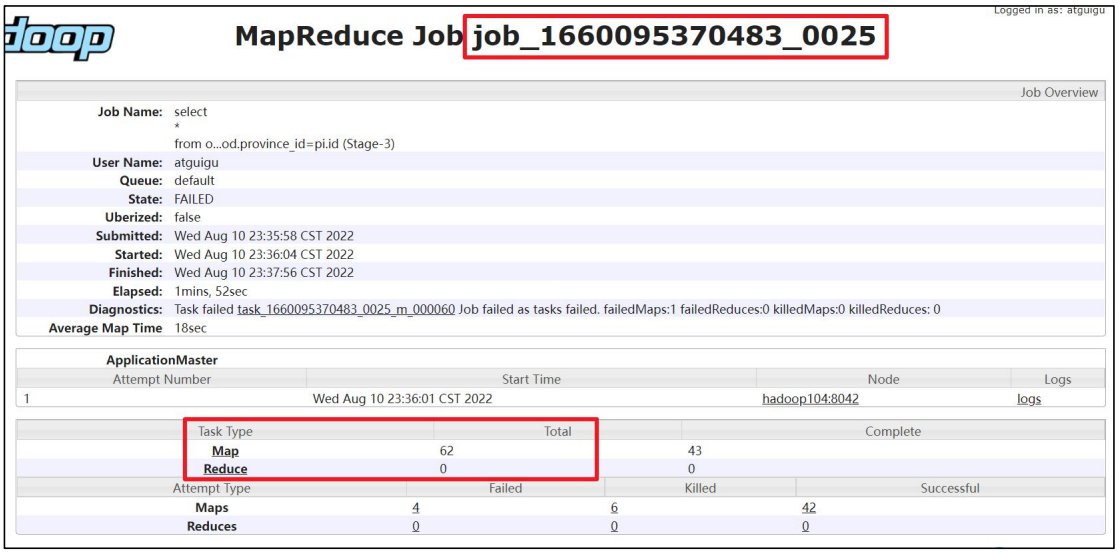
开启 skew join 后，使用 explain 可以很明显看到执行计划如下图所示，说明 skew join 生效，任务既有 common join，又有部分 key 走了 map join。

Stage-1
Map Operator Tree
TableScan
Filter Operator
Select Operator
Reduce Output Operator
TableScan
Filter Operator
Select Operator
Reduce Output Operator
Reduce Operator Tree
Join Operator
File Output Operator
Stage-4
Conditional Operator
Conditional Operator
Stage-5
Map Local Operator Tree
TableScan
HashTable Sink Operator
Stage-3
Map Operator Tree
TableScan
Map Join Operator
File Output Operator
Stage-0
Fetch Operator
Fetch Operator

并且该 sql 在 yarn 上最终启动了两个 mr 任务，而且第二个任务只有 map 没有 reduce 阶段，说明第二个任务是对倾斜的 key 进行了 map join。



ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserv CPU VCor
application_1660095370483_0025	atguigu	select * from o...od.province_id=pi.id (Stage-3)	MAPREDUCE	default	0	Wed Aug 10 23:35:58 +0800 2022	Wed Aug 10 23:35:59 +0800 2022	N/A	RUNNING	UNDEFINED	11	11	11776	0
application_1660095370483_0024	atguigu	select * from o...od.province_id=pi.id (Stage-1)	MAPREDUCE	default	0	Wed Aug 10 23:33:38 +0800 2022	Wed Aug 10 23:33:38 +0800 2022	Wed Aug 10 23:35:51 +0800 2022	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A



MapReduce Job job_1660095370483_0025

Job Overview

Job Name: select * from o...od.province_id=pi.id (Stage-3)
User Name: atguigu
Queue: default
State: FAILED
Uberized: false
Submitted: Wed Aug 10 23:35:58 CST 2022
Started: Wed Aug 10 23:36:04 CST 2022
Finished: Wed Aug 10 23:37:56 CST 2022
Elapsed: 1mins, 52sec
Diagnostics: Task failed task_1660095370483_0025_m_000060 Job failed as tasks failed. failedMaps:1 failedReduces:0 killedMaps:0 killedReduces: 0
Average Map Time 18sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Aug 10 23:36:01 CST 2022	hadoop104:8042	logs

Task Type	Total	Complete
Map	62	43
Reduce	0	0

Attempt Type	Failed	Killed	Successful
Maps	4	6	42
Reduces	0	0	0

10.7 HQL 语法优化之任务并行度

10.7.1 优化说明

对于一个分布式的计算任务而言，设置一个合适的并行度十分重要。Hive 的计算任务由 MapReduce 完成，故并行度的调整需要分为 Map 端和 Reduce 端。

10.7.1.1 Map 端并行度

Map 端的并行度，也就是 Map 的个数。是由输入文件的切片数决定的。一般情况下，Map 端的并行度无需手动调整。

以下特殊情况可考虑调整 map 端并行度：

1）查询的表中存在大量小文件

按照 Hadoop 默认的切片策略，一个小文件会单独启动一个 map task 负责计算。若查询的表中存在大量小文件，则会启动大量 map task，造成计算资源的浪费。这种情况下，可以使用 Hive 提供的 CombineHiveInputFormat，多个小文件合并为一个切片，从而控制 map task 个数。相关参数如下：

```
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
```

2）map 端有复杂的查询逻辑

若 SQL 语句中有正则替换、json 解析等复杂耗时的查询逻辑时，map 端的计算会相对慢一些。若想加快计算速度，在计算资源充足的情况下，可考虑增大 map 端的并行度，令 map task 多一些，每个 map task 计算的数据少一些。相关参数如下：

```
--一个切片的最大值
set mapreduce.input.fileinputformat.split.maxsize=256000000;
```

10.7.1.2 Reduce 端并行度

Reduce 端的并行度，也就是 Reduce 个数。相对来说，更需要关注。Reduce 端的并行度，可由用户自己指定，也可由 Hive 自行根据该 MR Job 输入的文件大小进行估算。

Reduce 端的并行度的相关参数如下：

```
--指定 Reduce 端并行度，默认值为-1，表示用户未指定
set mapreduce.job.reduces;
--Reduce 端并行度最大值
set hive.exec.reducers.max;
--单个 Reduce Task 计算的数据量，用于估算 Reduce 并行度
set hive.exec.reducers.bytes.per.reducer;
```

Reduce 端并行度的确定逻辑如下：

若指定参数 mapreduce.job.reduces 的值为一个非负整数，则 Reduce 并行度为指定值。否则，Hive 自行估算 Reduce 并行度，估算逻辑如下：

假设 Job 输入的文件大小为 totalInputBytes

参数 hive.exec.reducers.bytes.per.reducer 的值为 bytesPerReducer。

参数 hive.exec.reducers.max 的值为 maxReducers。

则 Reduce 端的并行度为：

$$\min\left(ceil\left(\frac{totalInputBytes}{bytesPerReducer}\right), maxReducers\right)$$

根据上述描述，可以看出，Hive 自行估算 Reduce 并行度时，是以整个 MR Job 输入的文件大小作为依据的。因此，在某些情况下其估计的并行度很可能并不准确，此时就需要用户根据实际情况来指定 Reduce 并行度了。

10.7.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    province_id,
    count(*)
from order_detail
group by province_id;
```

2）优化前

上述 sql 语句，在不指定 Reduce 并行度时，Hive 自行估算并行度的逻辑如下：

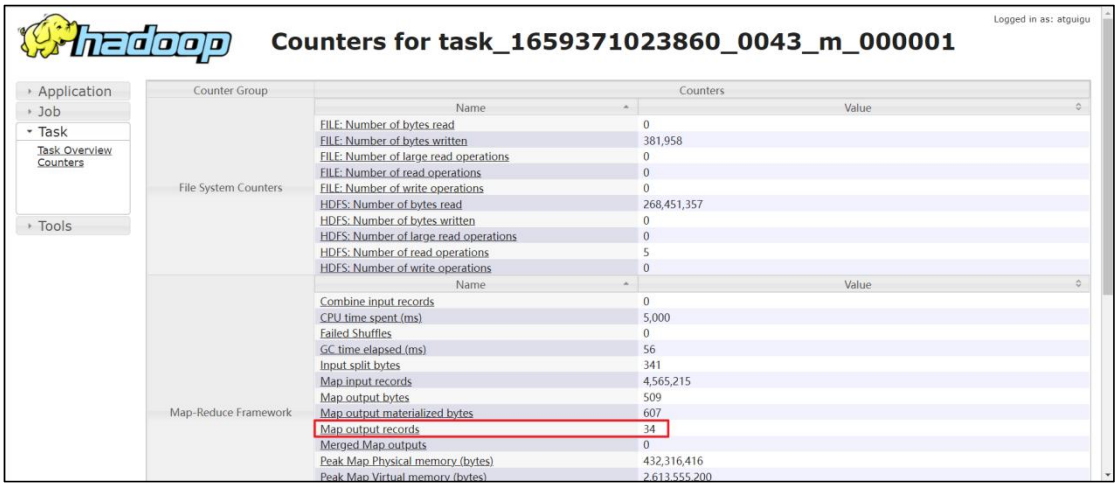
```
totalInputBytes= 1136009934
bytesPerReducer=256000000
maxReducers=1009
```

经计算，Reduce 并行度为

$$\text{numReducers} = \min\left(ceil\left(\frac{\mathbf{1136009934}}{\mathbf{256000000}}\right), \mathbf{1009}\right) = 5$$

3）优化思路

上述 sql 语句，在默认情况下，是会进行 map-side 聚合的，也就是 Reduce 端接收的数据，实际上是 map 端完成聚合之后的结果。观察任务的执行过程，会发现，每个 map 端输出的数据只有 34 条记录，共有 5 个 map task。



也就是说 Reduce 端实际只会接收 170（34*5）条记录，故理论上 Reduce 端并行度设置为 1 就足够了。这种情况下，用户可通过以下参数，自行设置 Reduce 端并行度为 1。

```
--指定 Reduce 端并行度，默认值为-1，表示用户未指定
set mapreduce.job.reduces=1;
```

10.8 HQL 语法优化之小文件合并

10.8.1 优化说明

小文件合并优化，分为两个方面，分别是 Map 端输入的小文件合并，和 Reduce 端输出的小文件合并。

10.8.1.1 Map 端输入文件合并

合并 Map 端输入的小文件，是指将多个小文件划分到一个切片中，进而由一个 Map Task 去处理。目的是防止为单个小文件启动一个 Map Task，浪费计算资源。

相关参数为：

```
--可将多个小文件切片，合并为一个切片，进而由一个 map 任务处理
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
```

10.8.1.2 Reduce 输出文件合并

合并 Reduce 端输出的小文件，是指将多个小文件合并成大文件。目的是减少 HDFS 小文件数量。其原理是根据计算任务输出文件的平均大小进行判断，若符合条件，则单独启动一个额外的任务进行合并。

相关参数为：

```
--开启合并 map only 任务输出的小文件
set hive.merge.mapfiles=true;

--开启合并 map reduce 任务输出的小文件
set hive.merge.mapredfiles=true;

--合并后的文件大小
set hive.merge.size.per.task=256000000;

--触发小文件合并任务的阈值，若某计算任务输出的文件平均大小低于该值，则触发合并
set hive.merge.smallfiles.avgsize=16000000;
```

10.8.2 优化案例

1）示例用表

现有一个需求，计算各省份订单金额总和，下表为结果表。

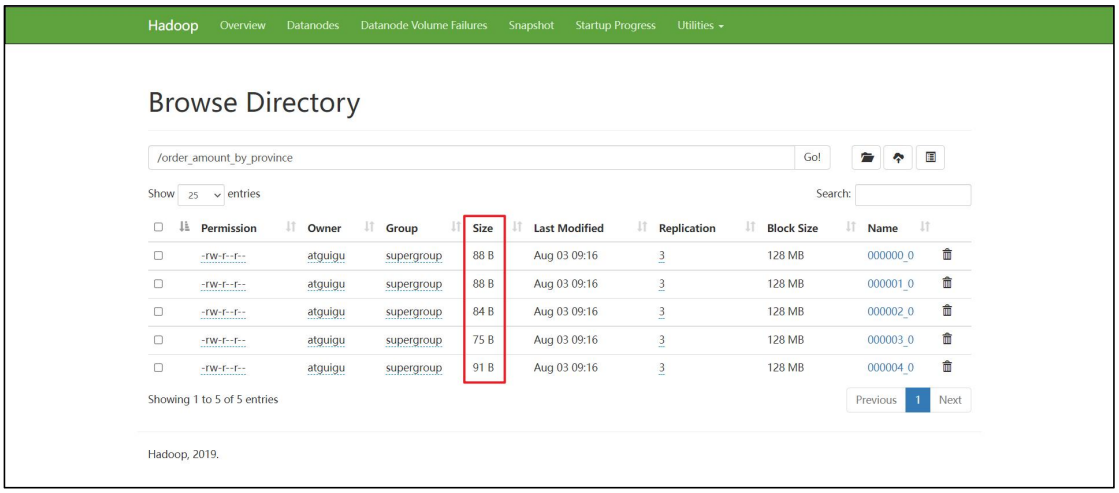
```
hive (default)>
drop table if exists order_amount_by_province;
create table order_amount_by_province(
    province_id string comment '省份 id',
    order_amount decimal(16,2) comment '订单金额'
)
location '/order_amount_by_province';
```

2）示例 SQL 语句

```
hive (default)>
insert overwrite table order_amount_by_province
select
    province_id,
    sum(total_amount)
from order_detail
group by province_id;
```

3）优化前

根据任务并行度一节所需内容，可分析出，默认情况下，该 sql 语句的 Reduce 端并行度为 5，故最终输出文件个数也为 5，下图为输出文件，可以看出，5 个均为小文件。



4）优化思路

若想避免小文件的产生，可采取方案有两个。

（1）合理设置任务的 Reduce 端并行度

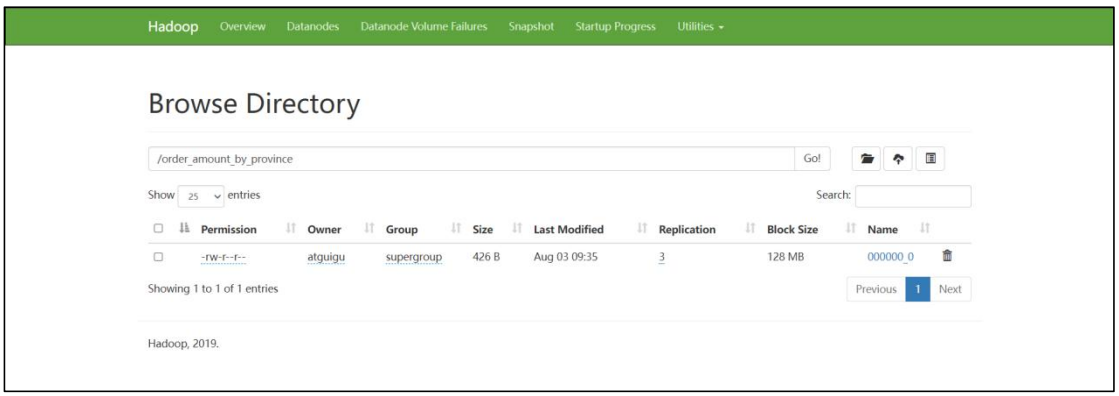
若将上述计算任务的并行度设置为 1，就能保证其输出结果只有一个文件。

（2）启用 Hive 合并小文件优化

设置以下参数：

```
--开启合并 map reduce 任务输出的小文件
set hive.merge.mapredfiles=true;

--合并后的文件大小
set hive.merge.size.per.task=256000000;

--触发小文件合并任务的阈值，若某计算任务输出的文件平均大小低于该值，则触发合并
set hive.merge.smallfiles.avgsize=16000000;
```

再次执行上述的 insert 语句，观察结果表中的文件，只剩一个了。



10.9 其他优化

10.9.1 CBO 优化

10.9.1.1 优化说明

CBO 是指 Cost based Optimizer，即基于计算成本的优化。

在 Hive 中，计算成本模型考虑到了：数据的行数、CPU、本地 IO、HDFS IO、网络 IO 等方面。Hive 会计算同一 SQL 语句的不同执行计划的计算成本，并选出成本最低的执行计划。目前 CBO 在 hive 的 MR 引擎下主要用于 join 的优化，例如多表 join 的 join 顺序。

相关参数为：

```
--是否启用 cbo 优化
set hive.cbo.enable=true;
```

10.9.2.2 优化案例

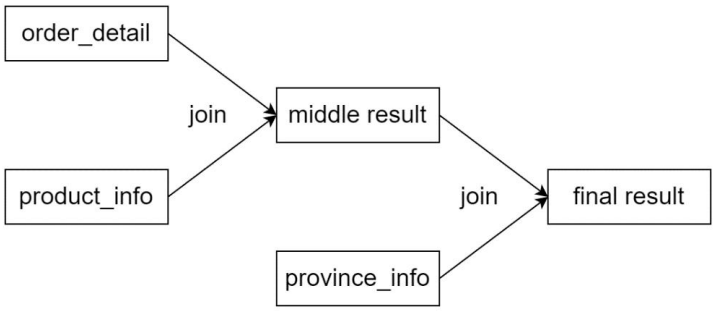
1）示例 SQL 语句

```
hive (default)>
select
    *
from order_detail od
join product_info product on od.product_id=product.id
join province_info province on od.province_id=province.id;
```

2）关闭 CBO 优化

```
--关闭 cbo 优化
set hive.cbo.enable=false;

--为了测试效果更加直观，关闭 map join 自动转换
set hive.auto.convert.join=false;
```

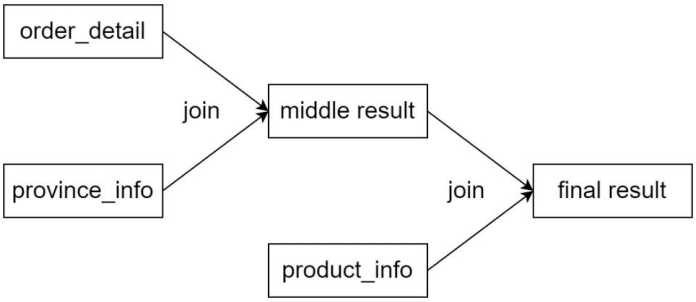
根据执行计划，可以看出，三张表的 join 顺序如下：



3）开启 CBO 优化

```
--开启 cbo 优化
set hive.cbo.enable=true;
--为了测试效果更加直观，关闭 map join 自动转换
set hive.auto.convert.join=false;
```

根据执行计划，可以看出，三张表的 join 顺序如下：



4）总结

根据上述案例可以看出，CBO 优化对于执行计划中 join 顺序是有影响的，其之所以会将 province_info 的 join 顺序提前，是因为 province info 的数据量较小，将其提前，会有更大的概率使得中间结果的数据量变小，从而使整个计算任务的数据量减小，也就是使计算成本变小。

10.9.2 谓词下推

10.9.2.1 优化说明

谓词下推（predicate pushdown）是指，尽量将过滤操作前移，以减少后续计算步骤的数据量。

相关参数为：

```
--是否启动谓词下推（predicate pushdown）优化
set hive.optimize.ppd = true;
```

需要注意的是：

CBO 优化也会完成一部分的谓词下推优化工作，因为在执行计划中，谓词越靠前，整个计划的计算成本就会越低。

10.9.2.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    *
from order_detail
join province_info
where order_detail.province_id='2';
```

2）关闭谓词下推优化

```
--是否启动谓词下推（predicate pushdown）优化
set hive.optimize.ppd = false;

--为了测试效果更加直观，关闭 cbo 优化
set hive.cbo.enable=false;
```

通过执行计划可以看到，过滤操作位于执行计划中的 join 操作之后。

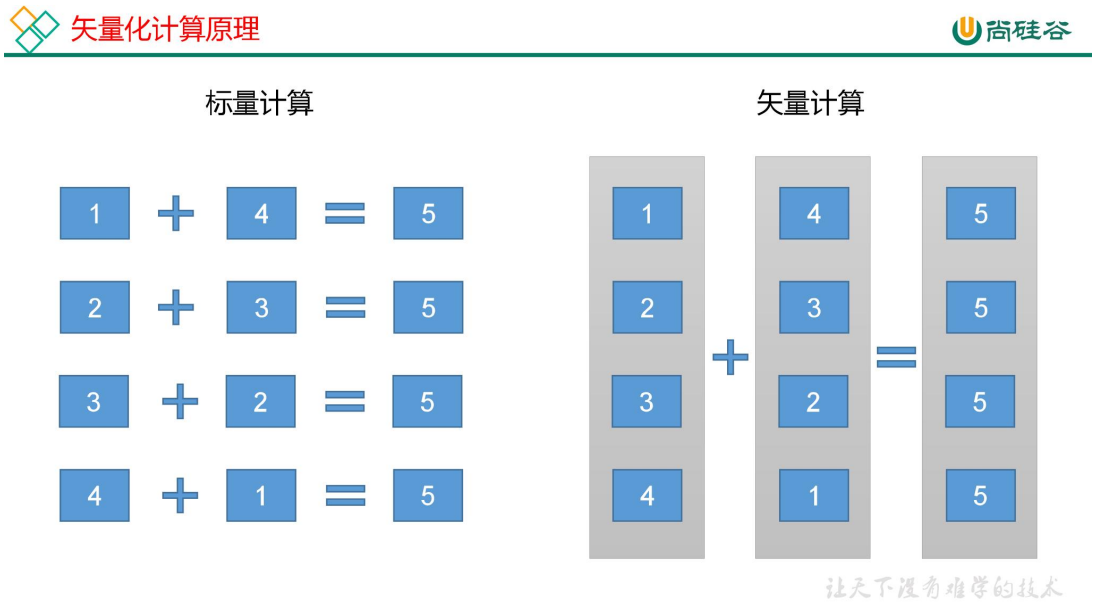
3）开启谓词下推优化

```
--是否启动谓词下推（predicate pushdown）优化
set hive.optimize.ppd = true;

--为了测试效果更加直观，关闭 cbo 优化
set hive.cbo.enable=false;
```

通过执行计划可以看出，过滤操作位于执行计划中的 join 操作之前。

10.9.3 矢量化查询

Hive 的矢量化查询优化，依赖于 CPU 的矢量化计算，CPU 的矢量化计算的基本原理如下图：



Hive 的矢量化查询，可以极大的提高一些典型查询场景（例如 scans, filters, aggregates, and joins）下的 CPU 使用效率。

相关参数如下：

```
set hive.vectorized.execution.enabled=true;
```

若执行计划中，出现“Execution mode: vectorized”字样，即表明使用了矢量化计算。

官网参考连接：

https://cwiki.apache.org/confluence/display/Hive/Vectorized+Query+Execution#VectorizedQueryExecution-Limitations

10.9.4 Fetch 抓取

Fetch 抓取是指，Hive 中对某些情况的查询可以不必使用 MapReduce 计算。例如：select * from emp;在这种情况下，Hive 可以简单地读取 emp 对应的存储目录下的文件，然后输出查询结果到控制台。

相关参数如下：

```
--是否在特定场景转换为 fetch 任务
--设置为 none 表示不转换
--设置为 minimal 表示支持 select *，分区字段过滤，Limit 等
--设置为 more 表示支持 select 任意字段,包括函数，过滤，和 limit 等
set hive.fetch.task.conversion=more;
```

10.9.5 本地模式

10.9.5.1 优化说明

大多数的 Hadoop Job 是需要 Hadoop 提供的完整的可扩展性来处理大数据集的。不过，有时 Hive 的输入数据量是非常小的。在这种情况下，为查询触发执行任务消耗的时间可能会比实际 job 的执行时间要多的多。对于大多数这种情况，Hive 可以通过本地模式在单台机器上处理所有的任务。对于小数据集，执行时间可以明显被缩短。

相关参数如下：

```
--开启自动转换为本地模式
set hive.exec.mode.local.auto=true;

--设置 local MapReduce 的最大输入数据量，当输入数据量小于这个值时采用
local  MapReduce 的方式，默认为 134217728，即 128M
set hive.exec.mode.local.auto.inputbytes.max=50000000;

--设置 local MapReduce 的最大输入文件个数，当输入文件个数小于这个值时采用
local MapReduce 的方式，默认为 4
set hive.exec.mode.local.auto.input.files.max=10;
```

10.9.5.2 优化案例

1）示例 SQL 语句

```
hive (default)>
select
    count(*)
from product_info
group by category_id;
```

2）关闭本地模式

```
set hive.exec.mode.local.auto=false;
```

3）开启本地模式

```
set hive.exec.mode.local.auto=true;
```

10.9.6 并行执行

Hive 会将一个 SQL 语句转化成一个或者多个 Stage，每个 Stage 对应一个 MR Job。默认情况下，Hive 同时只会执行一个 Stage。但是某 SQL 语句可能会包含多个 Stage，但这多个 Stage 可能并非完全互相依赖，也就是说有些 Stage 是可以并行执行的。此处提到的并行执行就是指这些 Stage 的并行执行。相关参数如下：

```
--启用并行执行优化
set hive.exec.parallel=true;

--同一个 sql 允许最大并行度，默认为 8
set hive.exec.parallel.thread.number=8;
```

10.9.7 严格模式

Hive 可以通过设置某些参数防止危险操作：

1）分区表不使用分区过滤

将 hive.strict.checks.no.partition.filter 设置为 true 时，对于分区表，除非 where 语句中含有分区字段过滤条件来限制范围，否则不允许执行。换句话说，就是用户不允许扫描所有分区。进行这个限制的原因是，通常分区表都拥有非常大的数据集，而且数据增加迅速。没有进行分区限制的查询可能会消耗令人不可接受的巨大资源来处理这个表。

2）使用 order by 没有 limit 过滤

将 hive.strict.checks.orderby.no.limit 设置为 true 时，对于使用了 order by 语句的查询，要求必须使用 limit 语句。因为 order by 为了执行排序过程会将所有的结果数据分发到同一个 Reduce 中进行处理，强制要求用户增加这个 limit 语句可以防止 Reduce 额外执行很长一段时间（开启了 limit 可以在数据进入到 Reduce 之前就减少一部分数据）。

3）笛卡尔积

将 hive.strict.checks.cartesian.product 设置为 true 时，会限制笛卡尔积的查询。对关系型数据库非常了解的用户可能期望在执行 JOIN 查询的时候不使用 ON 语句而是使用 where 语句，这样关系数据库的执行优化器就可以高效地将 WHERE 语句转化成那个 ON 语句。不幸的是，Hive 并不会执行这种优化，因此，如果表足够大，那么这个查询就会出现不可控的情况。

附录：常见错误及解决方案

1）连接不上 MySQL 数据库

——

（1）导错驱动包，应该把 mysql-connector-java-5.1.27-bin.jar 导入/opt/module/hive/lib 的不是这个包。错把 mysql-connector-java-5.1.27.tar.gz 导入 hive/lib 包下。

（2）修改 user 表中的主机名称没有都修改为%，而是修改为 localhost

2）Hive 默认的输入格式处理是 CombineHiveInputFormat，会对小文件进行合并。

```
hive (default)> set hive.input.format;
hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFo
rmat
```

可以采用 HiveInputFormat 就会根据分区数输出相应的文件。

```
hive (default)> set
hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
```

3）不能执行 MapReduce 程序

可能是 Hadoop 的 Yarn 没开启。

4）启动 MySQL 服务时，报 MySQL server PID file could not be found! 异常。

在/var/lock/subsys/mysql 路径下创建 hadoop102.pid，并在文件中添加内容：4396

5）报 service mysql status MySQL is not running, but lock file (/var/lock/subsys/mysql[失败]) 异常。

解决方案：在/var/lib/mysql 目录下创建：-rw-rw----. 1 mysql mysql 5 12 月 22 16:41 hadoop102.pid 文件，并修改权限为 777。

6）JVM 堆内存溢出（Hive 集群运行模式）

描述：java.lang.OutOfMemoryError: Java heap space

解决：在 yarn-site.xml 中加入如下代码。

```
<property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>2048</value>
</property>
<property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>2048</value>
</property>
<property>
    <name>yarn.nodemanager.vmem-pmem-ratio</name>
    <value>2.1</value>
</property>
<property>
    <name>mapred.child.java.opts</name>
    <value>-Xmx1024m</value>
</property>
```

7）JVM 堆内存溢出（Hive 本地运行模式）

描述：在启用 Hive 本地模式后，hive.log 报错 java.lang.OutOfMemoryError: Java heap

space

解决方案 1（临时）：

在 Hive 客户端临时设置 io.sort.mb 和 mapreduce.task.io.sort.mb 两个参数的值为 10。

```
0: jdbc:hive2://hadoop102:10000> set io.sort.mb;
+-----------------+
|        set        |
+-----------------+
| io.sort.mb=100  |
+-----------------+
1 row selected (0.008 seconds)
0: jdbc:hive2://hadoop102:10000> set mapreduce.task.io.sort.mb;
+--------------------------------+
|              set                    |
+--------------------------------+
| mapreduce.task.io.sort.mb=100   |
+--------------------------------+
1 row selected (0.008 seconds)
0: jdbc:hive2://hadoop102:10000> set io.sort.mb = 10;
No rows affected (0.005 seconds)
0: jdbc:hive2://hadoop102:10000> set mapreduce.task.io.sort.mb =
10;
No rows affected (0.004 seconds)
```

解决方案 2（永久生效）：

在$HIVE_HOME/conf 下添加 hive-env.sh。

```
[atguigu@hadoop102 conf]$ pwd
/opt/module/hive/conf
[atguigu@hadoop102 conf]$ cp hive-env.sh.template hive-env.sh
```

然后将其中的参数 export HADOOP_HEAPSIZE=1024 的注释放开，然后重启 Hive。

```
# The heap size of the jvm stared by hive shell script can be controlled via:
#
# export HADOOP_HEAPSIZE=1024
#
# Larger heap size may be required when running queries over large number of files or partitions.
# By default hive shell scripts use a heap size of 256 (MB).  Larger heap size would also be
# appropriate for hive server.
```

```
# The heap size of the jvm stared by hive shell script can be controlled via:
#
export HADOOP_HEAPSIZE=1024
#
# Larger heap size may be required when running queries over large number of files or partitions.
# By default hive shell scripts use a heap size of 256 (MB).  Larger heap size would also be
# appropriate for hive server.
```

8）虚拟内存限制

在 yarn-site.xml 中添加如下配置:

```
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
 </property>
```