1 数据仓库

1.1 概念

英文名称为Data Warehouse，可简写为DW或DWH。数据仓库的目的是构建面向分析的集成化数据环境，为企业提供决策支持（Decision Support）。

数据仓库是存储数据的，企业的各种数据会在数据仓库中存储，主要是为了分析有效数据，后续会基于它产出供分析挖掘的数据，或者数据应用需要的数据，如企业的分析性报告和各类报表等。

可以理解为：面向分析的存储系统。

1.2 特征

数据仓库是面向主题的（Subject-Oriented ）、集成的（Integrated）、非易失的（Non-Volatile）和时变的（Time-Variant ）数据集合，用以支持管理决策。

1.2.1 面向主题

数据仓库中的数据是按照一定的主题域进行组织的，每一个主题对应一个宏观的分析领域。数据仓库排除对于决策无用的数据，提供特定主题的简明视图。

1.2.2 集成性

数据仓库会将不同源数据库中的数据汇总到一起,数据仓库中的综合数据不能从原有的数据库系统直接得到。因此在数据进入数据仓库之前，必然要经过统一与整合，这一步是数据仓库建设中最关键、最复杂的一步(ETL)，要统一源数据中所有矛盾之处，如字段的同名异义、异名同义、单位不统一、字长不一致，等等。

1.2.3 非易失的

操作型数据库主要服务于日常的业务操作，使得数据库需要不断地对数据实时更新，以便迅速获得当前最新数据，不至于影响正常的业务运作。

在数据仓库中只要保存过去的业务数据，不需要每一笔业务都实时更新数据仓库，而是根据商业需要每隔一段时间把一批较新的数据导入数据仓库。 数据仓库的数据反映的是一段相当长的时间内历史数据的内容，是不同时点的数据库的集合，以及基于这些快照进行统计、综合和重组的导出数据。数据仓库中的数据一般仅执行查询操作，很少会有删除和更新。但是需定期加载和刷新数据。

1.2.4 时变的

数据仓库包含各种粒度的历史数据。数据仓库中的数据可能与某个特定日期、星期、月份、季度或者年份有关。数据仓库的目的是通过分析企业过去一段时间业务的经营状况，挖掘其中隐藏的模式。虽然数据仓库的用户不能修改数据，但并不是说数据仓库的数据是永远不变的。分析的结果只能反映过去的情况，当业务变化后，挖掘出的模式会失去时效性。因此数据仓库的数据需要定时更新，以适应决策的需要。

1.3 区别

数据库与数据仓库的区别实际讲的是**OLTP** 与**OLAP** 的区别。

操作型处理，叫联机事务处理 OLTP（On-Line Transaction Processing，），也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。

分析型处理，叫联机分析处理 OLAP（On-Line Analytical Processing）一般针对某些主题的历史数据进行分析，支持管理决策。

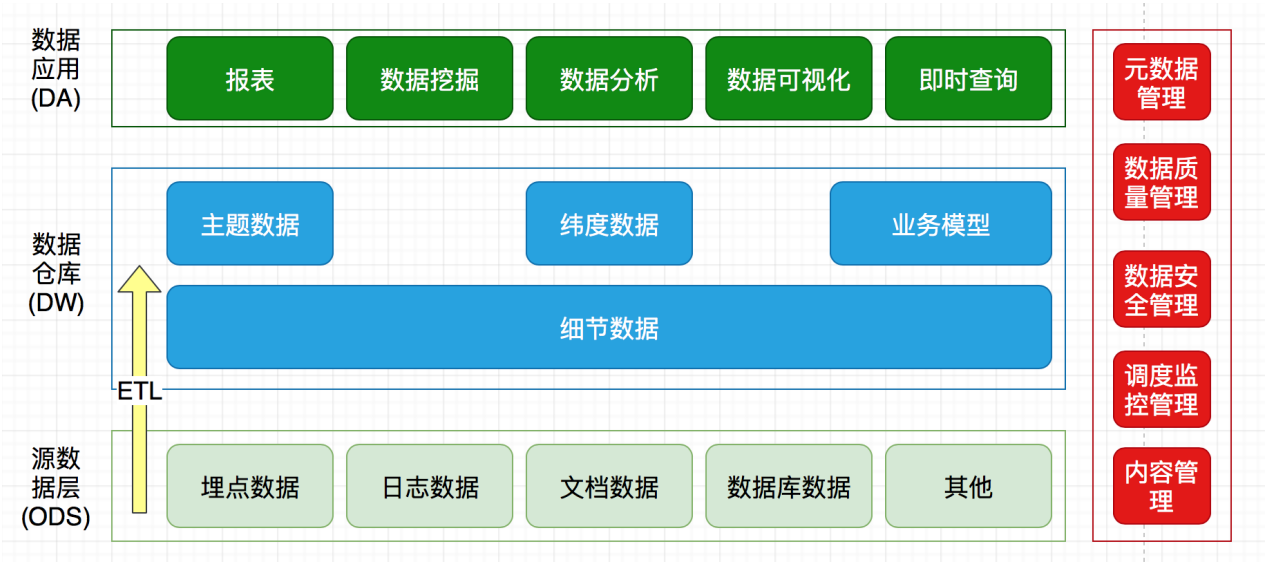
> 数据仓库的出现，并不是要取代数据库

操作型处理	分析型处理
细节的	综合的或提炼的
实体-关系（ER）模型	星型或雪花模型
存取瞬间数据	存储历史数据，不包含最近的数据
可更新的	只读，只追加
一次操作一个单元	一次操作一个集合
性能要求高，响应时间短	性能要求宽松
面向事务	面向分析
一次操作数据量小	一次操作数据量大
支持日常操作	支持决策需求
数据量小	数据量大
客户订单、库存水平和银行账户等	客户收益分析、市场细分等

> **数据仓库，是在数据库已经大量存在的情况下，为了进一步挖掘数据资源、为了决策需要而产生的，它绝不是所谓的“大型数据库”。**

1.4 架构

按照数据流入流出的过程，数据仓库架构可分为三层——源数据、数据仓库、数据应用。



数据仓库的数据来源于不同的源数据，并提供多样的数据应用，数据自下而上流入数据仓库后向上层开放应用，而数据仓库只是中间集成化数据管理的一个平台。

- `源数据层（ODS）`：此层数据无任何更改，直接沿用外围系统数据结构和数据，不对外开放；为临时存储层，是接口数据的临时存储区域，为后一步的数据处理做准备。
- `数据仓库层（DW）`：也称为细节层，DW层的数据应该是一致的、准确的、干净的数据，即对源系统数据进行了清洗（去除了杂质）后的数据。
- `数据应用层（DA或APP）`：前端应用直接读取的数据源；根据报表、专题分析需求而计算生成的数据。

数据仓库从各数据源获取数据及在数据仓库内的数据转换和流动都可以认为是ETL（抽取Extra, 转化Transfer, 装载Load）的过程，ETL是数据仓库的流水线，也可以认为是数据仓库的血液，它维系着数据仓库中数据的新陈代谢，而数据仓库日常的管理和维护工作的大部分精力就是保持ETL的正常和稳定。

1.5 元数据

元数据（Meta Date），主要记录数据仓库中模型的定义、各层级间的映射关系、监控数据仓库的数据状态及ETL的任务运行状态。一般会通过元数据资料库（Metadata Repository）来统一地存储和管理元数据，其主要目的是使数据仓库的设计、部署、操作和管理能达成协同和一致。

元数据是数据仓库管理系统的重要组成部分，元数据管理是企业级数据仓库中的关键组件，贯穿数据仓库构建的整个过程，直接影响着数据仓库的构建、使用和维护。

- 构建数据仓库的主要步骤之一是ETL。这时元数据将发挥重要的作用，它定义了源数据系统到数据仓库的映射、数据转换的规则、数据仓库的逻辑结构、数据更新的规则、数据导入历史记录以及装载周期等相关内容。数据抽取和转换的专家以及数据仓库管理员正是通过元数据高效地构建数据仓库。
- 用户在使用数据仓库时，通过元数据访问数据，明确数据项的含义以及定制报表。
- 数据仓库的规模及其复杂性离不开正确的元数据管理，包括增加或移除外部数据源，改变数据清洗

方法，控制出错的查询以及安排备份等。

元数据可分为技术元数据、业务元数据和管理元数据。

技术元数据：技术元数据指描述系统中技术细节相关的概念、关系和规则的数据，包括对数据结构、数据处理方面的描述，以及数据仓库、ETL、前端展现等技术细节方面的信息。技术元数据又细分为：

1. 数据源元数据
2. ETL元数据
3. 数据仓库元数据
4. BI元数据

业务元数据：业务元数据指从业务角度描述业务领域相关的概念、关系和规则的数据，包括业务术语和业务规则等信息。

管理元数据：管理元数据指描述管理领域相关的概念、关系和规则的数据，主要包括管理流程、人员组织、角色职责等信息。

元数据获取途径：。

1. 外部数据源：源系统、ETL工具、报表工具的元数据。
2. 数据仓库：数据库物理模型的元数据。
3. 手动录入：映射文档、任务配置、业务规则、业务术语。

2 Hive

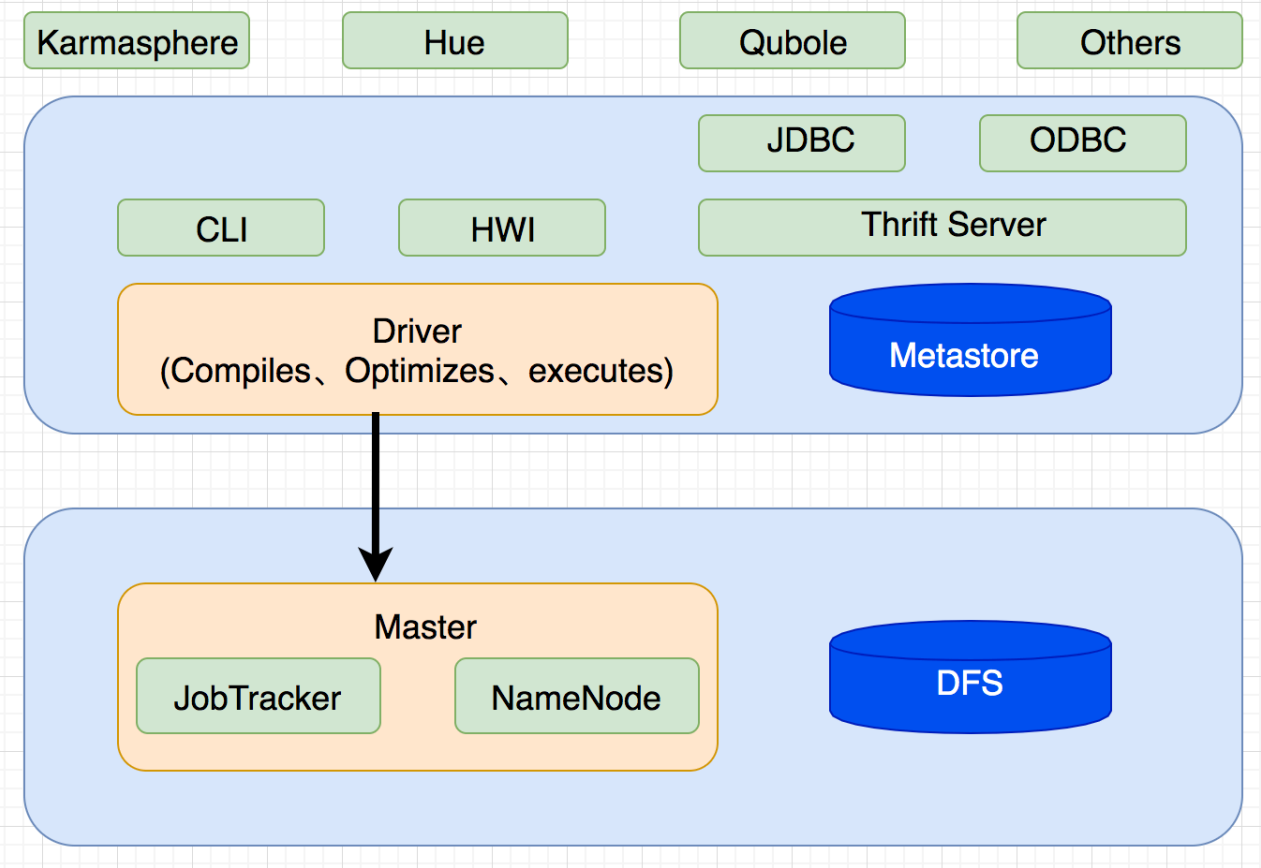
2.1 概念

Hive是基于Hadoop的一个数据仓库工具，可以将**结构化的数据**文件映射为一张数据库表，并提供类SQL查询功能。

- Hive提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 QL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者开发自定义的 mapper 和 reducer，来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作的问题。
- Hive是SQL解析引擎，它将SQL语句转译成M/R Job然后在Hadoop执行
- Hive的表其实就是HDFS的目录/文件，按表名把文件夹分开。如果是分区表，则分区值是子文件夹，可以直接在M/R Job里使用这些数据

> 为什么使用Hive?
>
> - 基于Hadoop的大数据的计算/扩展能力
> - 支持SQL like查询语言
> - 统一的元数据管理
> - 简单编程

2.2 架构

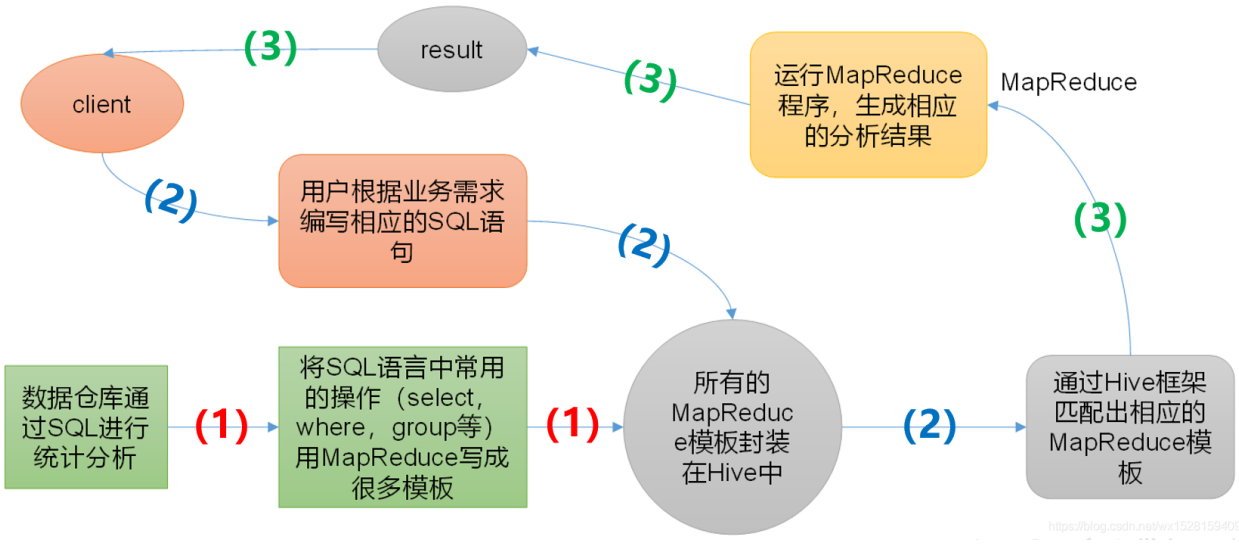


- **用户接口：** 包括CLI、JDBC/ODBC、WebGUI。其中，CLI(command line interface)为shell命令行；JDBC/ODBC是Hive的JAVA实现，与传统数据库JDBC类似；WebGUI是通过浏览器访问Hive。
- **元数据存储：** 通常是存储在关系数据库如mysql/derby中。Hive 将元数据存储在数据库中。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
- **解释器、编译器、优化器、执行器:** 完成HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在HDFS 中，并在随后由MapReduce 调用执行。

2.3 关系

Hive就是把sql语句转化为MapReduce程序。

Hive没有服务端，它本质是Hadoop或者说是HDFS的一个客户端，对HDFS的数据和Meta store的元数据进行操作。



1. 事先将常用的SQL操作封装成MapReduce模板存放在Hive中；
2. client依据实际需求写SQL语句，匹配对应的MapReduce模板，然后运行对应的MapReduce程序，
3. 生成相应的分析结果，返回给client。

> 为什么说Hive是基于Hadoop的呢?
>
> - Hive处理的数据实际存放在HDFS中
> - Hive分析数据的底层实现还是MapReduce程序
> - Hive调度资源时，用的是Yarn框架
> - 在服务器中运行Hive之前需要启动HDFS和YARN

Hive与传统数据库对比

1. 数据存储位置不同

   Hive中处理的结构化数据存储在HDFS中，元数据存储在mysql的Meta store中；

数据库将数据保存在块设备或本地文件系统中；

2. 数据更新

   Hive是针对数据仓库设计的，主要用于读，所有的数据在加载时已经确定好，适合处理静态数据；

   数据库通常是实时进行修改的，增删改查，适合处理动态数据；

3. 执行机制

   Hive大多数查询的执行是通过Hadoop提供的MapReduce实现的；

   数据库通常使用自己的引擎innodb；

4. 执行延迟

   Hive因为没有索引、利用MapReduce框架执行查询，所以Hive本身的延迟较高；

   数据库的延迟较低，但是不太适合处理PB级别以上海量数据；

处理海量数据时，Hive的优势就显现出来了；

5. 可扩展性

   Hive是建立在Hadoop上的，所以Hive也具备可扩展性，并发运行；

数据库由于ACID语义的严格限制，扩展性非常有限，例如目前最先进的并行数据库oracle在理论上扩展能力也就只有100台左右。

> 除了都用sql语句，Hive和数据库其实没啥太大关系。

2.4 安装

2.4.1 结合derby

`Hive内嵌derby搭建`

1. 把apache-hive-3.1.2-bin.tar.gz解压到家目录下software文件夹

```
tar -zxvf apache-hive-3.1.2-bin.tar.gz -C software/
```

2. 给解压的包构建软连接

```
cd software
ln -s apache-hive-3.1.2-bin hive
```

3. 编辑环境变量配置文件

```
cd
vi ~/.bashrc
```

4. 环境变量配置文件最后的位置增加内容

```
export HIVE_HOME=/home/briup/software/hive
export PATH=$PATH:$JAVA_HOME/bin:$HIVE_HOME/bin
```

5. 保存环境变量配置文件，并使环境变量配置文件生效

```
source ~/.bashrc
```

6. 初始化元数据库

```
schematool -dbType derby -initSchema
```

7. 开启命令行，启动hive软件

```
hive
```

> Hive基于内嵌derby搭建，derby是一个内置数据库，存储hive元数据，启动hive命令时必须是当前的位置
>
> derby一次只能打开一个会话，不支持多用户同时访问

2.4.2 Hive结合mysql搭建

1. 通过命令窗口登录数据库

```
mysql -uroot -proot
```

2. 创建数据库hive

```
create database hive;
```

3. 给hive数据库赋予权限

```
create user 'root'@'%' identified by 'root';
grant all privileges on *.* to 'root'@'%' with grant option;
flush privileges;
```

4. 把apache-hive-3.1.2-bin.tar.gz复制到hdfs主节点，解压压缩包

```
tar -zxvf apache-hive-3.1.2-bin.tar.gz -C software/
```

5. 给解压的包构建软连接

```
cd software
ln -s apache-hive-3.1.2-bin hive
```

6. 编辑环境变量配置文件

```
cd
vi ~/.bashrc
```

7. 环境变量配置文件最后的位置增加内容

```
export HIVE_HOME=/home/briup/software/hive
export PATH=$PATH:$JAVA_HOME/bin:$HIVE_HOME/bin
```

8. 保存环境变量配置文件，并使环境变量配置文件生效

```
source ~/.bashrc
```

9. 进入hive的安装配置文件路径

```
cd /home/briup/software/hive/conf/
```

10. 在conf下创建配置文件hive-site.xml

```
vi hive-site.xml
```

11. hive-site.xml中内容如下

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
        <!--连接数据库的url-->
        <property>
```

```
                <name>javax.jdo.option.ConnectionURL</name>
                <value>jdbc:mysql://master:3306/hive?
createDatabaseIfNotExist=true&amp;useSSL=false&amp;serverTimezone=UTC&amp;allowPublicK
eyRetrieval=true</value>
        </property>
        <!--数据库的驱动程序-->
        <property>
                <name>javax.jdo.option.ConnectionDriverName</name>
                <value>com.mysql.cj.jdbc.Driver</value>
        </property>
        <!--数据库的账号-->
        <property>
                <name>javax.jdo.option.ConnectionUserName</name>
                <value>root</value>
        </property>
        <!--数据库的密码-->
        <property>
                <name>javax.jdo.option.ConnectionPassword</name>
                <value>root</value>
        </property>
        <!--强制metastore和schema一致性，开启的话会校验在metastore中存储的信息版本和hive的jar包
中版本信息一致，并且关闭元数据的迁移，用户必须手动升级hive并且迁移schema，关闭的话只有在版本不一致的时
候给出警告，默认false,最简单的理解，关闭hive的元数据验证-->
        <property>
                <name>hive.metastore.schema.verification</name>
                <value>false</value>
        </property>
        <!--设置对外远程提供服务的端口，如jdbc连接；该服务需要启动hiveserver2服务，如果不需要远程
连接hive，可以省略-->
        <property>
                <name>hive.server2.thrift.port</name>
                <value>10000</value>
        </property>
        <!--设置对外远程提供服务，服务绑定的计算机-->
        <property>
                <name>hive.server2.thrift.bind.host</name>
                <value>0.0.0.0</value>
        </property>
        <!--hive数据仓库对应hdfs文件系统的路径，默认路径如下-->
        <property>
                <name>hive.metastore.warehouse.dir</name>
                <value>/user/hive/warehouse</value>
        </property>
        <!--查询表数据的时候显示列名，默认不显示，可选配置-->
        <property>
                <name>hive.cli.print.header</name>
                <value>true</value>
        </property>
        <!--客户端显示当前库-->
        <property>
                <name>hive.cli.print.current.db</name>
                <value>true</value>
```

```
        </property>
</configuration>
```

12. 将mysql-connector-java-8.0.30.jar上传到hive安装包lib下

```
cp mysql-connector-java-8.0.30.jar /home/briup/software/hive/lib/
```

13. 初始化元数据库

```
schematool  -dbType mysql -initSchema
```

```
hdfs@slave:~$ schematool  -dbType mysql -initSchema
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apache-hive-2.3.5-bin/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/hadoop-3.0.3/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Metastore connection URL:        jdbc:mysql://127.0.0.1:3306/hive?createDatabaseIfNotExist=true&useSSL=false&serverTimezone=UTC
Metastore Connection Driver :    com.mysql.jdbc.Driver
Metastore connection User:       root
Starting metastore schema initialization to 2.3.0
Initialization script hive-schema-2.3.0.mysql.sql
Initialization script completed
schemaTool completed
```

14. 开启命令行，启动hive软件

```
hive
```

```
hdfs@master:~$ hive
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/apache-hive-2.3.5-bin/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/hadoop-3.0.3/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]

Logging initialized using configuration in jar:file:/opt/apache-hive-2.3.5-bin/lib/hive-common-2.3.5.jar!/hive-log4j2.properties Async: true
Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
hive>
```

2.4.3 交互式操作

第一种方式：

基于命令hive开启命令行操作

```
hive
```

创建数据库

```
create database if not exists briup;
```

使用数据库

```
use briup;
```

第二种方式：

使用sql语句或者sql脚本进行交互

```
hive -e "create database if not exists briup"
```

第三种方式：

将hql写成sql脚本

```
vi briup.sql
```

文件内容为：

```
create database if not exists briup;
use briup;
create table tea(id int,name string);
```

执行sql脚本

```
hive -f briup.sql
```

第四种方式：

hive经过发展，推出了第二代客户端beeline，但是beeline客户端不是直接访问metastore服务的，而是需要单独启动hiveserver2服务。

1. 在hdfs分布式集群中主节点core-site.xml文件中添加如下内容

```
<!--设置代理用户-->
<property>
  <name>hadoop.proxyuser.briup.hosts</name>
  <value>*</value>
</property>
<!--设置代理用户组-->
<property>
  <name>hadoop.proxyuser.briup.groups</name>
  <value>*</value>
</property>
```

2. 在hive的配置文件中设置远程服务端口及ip

```
<!--设置对外远程提供服务的端口，如jdbc连接；该服务需要启动hiveserver2服务，如果不需要远程连接hive，可以省略-->
<property>
  <name>hive.server2.thrift.port</name>
  <value>10000</value>
</property>
<!--设置对外远程提供服务，服务绑定的计算机-->
<property>
  <name>hive.server2.thrift.bind.host</name>
  <value>0.0.0.0</value>
</property>
```

3. 启动hiveserver2服务

```
hive --service hiveserver2 &
```

4. 开启beeline客户端

```
beeline
```

```
briup@master:~$ beeline
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/briup/software/apache-hive-3.1.2-bin/lib/log4j-slf4j-impl-2.10.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/briup/software/hadoop-3.1.3/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Beeline version 3.1.2 by Apache Hive
beeline>
```

5. 连接hive的远程服务,输入用户名，连接成功之后即可执行hive相关的命令操作

```
!connect jdbc:hive2://master:10000
```

```
beeline> !connect jdbc:hive2://master:10000
Connecting to jdbc:hive2://master:10000
Enter username for jdbc:hive2://master:10000: briup
Enter password for jdbc:hive2://master:10000:
Connected to: Apache Hive (version 3.1.2)
Driver: Hive JDBC (version 3.1.2)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://master:10000>
```

> beeline常规命令：
>
> control+l 清除屏幕
>
> !help 帮助菜单
>
> !quit 退出beeline终端
>
> !tables 列出所有的表

2.4.4 可视化工具

IntelliJ IDEA可以在Windows、Mac平台中通过JDBC连接HiveServer2的图形界面工具，该类工具专门针对SQL类软件进行开发优化，页面美观大方，操作简单，SQL语法智能提示补全，关键字高亮、查询结果智能显示，按键操作大于命令操作。要求IntelliJ IDEA是企业版。

1. 在hdfs分布式集群中主节点hdfs-site.xml文件中添加如下内容

```
<!--启动hdfs的web访问-->
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
```

2. 在hdfs分布式集群中主节点core-site.xml文件中添加如下内容

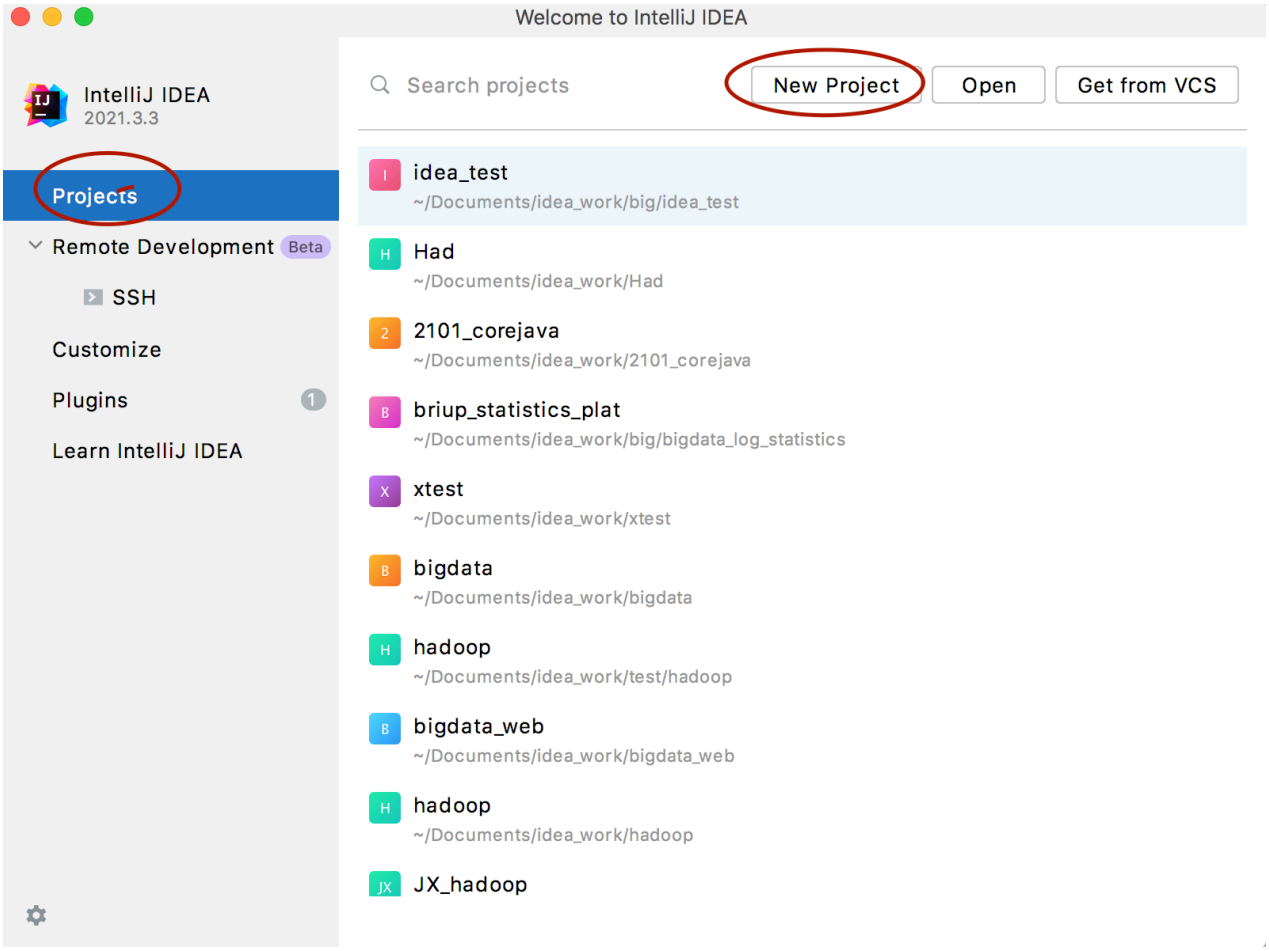
```
<!--设置代理用户-->
<property>
  <name>hadoop.proxyuser.briup.hosts</name>
  <value>*</value>
</property>
<!--设置代理用户组-->
<property>
  <name>hadoop.proxyuser.briup.groups</name>
  <value>*</value>
</property>
```

3. 重启hadoop集群，并开启hive服务

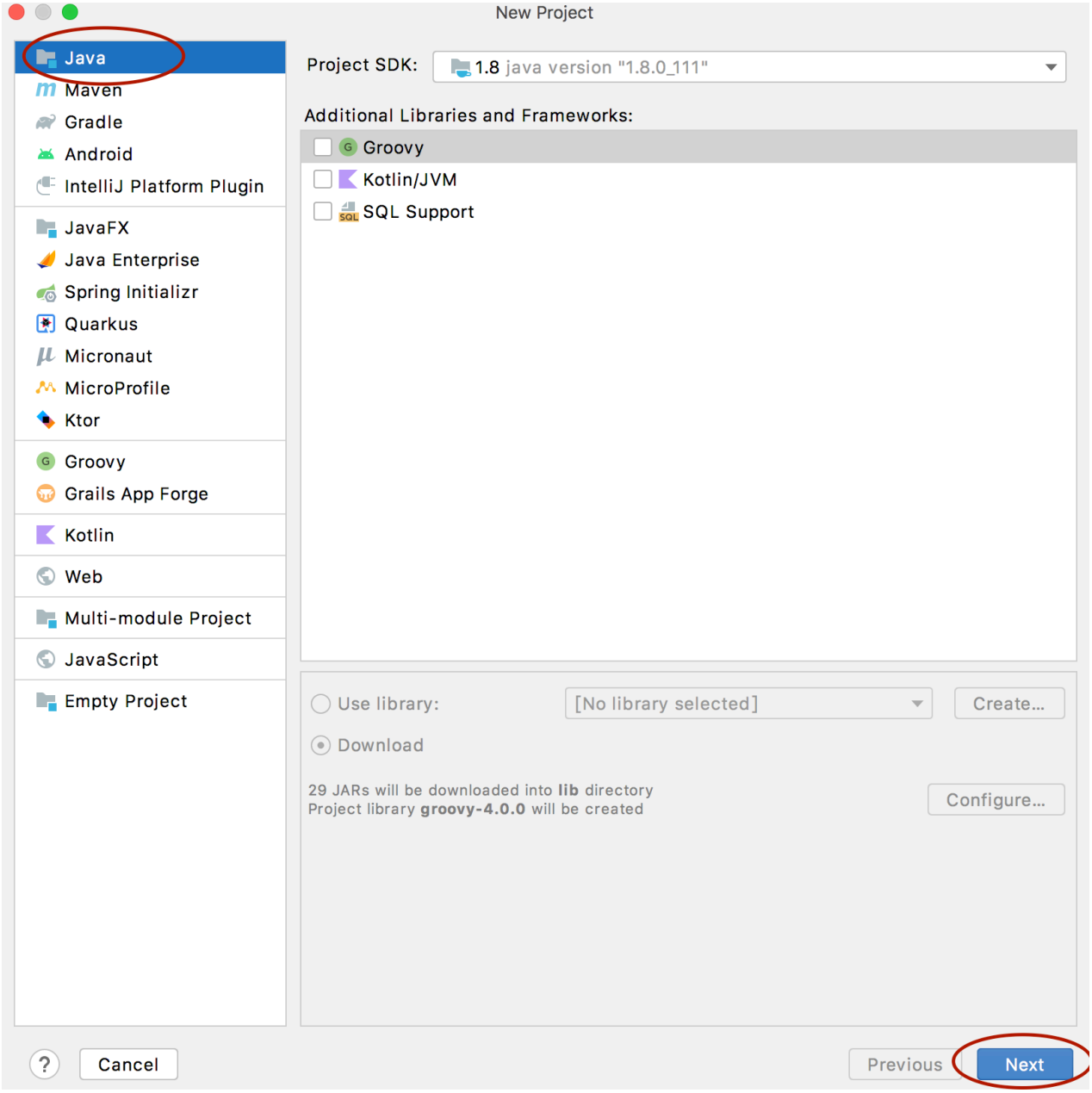
```
hive --service hiveserver2 &
```

> 服务开启默认端口10000

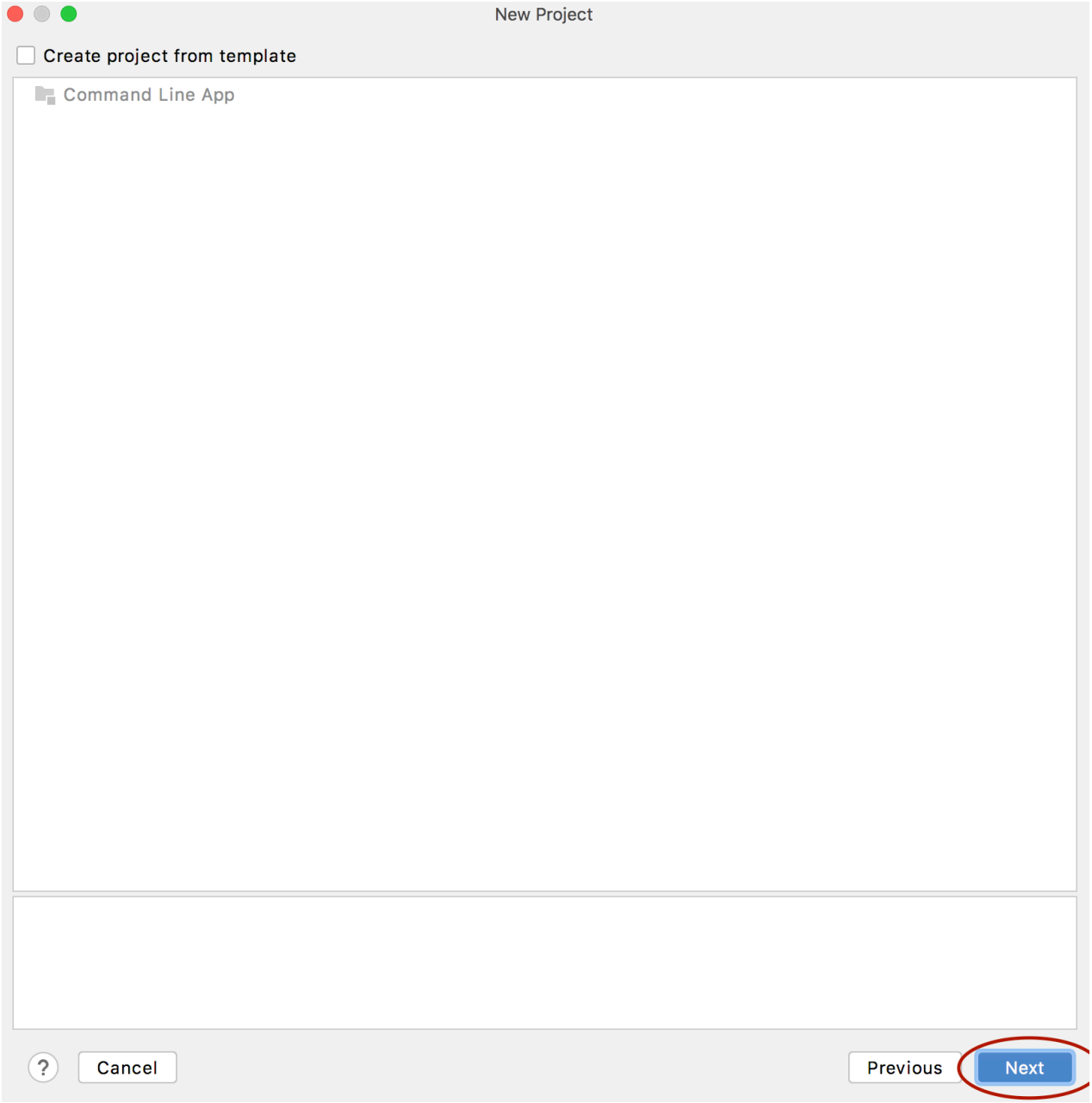
4. 构建工程，点击Projects,再点击New Project



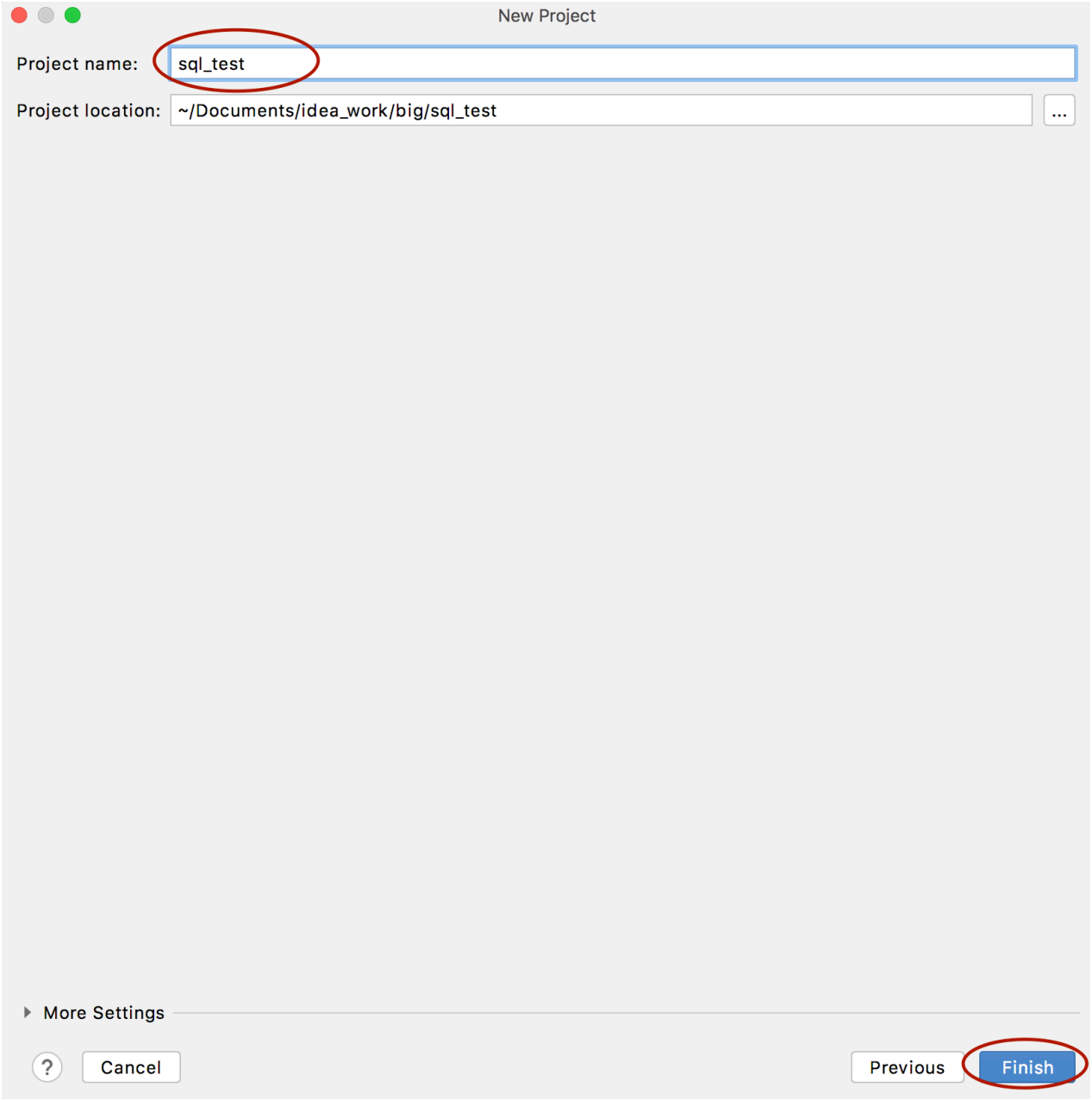
5. 点击Java，再点击Next


New Project
Java
Maven
Gradle
Android
IntelliJ Platform Plugin
JavaFX
Java Enterprise
Spring Initializr
Quarkus
Micronaut
MicroProfile
Ktor
Groovy
Grails App Forge
Kotlin
Web
Multi-module Project
JavaScript
Empty Project
Project SDK:
1.8 java version "1.8.0_111"
Additional Libraries and Frameworks:
Groovy
Kotlin/JVM
SQL Support
Use library:
[No library selected]
Create...
Download
29 JARs will be downloaded into lib directory
Project library groovy-4.0.0 will be created
Configure...
Cancel
Previous
Next

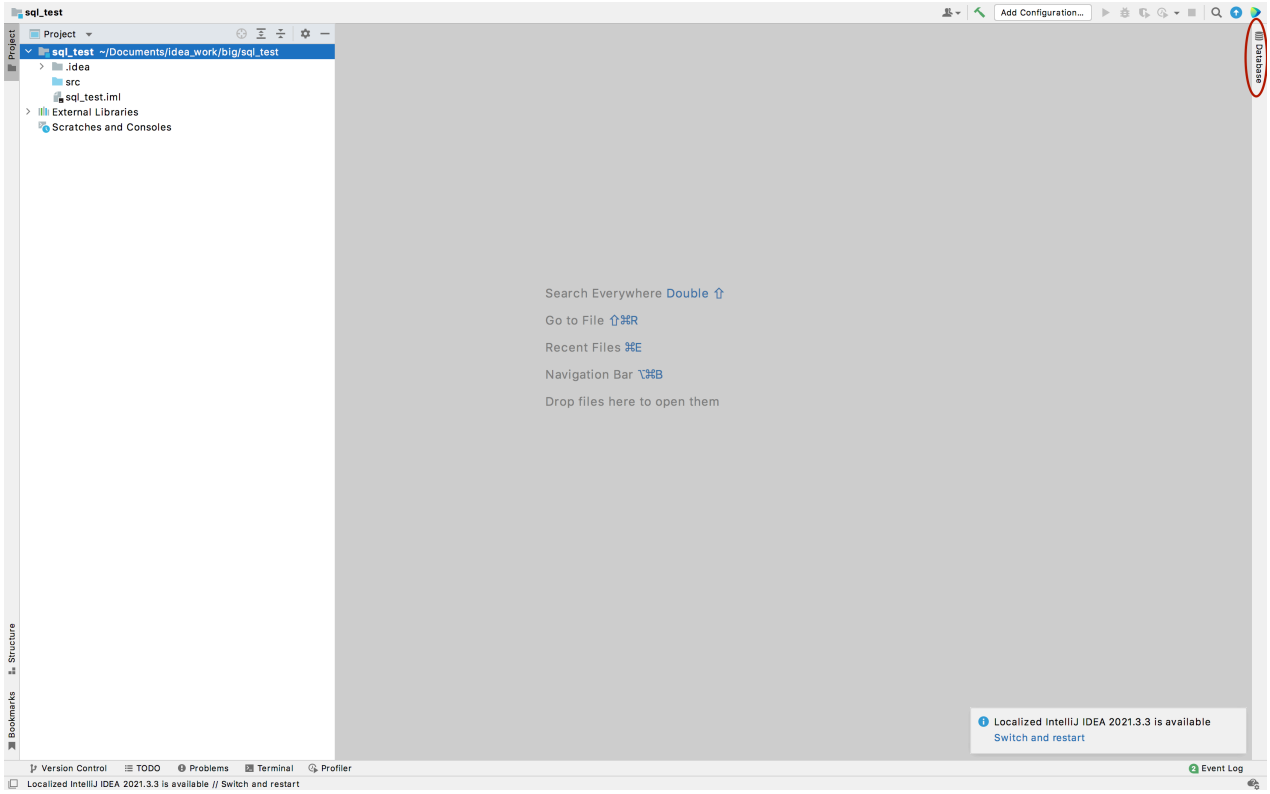
6. 点击next



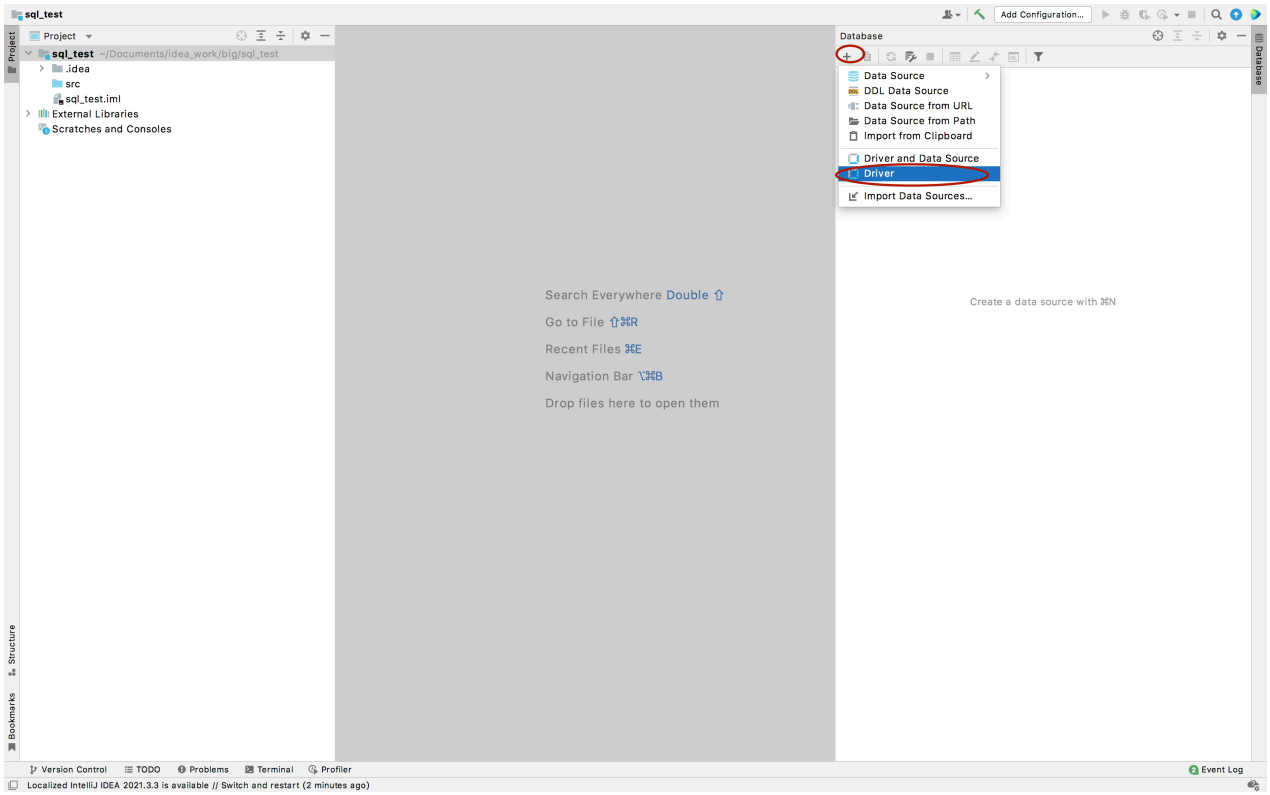
7. 填写项目名字，点击Finish



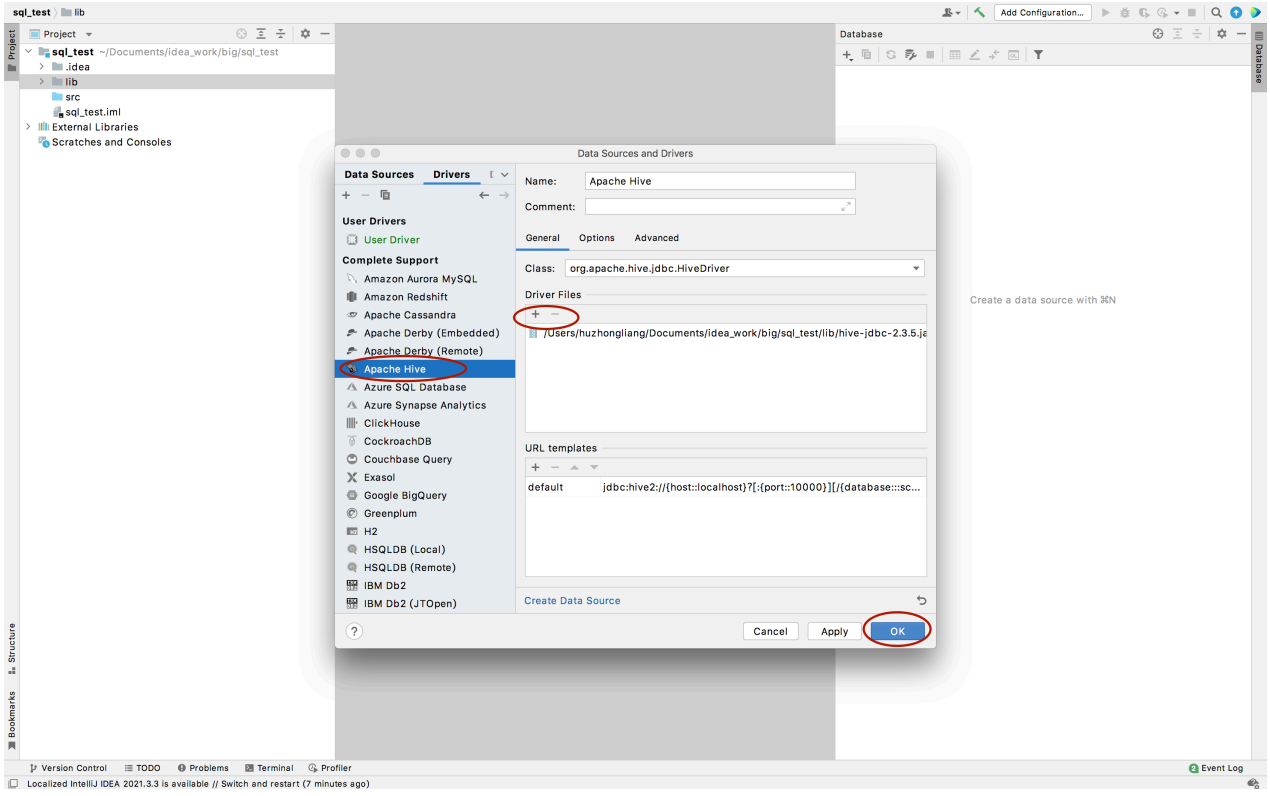
8. 选择右侧Database



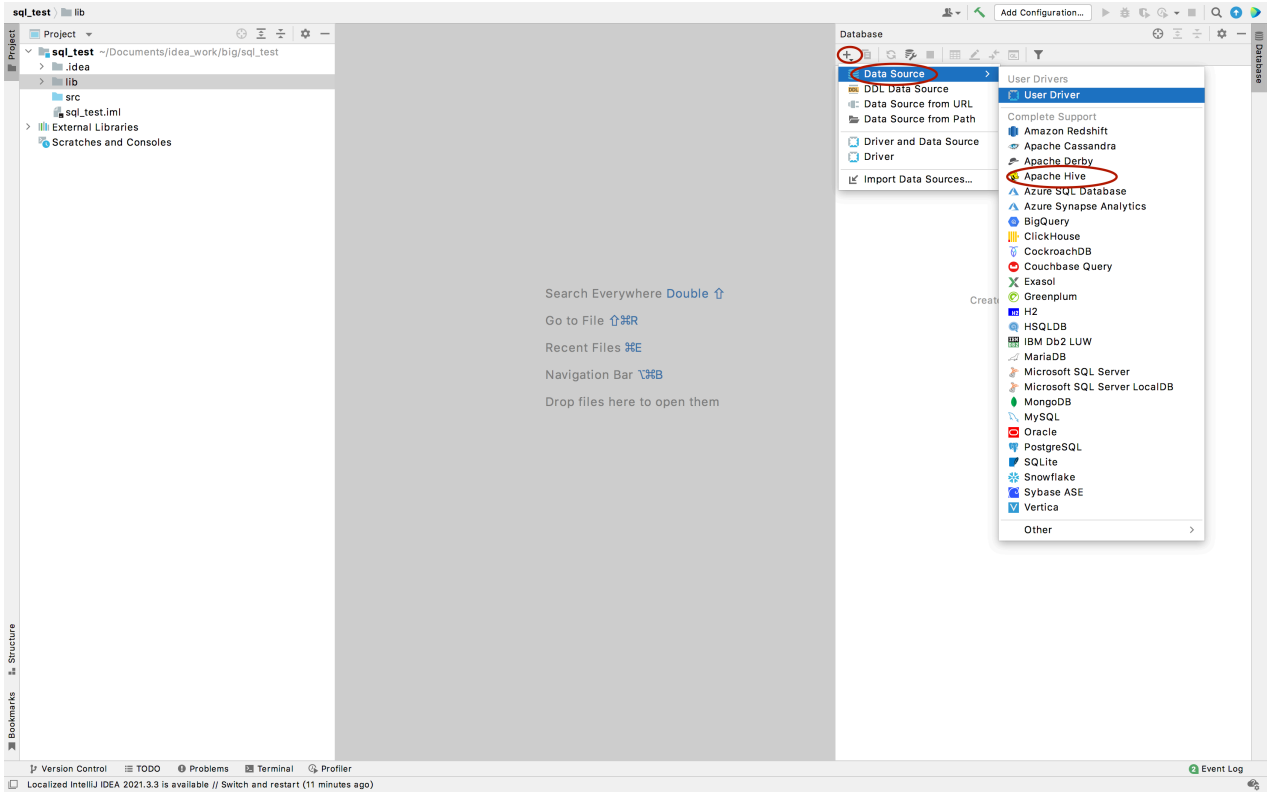
9. 点击+号，再点击Driver



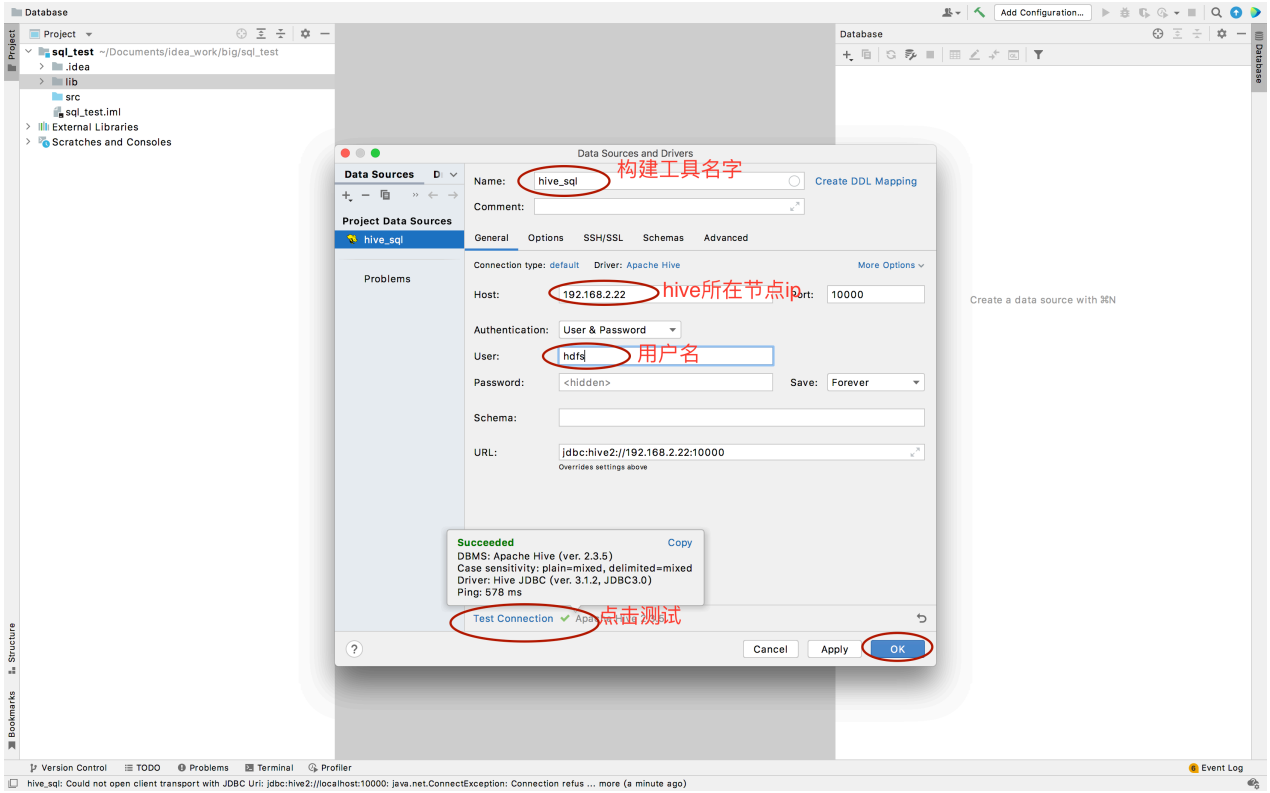
10. 左侧选择Apache Hive，选中默认的Hive的jar包点击-号移除，再点击+引入连接hive的驱动jar包。最后点击OK。



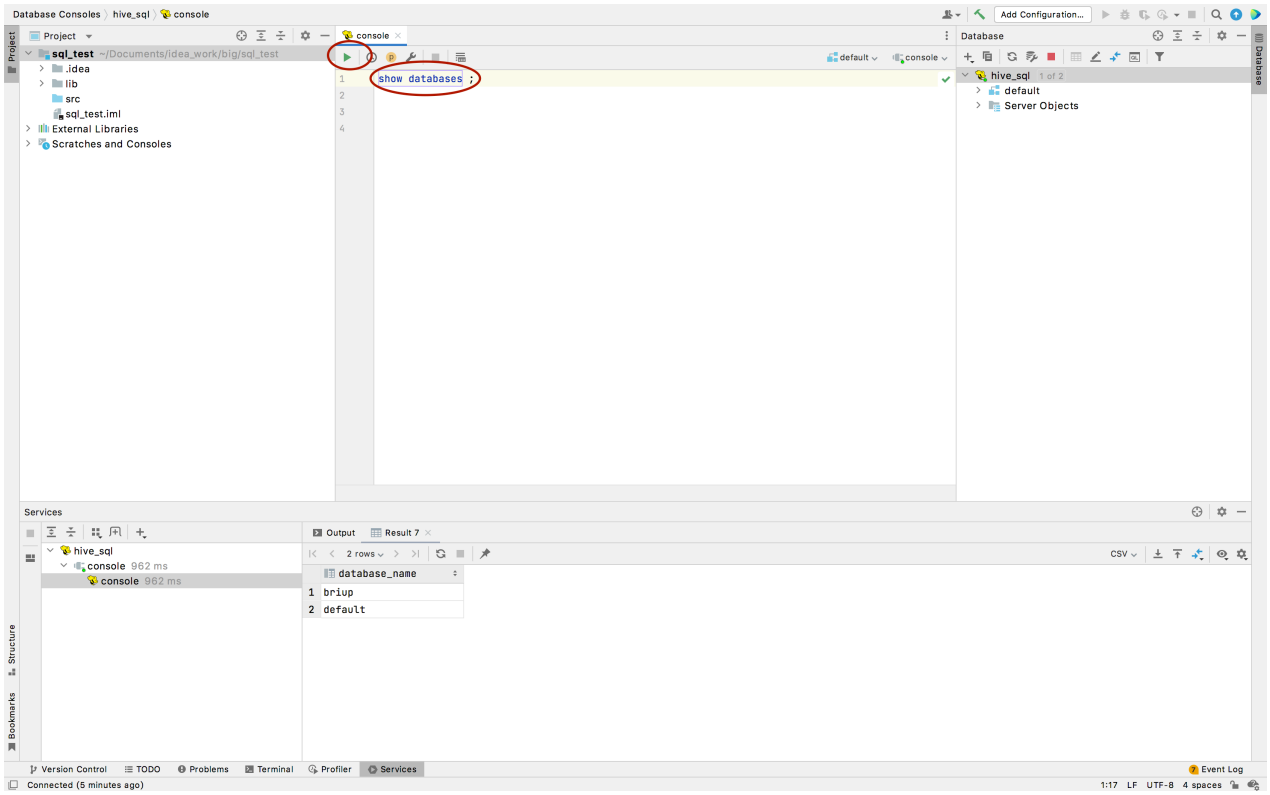
11. 点击+，再点击DataSource,最后点击Apache Hive.



12. 给sql工具起名字，在Host填写hive所在机器的Ip，填写用户名，点击Test Connection测试连接，连接成功直接点击OK。



13. 测试，在console界面编写sql语句，点击图中绿色的图标直接执行。



2.5 HiveQL

2.5.1 定义

HiveQL 是 Hive 查询语言，它不完全遵守任何一种 ANSI SQL 标准的修订版，但它与 MySQL 最接近，但还有显著的差异，Hive 不支持行级插入，更新和删除的操作，也不支持事务，但 Hive 增加了在 Hadoop 背景下的可以提供更高性能的扩展，以前个性化的扩展，还有一些外部程序。

2.5.2 数据类型

数据类型	说明	例子
BOOLEAN	true或false	true
TINYINT	1字节有符号整数	1Y
SMALLINT	2字节有符号整数	1S
INT	4字节有符号整数	1
BIGINT	8字节有符号整数	1L
FLOAT	4字节单精度浮点数	1.0
DOUBLE	8字节单精度浮点数	1.0
DEICIMAL	任意精度的带符号小数	1.0
STRING	变长字符串	"briup"
VARCHAR(n)	变长字符串	"briup"
CHAR(n)	固定长度字符串	"briup"
BINARY	字节数组	
TIMESTAMP	时间戳，毫秒值精度	188327493795
DATE	日期	'1992-2-2'
ARRAY	有序的同类型集合	array(1,2)
MAP	key-value,key必须为原始类型，value可以是任意类型	map(1,"t1",2,"t2")
STRUCT	字段类型，类型可以不一致	struct("1",1,2)

2.5.3 存储结构

Hive 中包含以下数据模型：

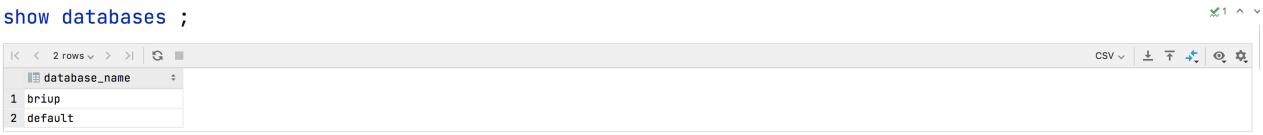
- database：在 HDFS 中表现为${hive.metastore.warehouse.dir}目录下一个文件夹
- table：在 HDFS 中表现所属 database 目录下一个文件夹
- external table：与 table 类似，不过其数据存放位置可以指定任意 HDFS 目录路径

- partition table：在 HDFS 中表现为 table 目录下的子目录
- bucket table：在 HDFS 中表现为同一个表目录或者分区目录下根据某个字段的值进行 hash 散列之后的多个文件
- view：与传统数据库类似，只读，基于基本表创建

展示所有的数据库

```
show databases;
```

```
show databases ;
```

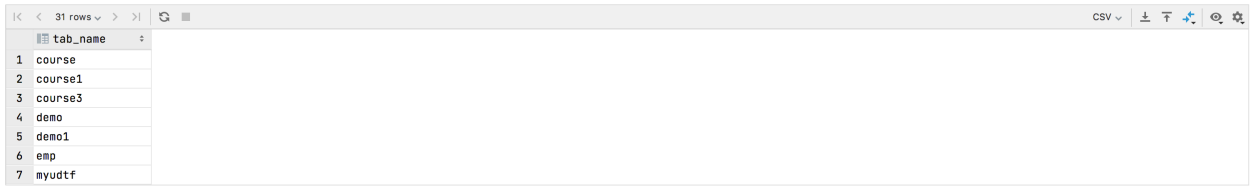


	database_name
1	briup
2	default

展示所有的表

```
show tables;
```

```
use briup;
show tables;
```



	tab_name
1	course
2	course1
3	course3
4	demo
5	demo1
6	emp
7	myudtf

3 Hive的基本操作

3.1 数据库

3.1.1 创建数据库

创建数据库briup

```
create database if not exists briup;
```

使用数据库

```
use briup
```

> 数据库默认创建之后存储的路径为：/user/hive/warehouse/briup.db
>
> hive的表存放位置模式是由hive-site.xml当中的一个属性指定的
>
> hive.metastore.warehouse.dir /user/hive/warehouse

3.1.2 数据库位置

创建数据库briup1，并指定位置为briup_test

```
create database  briup1 location '/briup_test';
```

> /briup_test指定是hdfs的存储目录

3.1.3 数据库键值

数据库可以有一些描述性的键值对信息，在创建时添加：

```
create database  briup2 with dbproperties ('owner'='briup', 'date'='20210101');
```

查看数据库的键值对信息：

```
describe database extended briup2;
```

修改数据库的键值对信息：

```
alter database briup2 set dbproperties ('owner'='my_briup');
```

3.1.4 数据库信息

查看briup2的详细信息：

```
desc database extended  briup2;
```

3.1.5 删除数据库

删除空的数据库，数据库中不能存在表

```
drop database briup2;
```

删除数据库的同时级联删除数据库下的表

```
drop database briup2 cascade;
```

3.2 数据库表

Hive中的表实际上是hdfs中的文件或者目录，所以在命令行操作完后，是可以在hdfs中看到对应的变化的.

3.2.1 创建表

语法：

```
create [external] table [if not exists] table_name (
col_name data_type [comment '字段描述信息'],
col_name data_type [comment '字段描述信息'])
[comment '表的描述信息']
[partitioned by (col_name data_type,...)]
[clustered by (col_name,col_name,...)]
[sorted by (col_name [asc|desc],...) into num_buckets buckets]
[row format row_format]
[storted as ....]
[location '指定表的路径']
```

- create table

  创建一个指定名字的表。如果相同名字的表已经存在，则抛出异常；用户可以用 IF NOT EXISTS 选项来忽略这个异常。

- external

  可以让用户创建一个外部表，在建表的同时指定一个指向实际数据的路径（LOCATION），Hive 创建内部表时，会将数据移动到数据仓库指向的路径；若创建外部表，仅记录数据所在的路径，不对数据的位置做任何改变。在删除表的时候，内部表的元数据和数据会被一起删除，而外部表只删除元数据，不删除数据。

- comment

  表示注释,默认不能使用中文

- partitioned by

  表示使用表分区,一个表可以拥有一个或者多个分区，每一个分区单独存在一个目录下 .

- clustered by 对于每一个表分文件， Hive可以进一步组织成桶，也就是说桶是更为细粒度的数据范围划分。Hive也是针对某一列进行桶的组织。

- sorted by

  指定排序字段和排序规则

- row format

  指定表文件字段分隔符

  ```
  row_format
    : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]
          [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
          [NULL DEFINED AS char]   -- (Note: Available in Hive 0.13 and later)
    | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)]
  ```

  row format delimited:

  1. fields terminated by '\' escaped by '\' 表中列与列之间指定分隔符，如分隔符为特殊符号由escaped by 指定。
  2. collection items terminated by ','表中列元素为map、array、struct时，元素与元素之间基于分隔符分割。
  3. map keys terminated by ':' 表中列为map，指定键值之间的分隔符。
  4. lines terminated by '\n' 表中数据落地到hdfs文件中，指定行与行之间的分隔符。默认分隔符

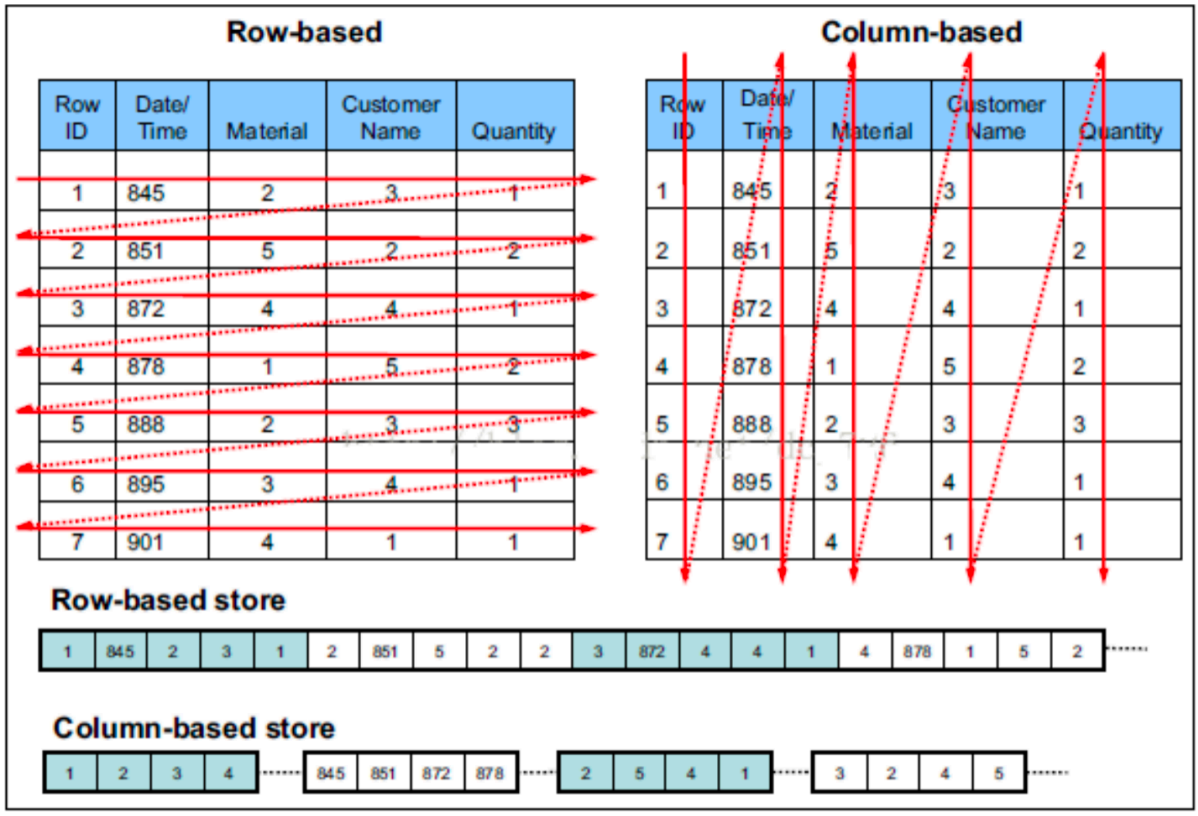
\n

5. NULL defined as 'no_val' 指定列中数据为NULL落地替换字符

row format SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value....)]:指定多分隔符。

- storted as

  指定表文件的存储格式, 常用格式:SEQUENCEFILE、 TEXTFILE、 RCFILE、ORC、PARQUET如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 storted as SEQUENCEFILE;RCFile文件格式是FaceBook开源的一种Hive的文件存储格式，首先将表分为几个行组，对每个行组内的数据进行按列存储，每一列的数据都是分开存储，正是先水平划分，再垂直划分的理念，ORC、PARQUET列式存储格式。



- location

  指定表文件的存储路径

> 默认hive数仓不显示列名，显示set hive.cli.print.header=true

3.2.2 内部表

内部表，就是一般的表，与数据库中的Table在概念上类似，每一个Table在Hive中都有一个相应的目录存数据，所有的内部表数据都保存在这个目录中，当表定义被删除的时候，表中的数据和元数据随之一并被删除。

1. 基本建表操作

```
use briup;
create table stu(id int,name string);
insert into stu(id,name) values(1,"lisi");
select id,name from stu;
```

2.创建表并指定数据落地的分隔符、存储路径和存储格式

```
create  table if not exists stu3(id int ,name string) row format delimited fields
terminated by ',' stored as textfile location '/stu3_test';
insert into stu3(id,name) values(1,"lisi");
select id,name from stu3;
```

> 1、row format delimited fields terminated by '':为数据落地的分隔符，按照‘’分割
>
> 2、location '/stu1_test'：表示存储路径
>
> 3、stored as textfile：表示存储的格式

3.基于表结构创建表

```
create table stu3 like stu2;
```

4.基于查询结果创建表

```
create table stu4 as select * from stu2;
```

> 复制的时候结构和内容一起复制

5.查看表的详细信息

```
desc formatted  stu2;
```

```
desc formatted  stu2;
```

29 rows

	col_name	data_type	comment
1	# col_name	data_type	comment
2		<null>	<null>
3	id	int	
4	name	string	
5		<null>	<null>
6	# Detailed Table Information	<null>	<null>
7	Database:	briup	<null>
8	Owner:	hdfs	<null>
9	CreateTime:	Mon Feb 28 11:13:42 CST 2022	<null>
10	LastAccessTime:	UNKNOWN	<null>
11	Retention:	0	<null>
12	Location:	hdfs://192.168.2.22:9000/stu_test1	<null>
13	Table Type:	MANAGED_TABLE	<null>
14	Table Parameters:	<null>	<null>
15		numFiles	3
16		totalSize	93
17		transient_lastDdlTime	1646021837
18		<null>	<null>
19	# Storage Information	<null>	<null>
20	SerDe Library:	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	<null>
21	InputFormat:	org.apache.hadoop.mapred.TextInputFormat	<null>
22	OutputFormat:	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat	<null>
23	Compressed:	No	<null>
24	Num Buckets:	-1	<null>
25	Bucket Columns:	[]	<null>
26	Sort Columns:	[]	<null>
27	Storage Desc Params:	<null>	<null>
28		field.delim	,
29		serialization.format	,

6.删除表

```
drop table stu4;
```

7.本地数据导入

表中数据可以使用insert语句插入，此时会发现提交了mr任务，效率极其低下，一般情况我们表中的数据是从文件中导入的，创建文件stu_data.txt

创建文件：

```
vi stu_data.txt
```

文件内容为：

```
1,tom
2,jake
3,lisi
4,rose
```

本地文件导入

```
load data local inpath '/home/hdfs/stu_data.txt' into table stu2;
```

本地文件导入覆盖

```
load data local inpath '/home/hdfs/stu_data.txt' overwrite into table stu2;
```

8.hdfs分布式文件导入

创建文件：

```
vi stu_data.txt
```

文件内容为：

```
1,tom
2,jake
3,lisi
4,rose
```

文件上传到hdfs分布式文件系统

```
hdfs dfs -put stu_data.txt /user/hdfs
```

hdfs分布式文件导入表

```
load data inpath '/user/hdfs/stu_data.txt' into table stu2
```

hdfs分布式文件导入表覆盖已有数据

```
load data inpath '/user/hdfs/stu_data.txt' overwrite into table stu2
```

3.2.3 外部表

外部表，数据存在与否和表的定义互不约束，仅仅只是表对hdfs上相应文件的一个引用，当删除表定义的时候，表中的数据依然存在。

1. 外部建表操作

```
use briup;
create external table tea(id int,name string);
insert into tea(id,name) values(1,"lisi");
select id,name from tea;
```

2.创建外部表并指定数据落地的分隔符、存储路径和存储格式

```
create external table if not exists tea3(id int ,name string) row format delimited
fields terminated by ',' stored as textfile location '/tea3_test';
insert into tea3(id,name) values(1,"lisi");
select id,name from tea3;
```

3.基于表结构创建外部表

```
create external table tea4 like tea3;
```

> 复制的时候结构和内容一起复制

4.查看外部表的详细信息

```
desc formatted  tea2;
```

```
desc formatted  tea2;
```

32 rows

	col_name	data_type	comment
1	# col_name	data_type	comment
2		<null>	<null>
3	id	int	
4	name	string	
5		<null>	<null>
6	# Detailed Table Information	<null>	<null>
7	Database:	briup	<null>
8	Owner:	hdfs	<null>
9	CreateTime:	Tue Mar 01 09:07:39 CST 2022	<null>
10	LastAccessTime:	UNKNOWN	<null>
11	Retention:	0	<null>
12	Location:	hdfs://192.168.2.22:9000/user/hive/warehouse/briup.db/tea2	<null>
13	Table Type:	EXTERNAL_TABLE	<null>
14	Table Parameters:	<null>	<null>
15		COLUMN_STATS_ACCURATE	{\"BASIC_STATS\":\"true\"}
16		EXTERNAL	TRUE
17		numFiles	0
18		numRows	0
19		rawDataSize	0
20		totalSize	0
21		transient_lastDdlTime	1646096859
22		<null>	<null>
23	# Storage Information	<null>	<null>
24	SerDe Library:	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	<null>
25	InputFormat:	org.apache.hadoop.mapred.TextInputFormat	<null>
26	OutputFormat:	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat	<null>
27	Compressed:	No	<null>
28	Num Buckets:	-1	<null>
29	Bucket Columns:	[]	<null>
30	Sort Columns:	[]	<null>
31	Storage Desc Params:	<null>	<null>
32		serialization.format	1

5.删除外部表

```
drop table tea2;
```

6.本地数据导入

表中数据可以使用insert语句插入，此时会发现提交了mr任务，效率极其低下，一般情况我们表中的数据是从文件中导入的，创建文件stu_data.txt

- 创建文件：
- 文件内容为：
- 本地文件导入
- 本地文件导入覆盖已有数据

> 操作同内部表，参考上面(stu_data.txt数据同内部表)

7.hdfs分布式文件导入

- 创建文件：
- 文件内容为：
- 文件上传到hdfs分布式文件系统
- hdfs分布式文件导入表
- hdfs分布式文件导入数据覆盖原有数据

> 操作同内部表，参考上面(stu_data.txt数据同内部表)

3.2.4 分区表

根据业务编码、日期、其他类型等维度创建分区表，比如一个重庆市的9个区域各自一个分区，如果要查某一个区域的数据，只需要访问一个分区的数据，而不需要从全量数据中进行筛选。分区底层实现逻辑为：在一个表对应的目录下，一个分区对应一个目录

单表数据量巨大，而且查询又经常限定某一个类别，那么可以将表按照该类别进行分区，以提高数据查询效率，减少资源开销

1. 分区建表操作

```
use briup;
create table score(id int,name string) partitioned by (month string);
insert into table score partition(month='01') values(1,"lisi");
select id,name from score;
select id,name from score where month='01';
```

2.创建分区表并指定数据落地的分隔符、存储路径和压缩数据

```
create  table if not exists score3(id int ,name string) partitioned by (month string)
row format delimited fields terminated by ',' stored as textfile location
'/score3_test';
insert into score3 partition(month='01') values(1,"lisi");
insert overwrite table score3 partition(month = '01') select id,name from score2;
select id,name from score3;
select id,name from score3 where month='01';
```

> partitioned by (month string)：表示根据month分区

3.基于分区表结构创建表

```
create table score4 like score2;
```

4.基于查询结果创建表

```
create table score5 as select * from score2;
```

> 复制的时候结构和内容一起复制

5.查看分区表的详细信息

```
desc formatted  score2;
```

```
desc formatted  score2;
```

37 rows

	col_name	data_type	comment
2		<null>	<null>
3	id	int	
4	name	string	
5		<null>	<null>
6	# Partition Information	<null>	<null>
7	# col_name	data_type	comment
8		<null>	<null>
9	month	string	
10		<null>	<null>
11	# Detailed Table Information	<null>	<null>
12	Database:	briup	<null>
13	Owner:	hdfs	<null>
14	CreateTime:	Fri Mar 04 10:43:32 CST 2022	<null>
15	LastAccessTime:	UNKNOWN	<null>
16	Retention:	0	<null>
17	Location:	hdfs://192.168.2.22:9000/score_test3	<null>
18	Table Type:	MANAGED_TABLE	<null>
19	Table Parameters:	<null>	<null>
20		numFiles	3
21		numPartitions	4
22		numRows	2
23		rawDataSize	12
24		totalSize	100
25		transient_lastDdlTime	1646361812
26		<null>	<null>
27	# Storage Information	<null>	<null>
28	SerDe Library:	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	<null>
29	InputFormat:	org.apache.hadoop.mapred.TextInputFormat	<null>
30	OutputFormat:	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat	<null>
31	Compressed:	No	<null>
32	Num Buckets:	-1	<null>
33	Bucket Columns:	[]	<null>
34	Sort Columns:	[]	<null>

6.删除分区表

```
drop table score4;
```

7.查看分区

```
show  partitions  score;
```

8.添加一个分区

```
alter table score add partition(month='02');
```

9.删除一个分区

```
alter table score drop partition(month = '02');
```

10.多分区表联合查询

```
select id,name from score where month = '01' union all select * from score where month
= '02';
```

11.本地数据导入

表中数据可以使用insert语句插入，此时会发现提交了mr任务，效率极其低下，一般情况我们表中的数据是从文件中导入的，创建文件stu_data.txt

创建文件：

```
vi stu_data.txt
```

文件内容为：

```
1,tom
2,jake
3,lisi
4,rose
```

本地文件导入

```
load data local inpath '/home/hdfs/stu_data.txt' into table score2 partition
(month='01');
```

本地文件导入覆盖

```
load data local inpath '/home/hdfs/stu_data.txt' overwrite into table score2 partition
(month='01');
```

12.hdfs分布式文件导入

创建文件：

```
vi stu_data.txt
```

文件内容为：

```
1,tom
2,jake
3,lisi
4,rose
```

文件上传到hdfs分布式文件系统

```
hdfs dfs -put stu_data.txt /user/hdfs
```

hdfs分布式文件导入表

```
load data inpath '/user/hdfs/stu_data.txt' into table score2 partition (month='01');
```

hdfs分布式文件导入表覆盖已有数据

```
load data inpath '/user/hdfs/stu_data.txt' overwrite into table score2 partition
(month='01');
```

13.hdfs分布式文件导入多个分区表中

创建文件：

```
vi stu_data.txt
```

文件内容为：

```
1,tom
2,jake
3,lisi
4,rose
```

文件上传到hdfs分布式文件系统

```
hdfs dfs -put stu_data.txt /user/hdfs
```

hdfs分布式文件导入表

```
load data inpath '/user/hdfs/stu_data.txt' into table score6 partition
(month='01',year='2021');
```

hdfs分布式文件导入表覆盖已有数据

```
load data inpath '/user/hdfs/stu_data.txt' overwrite into table score6 partition
(month='01',year='2021');
```

3.2.5 桶表

分桶，就是将数据按照指定的字段进行划分到多个文件当中去,分桶就是MapReduce中的分区.

将大表进行哈希散列抽样存储，方便做和代码验证。比如将表分成10分，每次只拿其中的十分之一来使用，可以快速的得到结果

- 分桶底层实现逻辑：

  在表对应的目录下，将源文件拆分成N个小文件

- 使用场景：

  对于一个庞大的数据集我们经常需要拿出来一小部分作为样例，然后在样例上验证我们的查询，优化我们的程序,利用分桶表可以实现数据的抽样

开启 Hive 的分桶功能

```
set hive.enforce.bucketing=true;
```

设置 Reduce 个数

```
set mapreduce.job.reduces=3;
```

1. 基本建表操作

```
use briup;
create table course(id int,name string) clustered by(id) into 3 buckets;
select id,name from course;
```

2.创建表并指定数据落地的分隔符、存储路径和压缩数据

```
create table course1(
id int,
name string,
age int
)clustered by(id) into 3 buckets row format delimited fields terminated by ',' stored
as textfile location '/course1';
select id,name from course3;
```

3.基于表结构创建表

```
create table course4 like course2;
```

4.基于查询结果创建表

```
create table course5 as select * from course2;
```

> 复制的时候结构和内容一起复制

5.查看表的详细信息

```
desc formatted  course2;
```

```
desc formatted  course;
```

31 rows

	col_name	data_type	comment
1	# col_name	data_type	comment
2		<null>	<null>
3	id	int	
4	name	string	
5		<null>	<null>
6	# Detailed Table Information	<null>	<null>
7	Database:	briup	<null>
8	Owner:	hdfs	<null>
9	CreateTime:	Fri Mar 04 11:41:42 CST 2022	<null>
10	LastAccessTime:	UNKNOWN	<null>
11	Retention:	0	<null>
12	Location:	hdfs://192.168.2.22:9000/user/hive/warehouse/briup.db/course	<null>
13	Table Type:	MANAGED_TABLE	<null>
14	Table Parameters:	<null>	<null>
15		COLUMN_STATS_ACCURATE	{\"BASIC_STATS\":\"true\"}
16		numFiles	0
17		numRows	0
18		rawDataSize	0
19		totalSize	0
20		transient_lastDdlTime	1646365302
21		<null>	<null>
22	# Storage Information	<null>	<null>
23	SerDe Library:	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	<null>
24	InputFormat:	org.apache.hadoop.mapred.TextInputFormat	<null>
25	OutputFormat:	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat	<null>
26	Compressed:	No	<null>
27	Num Buckets:	3	<null>
28	Bucket Columns:	[id]	<null>
29	Sort Columns:	[]	<null>
30	Storage Desc Params:	<null>	<null>
31		serialization.format	1

6.删除表

```
drop table course4;
```

9. 本地数据导入

表中数据可以使用insert语句插入，此时会发现提交了mr任务，效率极其低下，一般情况我们表中的数据是从文件中导入的，创建文件stu_data.txt

- 创建文件：
- 文件内容为：
- 本地文件导入
- 本地文件导入覆盖

> 操作同内部表，参考上面(stu_data.txt数据同内部表)

通过insert into给桶表中加载数据(追加)

```
insert into table course select * from tea2 cluster by(id);
或
insert into table course select * from tea2;
```

通过insert overwrite给桶表中加载数据(覆盖)

```
insert overwrite table course select * from tea2 cluster by(id);
或
insert overwrite table course select * from tea2;
```

10. hdfs分布式文件导入

- 创建文件：
- 文件内容为：
- 文件上传到hdfs分布式文件系统
- hdfs分布式文件导入表
- hdfs分布式文件导入表覆盖已有数据

> 操作同内部表，参考上面(stu_data.txt数据同内部表)

11. 查询第二桶数据

```
select id,name from course tablesample(bucket 2 out of 3 on id);
```

> tablesample (bucket x out of y on column_name)
>
> x表示抽取的是第几个桶
>
> y表示划分为多少桶,可以是建表桶的倍数
>
> column_name 分桶的列

12. 基于随机抽取第二桶数据

```
select id,name from course tablesample(bucket 2 out of 3 on rand());
```

3.3 表结构

1. 表重命名

```
alter  table  tea1  rename  to  teacher;
```

2. 查询表结构

```
desc teacher;
```

3. 添加列

```
alter table teacher add columns (age int);
```

4. 更新列

```
alter table teacher change column name username  string;
```

3.4 集合类型

3.4.1 数组类型

1. 第一种用法：

基本建表操作

```
use briup;
create table tab_array (id int,arr array<string>)
row format delimited
fields terminated by '#'
collection items terminated by ',';
```

> 数组类型Array不支持insert操作

创建文件array.txt

```
vi array.txt
```

内容如下:

```
1#java,scala,python
2#hadoop,hive,hbase
3#spring,springMVC,Mybatis
4#Html,css,JavaScript,jquery
```

本地导入文件到表中

```
load data local inpath '/home/hdfs/array.txt' into table tab_array;
```

hdfs分布式文件导入

- 文件上传到hdfs集群

```
hdfs dfs -put array.txt .
```

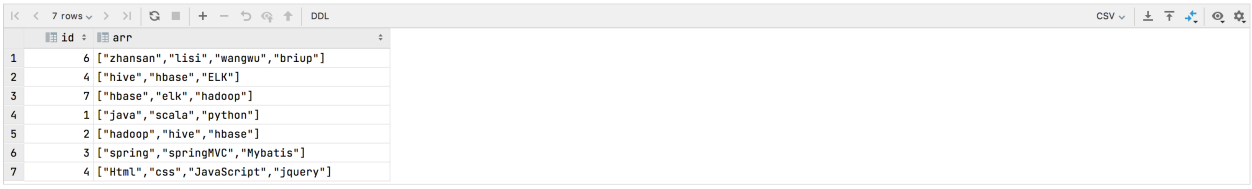
- hdfs文件导入到表中

```
load data inpath '/user/hdfs/array.txt' into table tab_array;
```

查询全部数据

```
select id,arr from tab_array;
```

```
select * from tab_array;
```

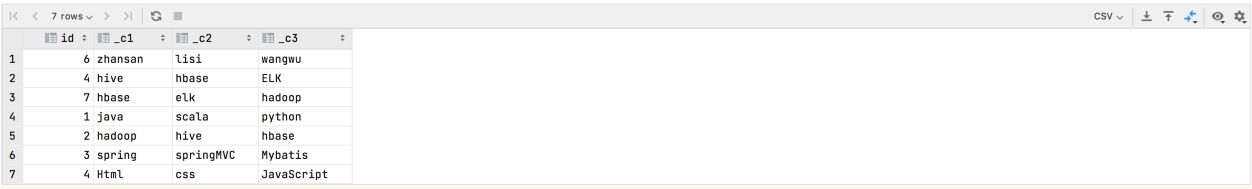


	id	arr
1	6	["zhansan","lisi","wangwu","briup"]
2	4	["hive","hbase","ELK"]
3	7	["hbase","elk","hadoop"]
4	1	["java","scala","python"]
5	2	["hadoop","hive","hbase"]
6	3	["spring","springMVC","Mybatis"]
7	4	["Html","css","JavaScript","jquery"]

查询数组中各个元素

```
select id,arr[0],arr[1],arr[2] from tab_array;
```

```
select id,arr[0],arr[1],arr[2] from tab_array;
```



	id	_c1	_c2	_c3
1	6	zhansan	lisi	wangwu
2	4	hive	hbase	ELK
3	7	hbase	elk	hadoop
4	1	java	scala	python
5	2	hadoop	hive	hbase
6	3	spring	springMVC	Mybatis
7	4	Html	css	JavaScript

2. 第二种用法

创建表

```
create table stu(id int,name string);
insert into stu(id,name) values(1,"zhao,qian,sui,li");
```

数据传入tab_array表

```
insert into tab_array select id,split(name,",") from stu;
```

3. 第三种方法

```
insert into tab_array select 4,array('hive','hbase','ELK');
```

4. 第四种方法

```
insert into tab_array values(1,array('xml','hadoop','hbase'));
```

3.4.2 映射类型

基本建表操作

```
create table tab_map (name string,info map<string,string>)
row format delimited
fields terminated by '#'
collection items terminated by ','
map keys terminated by ':';
```

创建文件map.txt

```
vi map.txt
```

内容如下:

```
lisi#id:10,age:30,sex:male
zhangsan#id:11,age:33,sex:female,hoby:basketball
lili#id:12,addr:kunshan
```

本地导入文件到表中

```
load data local inpath '/home/hdfs/map.txt' into table tab_map;
```

hdfs分布式文件导入

- 文件上传到hdfs集群

```
hdfs dfs -put map.txt .
```

- hdfs文件导入到表中

```
load data inpath '/user/hdfs/map.txt' into table tab_map;
```

查询所有数据

```
select * from tab_map;
```

```
select * from tab_map;
```

5 rows DDL CSV

	name	info
1	lili	{"id":"1","name":"lisi"}
2	briup	{"id":"21","phone":"15098278990"}
3	lisi	{"id":"10","age":"30","sex":"male"}
4	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}
5	lili	{"id":"12","addr":"kunshan"}

基于键查询数据

```
select name,info['name'],info['age'] from tab_map;
```

```
select name,info['name'],info['age'] from tab_map;
```

5 rows CSV

	name	_c1	_c2
1	lili	lisi	<null>
2	briup	<null>	<null>
3	lisi	<null>	30
4	zhangsan	<null>	33
5	lili	<null>	<null>

插入数据

```
insert into tab_map select 'lili',map('id','1','name','lisi');
或
insert into tab_map select 'briup',str_to_map('id:21,phone:15098278990');
```

3.4.3 字段类型

基本建表操作

```
create table tab_struct(name string,info  struct<age:int,tel:string,salary:double>)
row format delimited
fields terminated by '|'
collection items terminated by '@';
```

> 数组类型struct不支持insert操作

创建文件struct.txt

```
vi struct.txt
```

内容如下:

```
zhansan|20@15066662334@3400
lisi|22@16768889223@4500
briup|30@18922223446@10000
```

本地导入文件到表中

```
load data local inpath '/home/hdfs/struct.txt' into table tab_struct;
```

hdfs分布式文件导入

- 文件上传到hdfs集群

```
hdfs dfs -put struct.txt .
```

- hdfs文件导入到表中

```
load data inpath '/user/hdfs/struct.txt' into table tab_struct;
```

查询所有数据

```
select * from tab_struct;
```

```
select * from tab_struct;
```

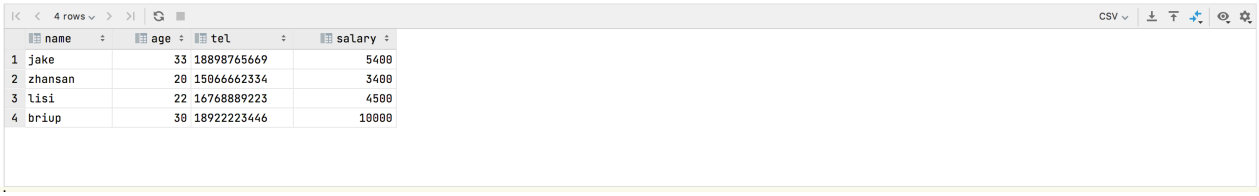
4 rows | DDL | CSV

	name	info
1	jake	{"age":33,"tel":"18898765669","salary":5400.0}
2	zhansan	{"age":20,"tel":"15066662334","salary":3400.0}
3	lisi	{"age":22,"tel":"16768889223","salary":4500.0}
4	briup	{"age":30,"tel":"18922223446","salary":10000.0}

基于键查询数据

```
select name,info.age,info.tel,info.salary from tab_struct;
```

```
select name,info.age,info.tel,info.salary from tab_struct;
```



	name	age	tel	salary
1	jake	33	18898765669	5400
2	zhansan	20	15066662334	3400
3	lisi	22	16768889223	4500
4	briup	30	18922223446	10000

插入数据

```
insert into tab_struct select
'jake',named_struct('age',33,'tel','18898765669','salary',cast(5400 as double));
或
insert into tab_struct
values('zhangliu',named_struct('age',30,'tel','198765676789','salary',cast(5400 as
double')));
```

3.5 特殊分隔符

Hive中默认的序列化类是org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe，其中支持单子节分隔符，默认分隔符“\001”，根据不同文件的不同分隔符，可以通过在创建表时使用row format delimited 指定文件的分隔符。

案例1: 创建表，列与列之间的分隔符为\。

```
create table demo1(
id int,
name string,
salary double
)
row format delimited fields  terminated by '\\' escaped by '\\' ;
```

创建文件

```
vi demo.txt
```

内容如下:

```
1\lisi\3000
2\zhansan\3500
3\briup\2700
4\zhansan\2900
5\wangwu\3200
```

加载数据

```
load data local inpath '/home/hdfs/demo.txt'  into table demo1;
```

查询表结果:

```
select * from demo1;
```

```
select * from demo1;
```

5 rows | DDL | CSV

	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900
5	5	<null>	<null>

表对应hdfs文件内容:

```
hdfs dfs -cat /user/hive/warehouse/briup.db/demo1/demo.txt
```

```
hdfs@master:~$ hdfs dfs -cat /user/hive/warehouse/briup.db/demo1/demo.txt
1\lisi\3000
2\zhansan\3500
3\briup\2700
4\zhansan\2900
5
```

案例2:创建表，基于||分隔表中列与列

```
create table demo2(
id int,
name string,
salary double
)
row format delimited fields  terminated by '||';
```

创建文件

```
vi demo2.txt
```

内容如下:

```
1||lisi||3000
2||zhansan||3500
3||briup||2700
4||zhansan||2900
```

加载数据

```
load data local inpath '/home/hdfs/demo2.txt'  into table demo2;
```

查询表结果:

```
select * from demo2;
```

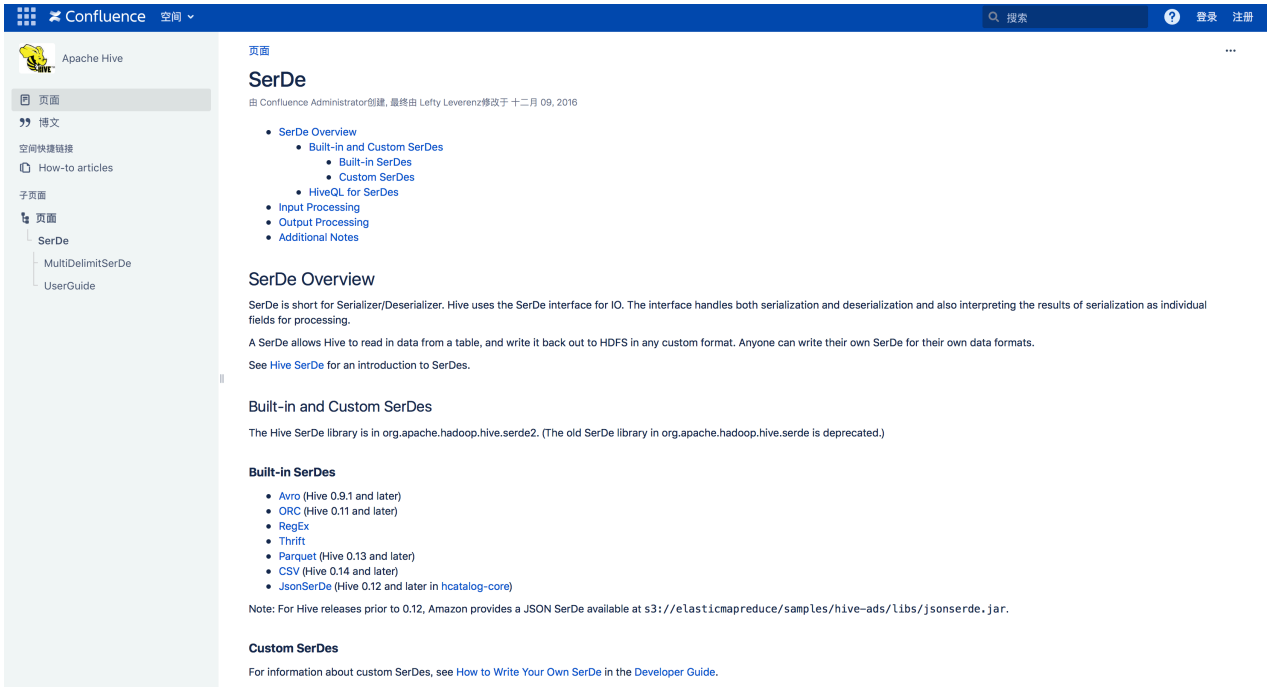
```
select * from demo2;
```

4 rows | DDL | CSV

	id	name	salary
1	1		<null>
2	2		<null>
3	3		<null>
4	4		<null>

> Hive默认序列化类是LazySimpleSerDe，其只支持使用单字节分隔符来加载文本数据

hive中本身支持多种分隔符方式：



网址： https://cwiki.apache.org/confluence/display/Hive/SerDe

解决方案1：使用MultiDelimitSerDe的方法来实现

建表：

```
drop table demo2;
create table demo2(
id int,
name string,
salary double
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.MultiDelimitSerDe' WITH
SERDEPROPERTIES ("field.delim"="||")
STORED AS TEXTFILE;
```

加载数据

```
load data local inpath '/home/hdfs/demo2.txt'  into table demo2;
```

查询表结果:

```
select * from demo2;
```

```
select * from demo2;
```

4 rows | DDL | CSV

	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900

解决方案2：使用RegexSerDe的方法实现：

建表：

```
drop table demo2;
create table demo2(
id string,
name string,
salary string
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe' WITH
SERDEPROPERTIES ("input.regex" = "([0-9]*)\\|\\|(.*)\\|\\|([0-9]*\\.?[0-9]*)")
STORED AS TEXTFILE;
```

> 使用正则匹配的时候列字段类型为string

加载数据

```
load data local inpath '/home/hdfs/demo2.txt'  into table demo2;
```

查询表结果:

```
select * from demo2;
```

```
select * from demo2;
```

4 rows | DDL | CSV

	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900

解决方案3 :自定义InputFormat

Hive中允许使用自定义InputFormat解决分隔符的问题，通过自定义InputFormat解析逻辑实现读取每一行数据。

构建maven项目，引入依赖

```
<dependencies>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.0.3</version>
  </dependency>
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>3.0.3</version>
  </dependency>
</dependencies>
```

构建自定义读取器

```
package com.briup.Separator;


import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
```

```
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.Seekable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.*;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.LineRecordReader;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapreduce.lib.input.CompressedSplitLineReader;
import org.apache.hadoop.mapreduce.lib.input.SplitLineReader;
import org.apache.hadoop.mapreduce.lib.input.UncompressedSplitLineReader;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import java.io.IOException;
import java.io.InputStream;

public class UserRecordReader implements RecordReader<LongWritable,Text> {
    private static final Logger LOG =
LoggerFactory.getLogger(LineRecordReader.class.getName());
    private CompressionCodecFactory compressionCodecs;
    private long start;
    private long pos;
    private long end;
    private SplitLineReader in;
    private FSDataInputStream fileIn;
    private final Seekable filePosition;
    int maxLineLength;
    private CompressionCodec codec;
    private Decompressor decompressor;

    public UserRecordReader(Configuration job, FileSplit split) throws IOException {
        this(job, split, (byte[])null);
    }

    public UserRecordReader(Configuration job, FileSplit split, byte[]
recordDelimiter) throws IOException {
        this.compressionCodecs = null;
        this.maxLineLength =
job.getInt("mapreduce.input.linerecordreader.line.maxlength", 2147483647);
        this.start = split.getStart();
        this.end = this.start + split.getLength();
        Path file = split.getPath();
        this.compressionCodecs = new CompressionCodecFactory(job);
        this.codec = this.compressionCodecs.getCodec(file);
        FileSystem fs = file.getFileSystem(job);
        this.fileIn = fs.open(file);
        if (this.isCompressedInput()) {
            this.decompressor = CodecPool.getDecompressor(this.codec);
            if (this.codec instanceof SplittableCompressionCodec) {
```

```
                SplitCompressionInputStream cIn =
((SplittableCompressionCodec)this.codec).createInputStream(this.fileIn,
this.decompressor, this.start, this.end,
SplittableCompressionCodec.READ_MODE.BYBLOCK);
                this.in = new CompressedSplitLineReader(cIn, job, recordDelimiter);
                this.start = cIn.getAdjustedStart();
                this.end = cIn.getAdjustedEnd();
                this.filePosition = cIn;
            } else {
                if (this.start != 0L) {
                    throw new IOException("Cannot seek in " +
this.codec.getClass().getSimpleName() + " compressed stream");
                }

                this.in = new
SplitLineReader(this.codec.createInputStream(this.fileIn, this.decompressor), job,
recordDelimiter);
                this.filePosition = this.fileIn;
            }
        } else {
            this.fileIn.seek(this.start);
            this.in = new UncompressedSplitLineReader(this.fileIn, job,
recordDelimiter, split.getLength());
            this.filePosition = this.fileIn;
        }

        if (this.start != 0L) {
            this.start += (long)this.in.readLine(new Text(), 0,
this.maxBytesToConsume(this.start));
        }

        this.pos = this.start;
    }

    public UserRecordReader(InputStream in, long offset, long endOffset, int
maxLineLength) {
        this(in, offset, endOffset, maxLineLength, (byte[])null);
    }

    public UserRecordReader(InputStream in, long offset, long endOffset, int
maxLineLength, byte[] recordDelimiter) {
        this.compressionCodecs = null;
        this.maxLineLength = maxLineLength;
        this.in = new SplitLineReader(in, recordDelimiter);
        this.start = offset;
        this.pos = offset;
        this.end = endOffset;
        this.filePosition = null;
    }

    public UserRecordReader(InputStream in, long offset, long endOffset, Configuration
job) throws IOException {
```

```
        this(in, offset, endOffset, job, (byte[])null);
    }

    public UserRecordReader(InputStream in, long offset, long endOffset, Configuration
job, byte[] recordDelimiter) throws IOException {
        this.compressionCodecs = null;
        this.maxLineLength =
job.getInt("mapreduce.input.linerecordreader.line.maxlength", 2147483647);
        this.in = new SplitLineReader(in, job, recordDelimiter);
        this.start = offset;
        this.pos = offset;
        this.end = endOffset;
        this.filePosition = null;
    }

    public LongWritable createKey() {
        return new LongWritable();
    }

    public Text createValue() {
        return new Text();
    }

    private boolean isCompressedInput() {
        return this.codec != null;
    }

    private int maxBytesToConsume(long pos) {
        return this.isCompressedInput() ? 2147483647 :
(int)Math.max(Math.min(2147483647L, this.end - pos), (long)this.maxLineLength);
    }

    private long getFilePosition() throws IOException {
        long retVal;
        if (this.isCompressedInput() && null != this.filePosition) {
            retVal = this.filePosition.getPos();
        } else {
            retVal = this.pos;
        }

        return retVal;
    }

    private int skipUtfByteOrderMark(Text value) throws IOException {
        int newMaxLineLength = (int)Math.min(3L + (long)this.maxLineLength,
2147483647L);
        int newSize = this.in.readLine(value, newMaxLineLength,
this.maxBytesToConsume(this.pos));
        this.pos += (long)newSize;
        int textLength = value.getLength();
        byte[] textBytes = value.getBytes();
```

```
        if (textLength >= 3 && textBytes[0] == -17 && textBytes[1] == -69 &&
textBytes[2] == -65) {
            LOG.info("Found UTF-8 BOM and skipped it");
            textLength -= 3;
            newSize -= 3;
            if (textLength > 0) {
                textBytes = value.copyBytes();
                value.set(textBytes, 3, textLength);
            } else {
                value.clear();
            }
        }

        return newSize;
    }

    public synchronized boolean next(LongWritable key, Text value) throws IOException
{
        while(this.getFilePosition() <= this.end ||
this.in.needAdditionalRecordAfterSplit()) {
            key.set(this.pos);
            int newSize;
            newSize = this.in.readLine(value, this.maxLineLength,
this.maxBytesToConsume(this.pos));
            String str=value.toString().replaceAll("\\|\\|","\\|");
            value.set(str);
            this.pos += (long)newSize;

            if (newSize == 0) {
                return false;
            }

            if (newSize < this.maxLineLength) {
                return true;
            }

            LOG.info("Skipped line of size " + newSize + " at pos " + (this.pos -
(long)newSize));
        }

        return false;
    }

    public synchronized float getProgress() throws IOException {
        return this.start == this.end ? 0.0F : Math.min(1.0F, (float)
(this.getFilePosition() - this.start) / (float)(this.end - this.start));
    }

    public synchronized long getPos() throws IOException {
        return this.pos;
    }
```

```
    public synchronized void close() throws IOException {
        try {
            if (this.in != null) {
                this.in.close();
            }
        } finally {
            if (this.decompressor != null) {
                CodecPool.returnDecompressor(this.decompressor);
                this.decompressor = null;
            }

        }

    }

    /** @deprecated */
    @Deprecated
    public static class LineReader extends org.apache.hadoop.util.LineReader {
        LineReader(InputStream in) {
            super(in);
        }

        LineReader(InputStream in, int bufferSize) {
            super(in, bufferSize);
        }

        public LineReader(InputStream in, Configuration conf) throws IOException {
            super(in, conf);
        }

        LineReader(InputStream in, byte[] recordDelimiter) {
            super(in, recordDelimiter);
        }

        LineReader(InputStream in, int bufferSize, byte[] recordDelimiter) {
            super(in, bufferSize, recordDelimiter);
        }

        public LineReader(InputStream in, Configuration conf, byte[] recordDelimiter)
throws IOException {
            super(in, conf, recordDelimiter);
        }
    }
}
```

构建自定义读取数据的处理器，继承TextInputFormat

```
package com.briup.Separator;

import com.google.common.base.Charsets;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapred.*;

import java.io.IOException;

public class UserInputFormat extends TextInputFormat {
    @Override
    public RecordReader<LongWritable, Text> getRecordReader(InputSplit genericSplit,
JobConf job, Reporter reporter) throws IOException {
        reporter.setStatus(genericSplit.toString());
        String delimiter = job.get("textinputformat.record.delimiter");
        byte[] recordDelimiterBytes = null;
        if (null != delimiter) {
            recordDelimiterBytes = delimiter.getBytes(Charsets.UTF_8);
        }

        return new UserRecordReader(job, (FileSplit)genericSplit);
    }
}
```

maven项目打包，并把jar包传入hive所在节点中

```
scp hive_briup-1.0-SNAPSHOT.jar hive用户名@hive节点ip:~
```

将hive中的jar包移动到hive的安装目录下lib中，重启hive服务

```
sudo cp hive_briup-1.0-SNAPSHOT.jar /opt/hive/lib/
或
add jar /home/hdfs/hive_briup-1.0-SNAPSHOT.jar
```

建表：

```
create table demo2(
    id string,
    name string,
    salary string
    )row format delimited fields terminated by '|'
   STORED AS INPUTFORMAT
  'com.briup.Sep.UserInputFormat'
  OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat';
```

> 使用正则匹配的时候列字段类型为string

加载数据

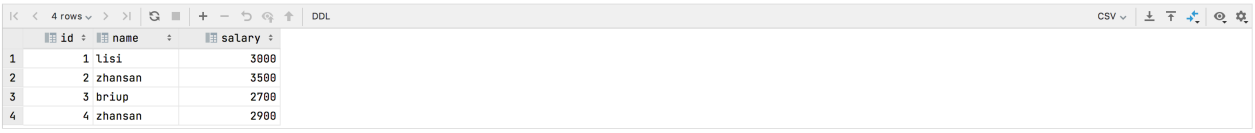
```
load data local inpath '/home/hdfs/demo2.txt'  into table demo2;
```

```
hdfs@master:~$ hdfs dfs -cat /user/hive/warehouse/briup.db/demo2/demo2.txt
1||lisi||3000
2||zhansan||3500
3||briup||2700
4||zhansan||2900
```

查询表结果：

```
select * from demo2;
```

select * from demo2;



	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900

案例3:创建表，将json对象格式的数据导入表中

```
create table demo3(
id int,
name string,
salary double
)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';
```

创建文件

```
vi demo3.txt
```

内容如下:

```
{"id":1,"name":"lisi","salary":3000}
{"id":2,"name":"zhansan","salary":3500}
{"id":3,"name":"briup","salary":2700}
{"id":4,"name":"zhansan","salary":2900}
```

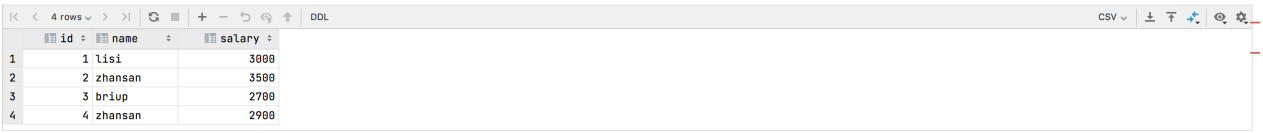
加载数据

```
load data local inpath '/home/hdfs/demo3.txt'  into table demo3;
```

查询表结果:

```
select * from demo3;
```

select * from demo3;



	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900

案例4:创建表，将json对象格式的数据导入表中

```
create table demo4(
    userinfo map<string,string>,
    hobyinfo map<string,string>
)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';
```

创建文件

```
vi demo4.txt
```

内容如下:

```
{"userinfo":{"id":"1","name":"jake","age":"30"},"hobyinfo":
{"eat":"fish","play":"basketball"}}
{"userinfo":{"id":"1","name":"jake","age":"30"},"hobyinfo":
{"eat":"fish","play":"basketball"}}
{"userinfo":{"id":"1","name":"jake","age":"30"},"hobyinfo":
{"eat":"fish","play":"basketball"}}
{"userinfo":{"ids":"1","username":"jake","age":"30"},"hobyinfo":
{"eats":"fish","plays":"basketball"}}
```

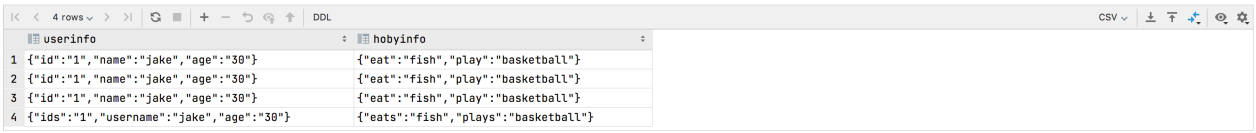
加载数据

```
load data local inpath '/home/hdfs/demo4.txt'  into table demo4;
```

查询表结果:

```
select * from demo4;
```

```
select * from demo4;
```



4 rows

DDL

CSV

	userinfo	hobyinfo
1	{"id":"1","name":"jake","age":"30"}	{"eat":"fish","play":"basketball"}
2	{"id":"1","name":"jake","age":"30"}	{"eat":"fish","play":"basketball"}
3	{"id":"1","name":"jake","age":"30"}	{"eat":"fish","play":"basketball"}
4	{"ids":"1","username":"jake","age":"30"}	{"eats":"fish","plays":"basketball"}

3.6 事务

Hive本身从设计之初时，就是不支持事务的，Hive的核心目标是将hdfs中存在的结构化数据文件映射成表，然后提供基于表的SQL分析处理，是一款面向分析的工具，不支持随便修改文件数据。意味着早起的Hive中sql语法不支持update、delete操作，也没有所谓的事务支持，完全是面向select查询操作。

从0.14版本开始，Hive把ACID语义的事务添加到hive中去，便于解决特定场景下的特定问题：

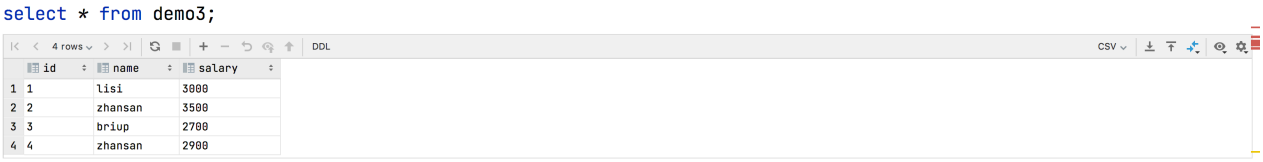
- 流式接收数据。许多用户都有ApacheFlume、ApacheStorm或ApacheKafka等工具，可以用来将数据流传输到他们的Hadoop集群中。虽然这些工具可以每秒写入数百行或更多行的数据，但Hive只能每15分钟到一小时添加一次分区。更频繁地添加分区会很快导致表中出现大量分区。这些工具可以将数据流式传输到现有分区中，但这会导致读卡器进行脏读（也就是说，他们会在开始查询后看到写入的数据），并在目录中留下许多小文件，这会给NameNode带来压力。有了这个新功能，这个用例将得到支持，同时允许读者获得一致的数据视图，并避免过多的文件。
- 缓慢变化的维度。在典型的星型模式数据仓库中，维度表会随着时间缓慢变化。例如，零售商将开设新的门店，需要将其添加到门店表中，或者现有门店可能会更改其占地面积或其他一些跟踪特征。这些更改导致插入单个记录或更新记录（取决于选择的策略）。从0.14开始，Hive能够支持这一点。
- 数据重述。有时收集的数据被发现不正确，需要纠正。或者，数据的第一个实例可能与稍后提供的完整数据近似（90%的服务器报告）。或者，业务规则可能要求由于后续交易（例如，在购买后，客户可能会购买会员资格，因此有权享受折扣价格，包括之前购买的折扣价格）而对某些交易进行重述。或者，合同可能要求用户在关系终止时删除其客户的数据。从Hive 0.14开始，可以通过插入、更新和删除来支持这些用例。
- 使用SQL MERGE语句进行批量更新。

Hive的局限性：

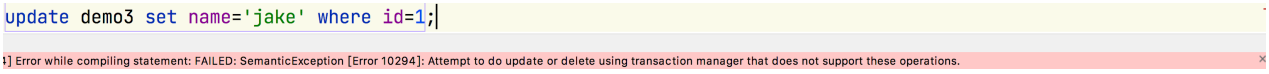
- 尚未支持开始、提交和回滚。所有语言操作都是自动提交的。计划在未来的版本中支持这些功能。
- 仅支持ORC文件格式
- 默认情况下，事务被配置为关闭，使用事务需要手动开启
- 表必须是分桶表才能使用事务功能
- 表的参数transaction必须true
- 外部表不能成为ACID，不允许从ACID会话读取 / 写入ACID表

案例1:查询demo3表中的数据，并对id为1的数据修改name属性为jake

```
select * from demo3;
```




```
update demo3 set name='jake' where id=1;
```



> 普通表默认情况下不支持事务操作

案例2:开启事务配置，创建事务表tran_demo

1. HIve开启支持并发

```
set hive.support.concurrency=true;
```

2. 开启分桶表的功能

```
set hive.enforce.bucketing=true;
```

> 从Hive2.0开始不需要配置，默认支持

3. 设置动态分区，非严格模式

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

4. 设置表的引擎支持事务

```
set hive.txn.manager =org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
```

5. 设置Metastore实例上运行启动线程和清理线程

```
set hive.compactor.initiator.on=true;
```

6. 设置开启Metastore实例上运行多少个压缩程序工作线程

```
set hive.compactor.worker.threads =1;
```

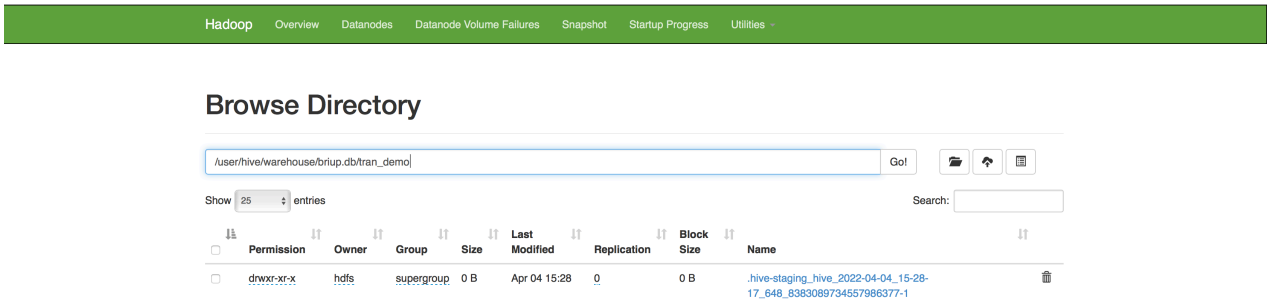
7. 创建表

```
CREATE TABLE tran_demo (
  id                int,
  name              string
)
CLUSTERED BY (id) INTO 2 BUCKETS STORED AS ORC
TBLPROPERTIES ("transactional"="true");
```

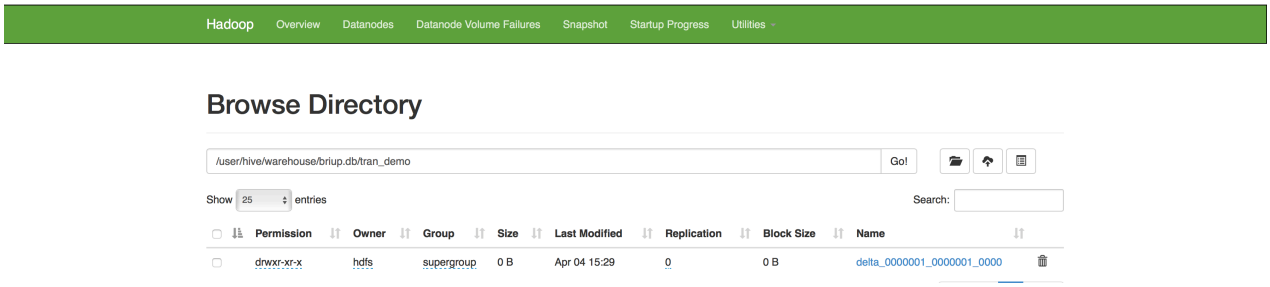
8. 插入数据

```
insert into tran_demo values(1,'jake');
```

插入数据，事务生效



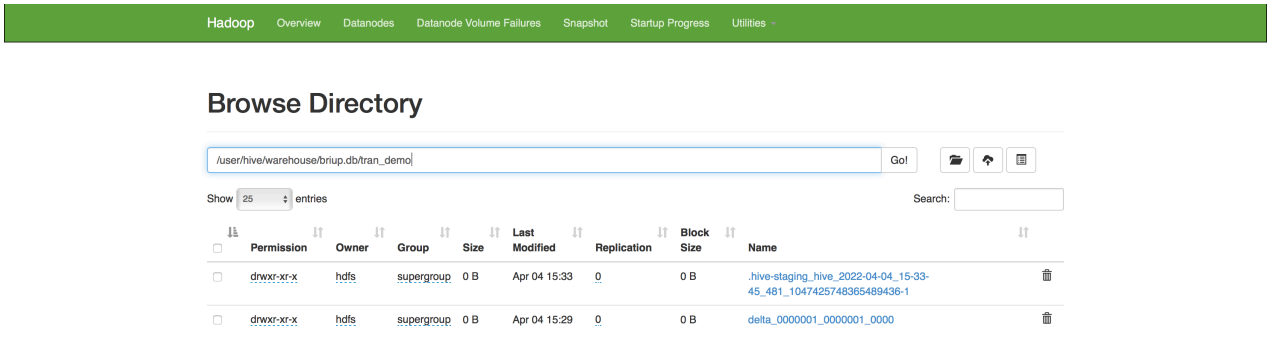
事务自动提交，事务结束



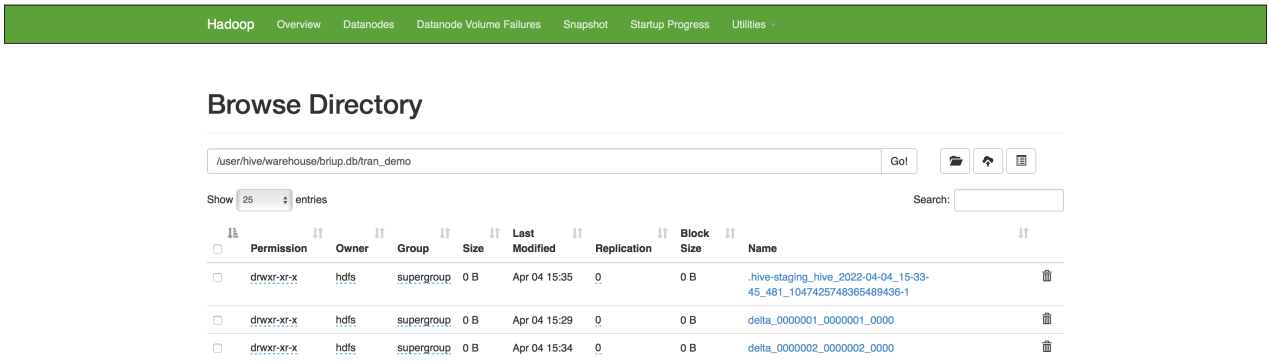
9. 修改数据

```
update   tran_demo set name='briup' where id=1;
```

插入数据，事务生效



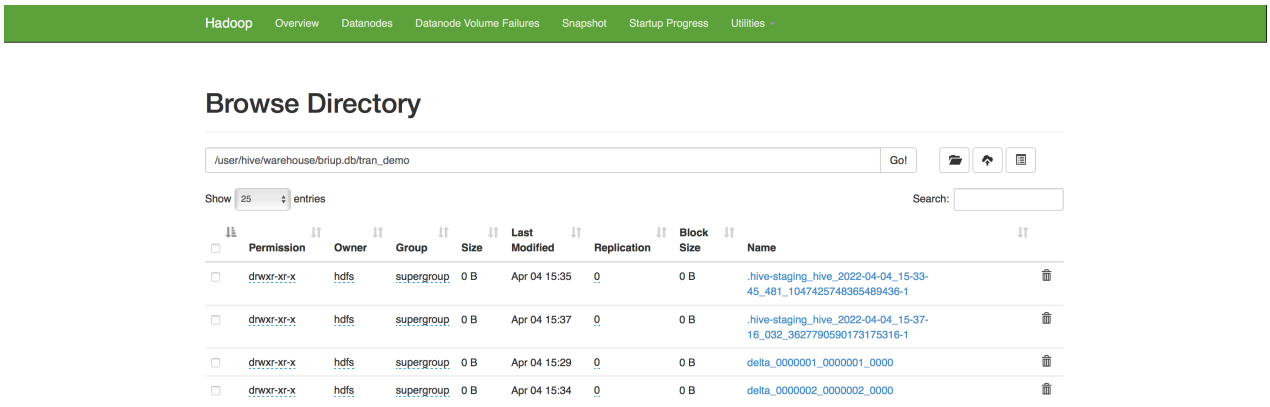
事务自动提交，事务结束



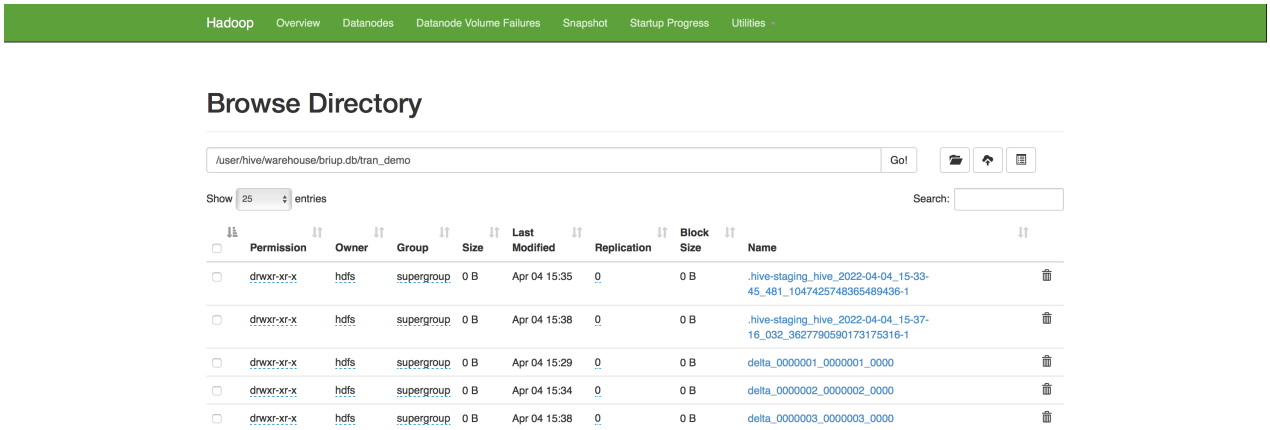
10. 删除数据

```
delete from tran_demo where id=1;
```

插入数据，事务生效



事务自动提交，事务结束



事务实现原理：

- Hdfs 文件为原始数据，用delta保存事务操作的记录增量数据
- 正在执行中的事务，以hive-staging开头的文件维护，执行结束就是delta文件目录，每次操作生成一个delta文件目录。
- 当访问数据时，根据hdfs原始文件和delta文件目录合并，查询最新的数据。

小合并:将一组delta增量文件写为单个增量文件，默认触发条件为10个delta文件；

```
set hive.compactor.delta.num.threshold=10
```

大合并：一个或多个增量文件和基础文件重新写为一个基础文件，默认触发条件为delta文件相当于基础文件占比10%

```
set hive.compactor.delta.pct.threshold=0.1
```

3.7 视图

Hive中的视图是一种虚拟表，只能保存定义，不能实际存储数据，视图是从真实表中查询创建生成，同样可以在已经存在的视图中创建新视图，如果删除真实表或修改真实表，视图将失效，视图的目的是简化操作，不缓存数据，不会提高查询性能。

案例1:查询demo3中所有数据

```
select * from demo3;
```

```
select * from demo3;
```

4 rows DDL CSV

	id	name	salary
1	1	lisi	3000
2	2	zhansan	3500
3	3	briup	2700
4	4	zhansan	2900

案例2:创建视图，隐藏demo3中salary;

```
create view v_demo3 as select id,name from demo3;
```

> 可以基于多个表查询作为视图

查询视图中的数据

```
select * from v_demo3;
```

```
select * from v_demo3;
```

4 rows DDL CSV

	id	name
1	1	lisi
2	2	zhansan
3	3	briup
4	4	zhansan

案例3:展示所有视图;

```
show views;
```

```
show views ;
```

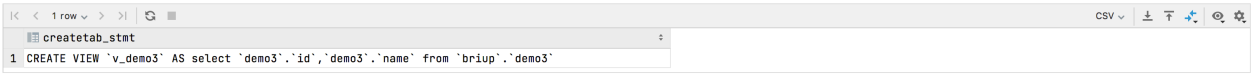
1 row CSV

	tab_name
1	v_demo3

案例4:查看视图的原始构建语句

```
show create table v_demo3;
```

```
show create table v_demo3;
```



> show create table 也可以查看建表的原始操作

案例5:删除视图

```
drop view v_demo3;
```

4 HiveQL查询语法

4.1 查询语法

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list [HAVING condition]]
[CLUSTER BY col_list
| [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]
]
[LIMIT number]
```

1. select 后跟需要展示的列
2. FROM 后跟查询的表
3. group by 分组的标准
4. having 对分组条件的过滤
5. order by 会对输入做全局排序，因此只有一个reducer，会导致当输入规模较大时，需要较长的计算时间。
6. sort by不是全局排序，其在数据进入reducer前完成排序。因此，如果用sort by进行排序，并且设置mapred.reduce.tasks>1，则sort by只保证每个reducer的输出有序，不保证全局有序。
7. distribute by(字段)根据指定的字段将数据分到不同的reducer，且分发算法是hash散列。
8. cluster by(字段) 除了具有distribute by的功能外，还会对该字段进行排序

> distribute 和sort字段是同时存在，`cluster by = distribute by + sort by`

4.2 基本查询

创建表

```
create  table emp(
id int ,
name string,
salary double
)
row format delimited fields terminated by ',';
```

创建文件

```
vi emp.txt
```

内容如下

```
1,lisi,3000
2,jake,2500
3,tom,3500
4,lili,4000
5,briup,800
6,rose,4300
7,zhangsan,2400
8,lulu,3500
```

加载数据

```
load data local inpath '/home/hdfs/emp.txt' into table emp;
```

- 全表查询

```
select * from emp;
```

```
select * from emp;
```

8 rows | DDL | CSV

	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500
4	4	lili	4000
5	5	briup	800
6	6	rose	4300
7	7	zhangsan	2400
8	8	lulu	3500

- 选择特定列

```
select id,name from emp;
```

```
select id,name from emp;
```

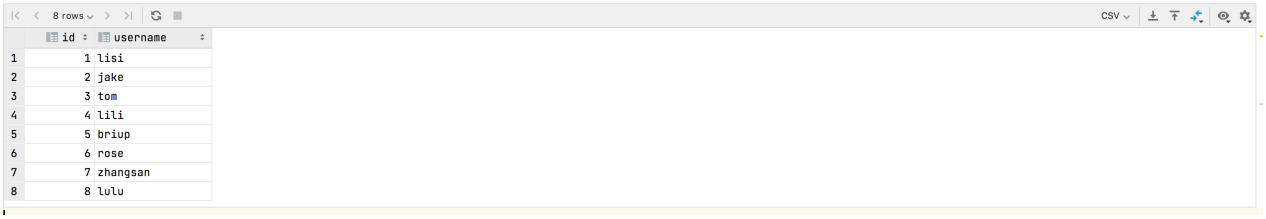
8 rows | DDL | CSV

	id	name
1	1	lisi
2	2	jake
3	3	tom
4	4	lili
5	5	briup
6	6	rose
7	7	zhangsan
8	8	lulu

- 列起别名

```
select id,name as username from emp;
```

```
select id,name as username from emp;
```



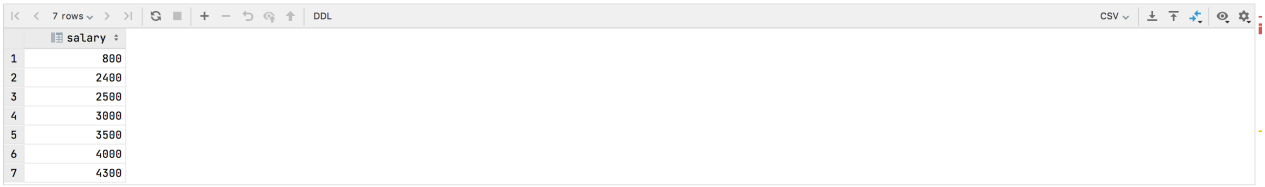
	id	username
1	1	lisi
2	2	jake
3	3	tom
4	4	lili
5	5	briup
6	6	rose
7	7	zhangsan
8	8	lulu

> 起别名的时候as可以省略

- 重复的列去重

```
select distinct salary from emp;
```

```
select distinct salary from emp;
```



	salary
1	800
2	2400
3	2500
4	3000
5	3500
6	4000
7	4300

> distinct去重的时候，当后面的列组合重复的时候才会去掉重复

4.3 条件筛选

- 使用WHERE 子句，将不满足条件的行过滤掉
- WHERE 子句紧随 FROM 子句

4.3.1 关系运算符：

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，则返回TRUE，其他的和等号（=）操作符的结果一致，如果任意为NULL则结果为NULL
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任何一个为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，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是一个正则表达式，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。

案例1:查询id小于3的员工信息：

```
select * from emp where id<3;
```

```
select * from emp where id<3;
```

	id	name	salary
1	1	lisi	3000
2	2	jake	2500

案例2:查询id等于1的员工信息:

```
select * from emp where id=1;
```

```
select * from emp where id=1;
```

	id	name	salary
1	1	lisi	3000

案例3:查询id在1到3的员工信息

```
select * from emp where id between 1 and 3;
```

```
select * from emp where id between 1 and 3;
```

	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500

> between A and B是闭区间，包含A和B两个节点

案例4:查询员工名字为null的员工信息

```
select * from emp where name is null;
```

```
select * from emp where name is null;
```

id	name	salary

案例5:查询员工编号为1和5的员工信息

```
select * from emp where id in(1,5);
```

```
select * from emp where id in(1,5);
```

	id	name	salary
1	1	lisi	3000
2	5	briup	800

案例7:查询员工名字中含有a的员工信息

```
select * from emp where name like '%a%';
```

```
select * from emp where name like '%a%';
```

	id	name	salary
1	2	jake	2500
2	7	zhangsan	2400

案例8:查询员工第二个字母为h的员工信息

```
select * from emp where name like '_h%';
```

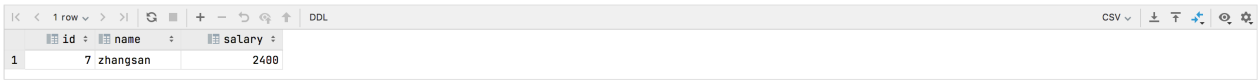
```
select * from emp where name like '_h%';
```

	id	name	salary
1	7	zhangsan	2400

> LIKE选择含有的值
>
> % 代表零个或多个字符(任意个字符)。 _ 代表一个字符。

案例9:查询name中含有h的数据

```
select * from emp where name rlike '[h]';
```



	id	name	salary
1	7	zhangsan	2400

4.3.2 逻辑运算符

操作符	含义
OR	逻辑或
AND	逻辑并
NOT	逻辑否

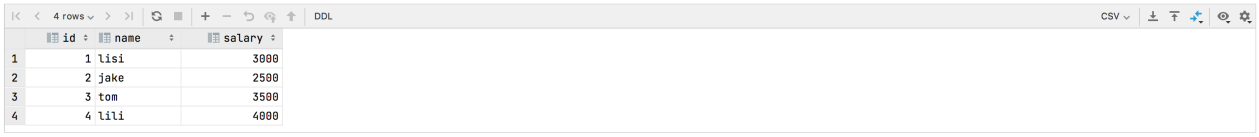
案例1:查询id为1，姓名中包含h的学生信息

```
select * from  emp where id=1 and name rlike '[h]';
```

	id	name	salary
1	7	zhangsan	2400

案例2:查询学号小于5或大于10的学生信息

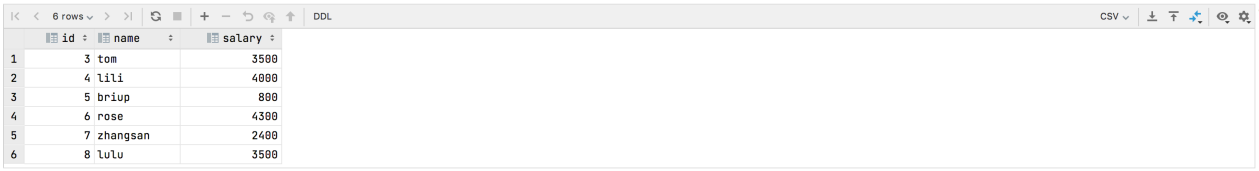
```
select * from emp where id<5 or id>10;
```



	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500
4	4	lili	4000

案例3:查询学号不是1和2的学生信息

```
select * from emp where id not in(1,2);
```



	id	name	salary
1	3	tom	3500
2	4	lili	4000
3	5	briup	800
4	6	rose	4300
5	7	zhangsan	2400
6	8	lulu	3500

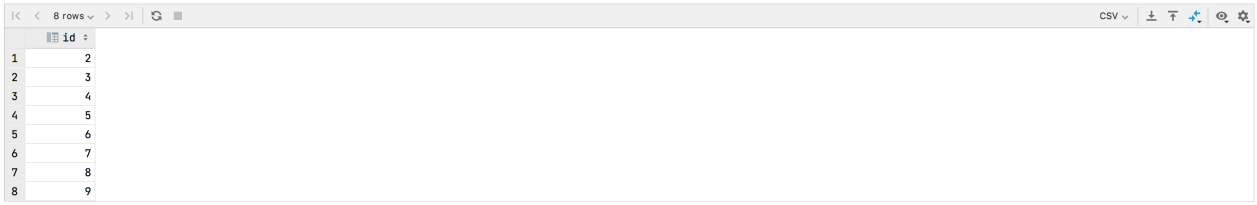
4.3.3 算数运算符

操作符	含义
A+B	加
A-B	减
A*B	乘
A/B	除
A%B	取余
A&B	按位与
A|B	按位或
A^B	按位异或
~B	按位非

案例1:查询所有学生信息，学号往后推一位

```
select id+1 as id from emp;
```

```
select id+1 as id from emp;
```



	id
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9

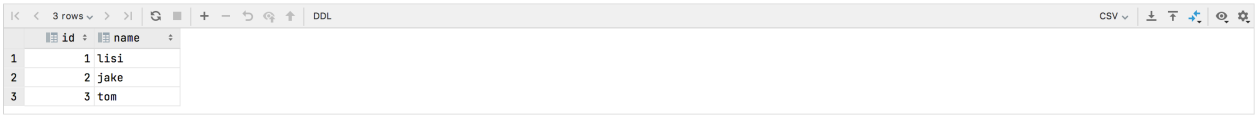
4.4 展示选择

查询会返回多行数据。LIMIT子句用于限制返回的行数。

案例1:查询学生信息，只显示前3行

```
select id,name  from emp limit 3;
```

```
select id,name  from emp limit 3;
```

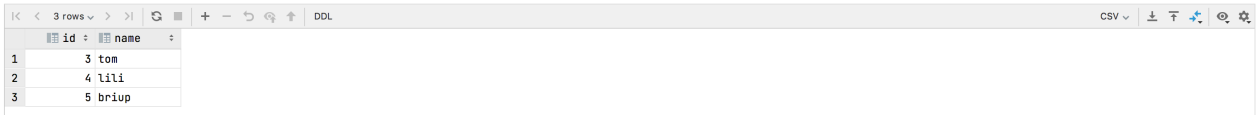


	id	name
1	1	lisi
2	2	jake
3	3	tom

案例2:查询学生信息，只显示第2开始的3行

```
select id,name  from emp limit 2,3;
```

```
select id,name  from emp limit 2,3;
```

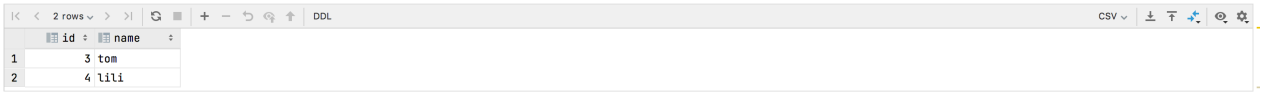


	id	name
1	3	tom
2	4	lili
3	5	briup

案例3:查询学生信息，只显示第2开始的2行

```
select id,name  from emp limit 2,2;
```

```
select id,name  from emp limit 2,2;
```



	id	name
1	3	tom
2	4	lili

4.5 常规函数

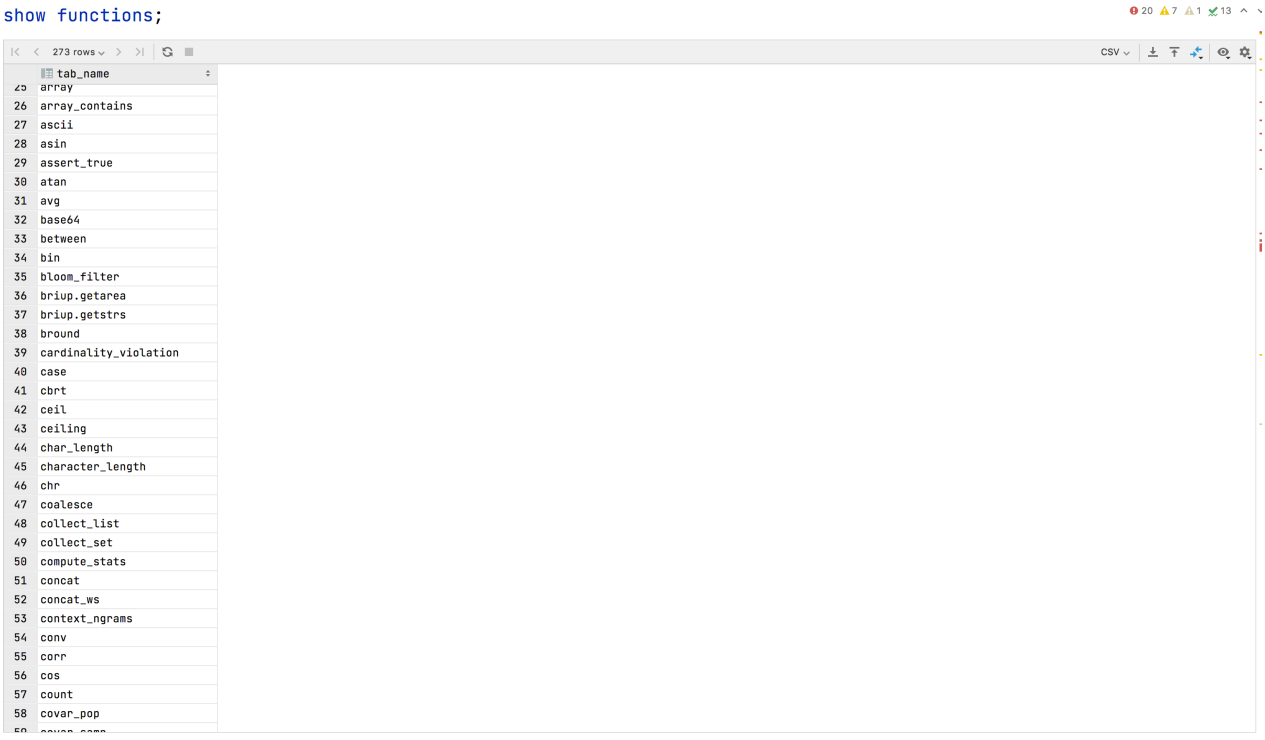
内容较多，见《Hive官方文档》

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

- 查看系统自带的函数

```
show functions;
```

```
show functions;
```



	tab_name
25	array
26	array_contains
27	ascii
28	asin
29	assert_true
30	atan
31	avg
32	base64
33	between
34	bin
35	bloom_filter
36	briup.getarea
37	briup.getstrs
38	bround
39	cardinality_violation
40	case
41	cbrt
42	ceil
43	ceiling
44	char_length
45	character_length
46	chr
47	coalesce
48	collect_list
49	collect_set
50	compute_stats
51	concat
52	concat_ws
53	context_ngrams
54	conv
55	corr
56	cos
57	count
58	covar_pop

- 显示自带的函数的用法

```
desc function upper;
```

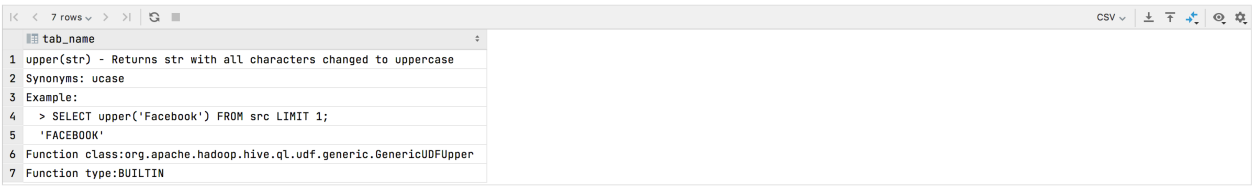
```
desc function upper;
```

	tab_name
1	upper(str) - Returns str with all characters changed to uppercase

- 详细显示自带的函数的用法

```
desc function extended upper;
```

```
desc function extended upper;
```



	tab_name
1	upper(str) - Returns str with all characters changed to uppercase
2	Synonyms: ucase
3	Example:
4	> SELECT upper('Facebook') FROM src LIMIT 1;
5	'FACEBOOK'
6	Function class:org.apache.hadoop.hive.ql.udf.generic.GenericUDFUpper
7	Function type:BUILTIN

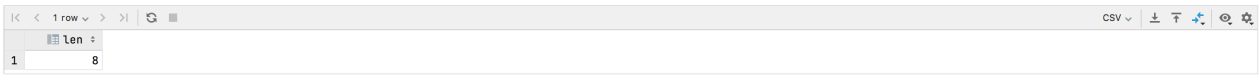
4.5.1 数学函数

数学函数	描述
count/sum/max/min/avg	聚合函数
exp(a)	e^a
pow(a,b)	a^b
sqrt(a)	平方根
ln/log2/log10/log(base,a)	对数
round(num,n)	四舍五入
floor(num)	地板
ceil(num)	天花板
rand(), rand(int,seed)	随机数
abs(a)	绝对值
sign(a)	如果a是正数则返回1.0，是负数则返回-1.0，否则返回0.0
e()/pi()	常数e/pi
greatest(T v1, T v2, ...)	最大值
least(T v1, T v2, ...)	最小值

案例1:查询员工信息总行数

```
select count(*) as len from emp;
```

```
select count(*) as len from emp;
```

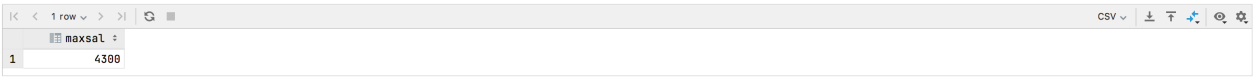


	len
1	8

案例2:查询员工最大的工资数

```
select max(salary) as maxsal from emp;
```

```
select max(salary) as maxsal from emp;
```



	maxsal
1	4300

案例3:查询员工最小的工资数

```
select min(salary) as minsal from emp;
```

```
select min(salary) as minsal from emp;
```

	minsal
1	800

案例4:查询员工薪资总和

```
select sum(salary) as sumsal from emp;
```

```
select sum(salary) as sumsal from emp;
```

	sumsal
1	24000

案例5:查询员工平均薪资

```
select avg(salary) as avgsal from emp;
```

```
select avg(salary) as avgsal from emp;
```

	avgsal
1	3000

案例6:查询e的2次方

```
select exp(2) as exp;
```

```
select exp(2) as exp;
```

	exp
1	7.38905609893065

案例7:查询员工薪资的2次方

```
select pow(salary,2) as pow from emp;
或
select sqrt(salary) as sqrt from emp;
```

```
select pow(salary,2) as pow from emp;
```

	pow
1	9000000
2	6250000
3	12250000
4	16000000
5	640000
6	18490000
7	5760000
8	12250000

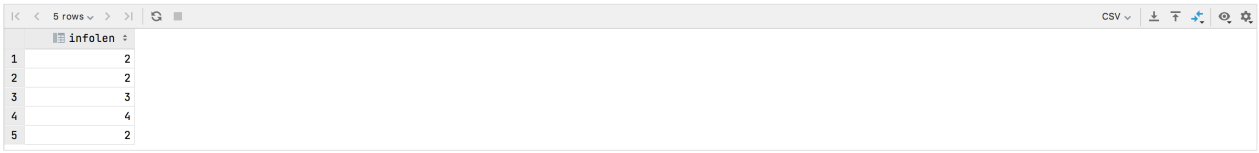
4.5.2 集合函数

集合函数	描述
`size(Map<K.V>)`	它返回在映射类型的元素的数量。
`size(Array<T>)`	它返回在数组类型元素的数量。
collect_set(col)	行转数组(去重)
collect_list(col)	行转数组(不去重)
`array_contains(Array<T>, value)`	数组中是否包含value
explode(ARRAY)	数组转行
`explode(Map<K.V>)`	每行对应每个map的key-value，返回key,value两个字段
`map_keys(Map<K.V>)`	返回所有的keys
`map_values(Map<K.V>)`	返回所有的values
`sort_array(Array<T>)`	对数据进行排序返回
concat_ws(sep,array)	将array中的元素合并成字符

案例1:查询tab_map中info列中map有多少键值对

```
select size(info) as infolen from tab_map;
```

```
select size(info) as infolen from tab_map;
```

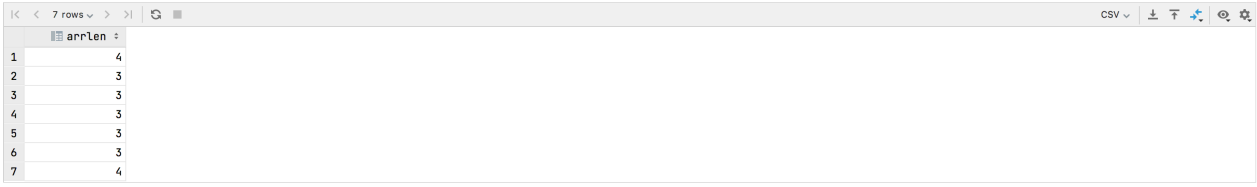


	infolen
1	2
2	2
3	3
4	4
5	2

案例2:查询tab_array中b列中array有多少元素

```
select size(arr)s as arrlen from tab_array;
```

```
select size(arr) as arrlen from tab_array;
```

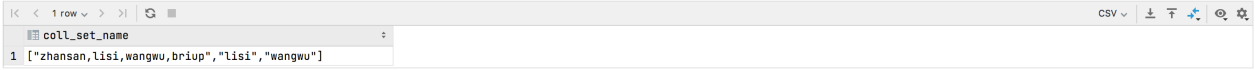


	arrlen
1	4
2	3
3	3
4	3
5	3
6	3
7	4

案例3:查询学生的所有名字，名字重复的去重，要求返回的是Set

```
select collect_set(name) as coll_set_name from stu;
```

```
select collect_set(name) as coll_set_name from stu;
```



coll_set_name
1 ["zhansan,lisi,wangwu,briup","lisi","wangwu"]

案例4:查询学生的所有名字，要求返回的是List

```
select collect_list(name) as coll_list_name from stu;
```

```
select collect_list(name) as coll_list_name from stu;
```

	coll_list_name
1	["zhansan,lisi,wangwu,briup","lisi","wangwu"]

案例5:查询数组中含有briup的行

```
select * from tab_array where array_contains(arr,'briup');
```

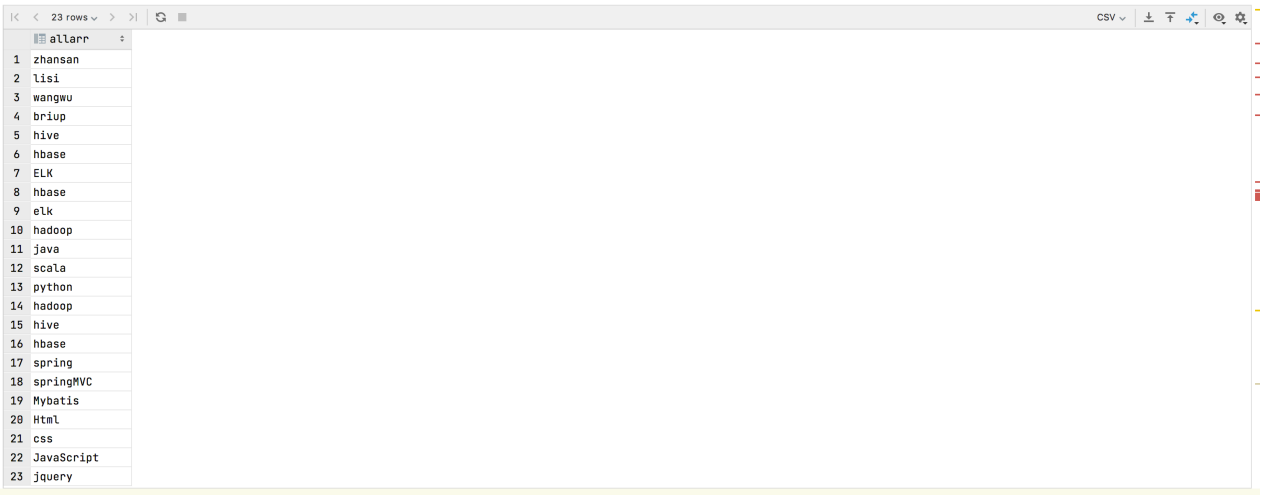
```
select * from tab_array where array_contains(arr,'briup');
```

	id	arr
1	6	["zhansan","lisi","wangwu","briup"]

案例5:将数组中每个元素转化为行

```
select explode(arr) as allarr from tab_array;
```

```
select explode(arr) as allarr from tab_array;
```

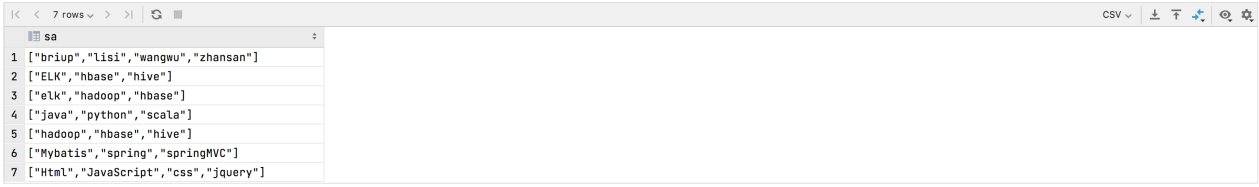


	allarr
1	zhansan
2	lisi
3	wangwu
4	briup
5	hive
6	hbase
7	ELK
8	hbase
9	elk
10	hadoop
11	java
12	scala
13	python
14	hadoop
15	hive
16	hbase
17	spring
18	springMVC
19	Mybatis
20	Html
21	css
22	JavaScript
23	jquery

案例6:将数组中元素排序

```
select sort_array(arr) as sa from tab_array;
```

```
select sort_array(arr) as sa from tab_array;
```

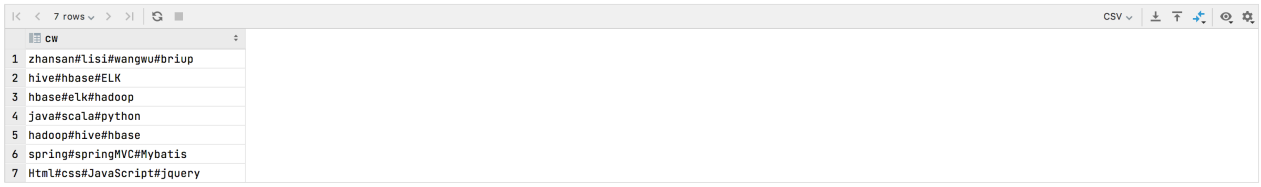


	sa
1	["briup","lisi","wangwu","zhansan"]
2	["ELK","hbase","hive"]
3	["elk","hadoop","hbase"]
4	["java","python","scala"]
5	["hadoop","hbase","hive"]
6	["Mybatis","spring","springMVC"]
7	["Html","JavaScript","css","jquery"]

案例7:将数组中元素基于#合并

```
select concat_ws('#',arr) as cw from tab_array;
```

```
select concat_ws('#',arr) as cw from tab_array;
```

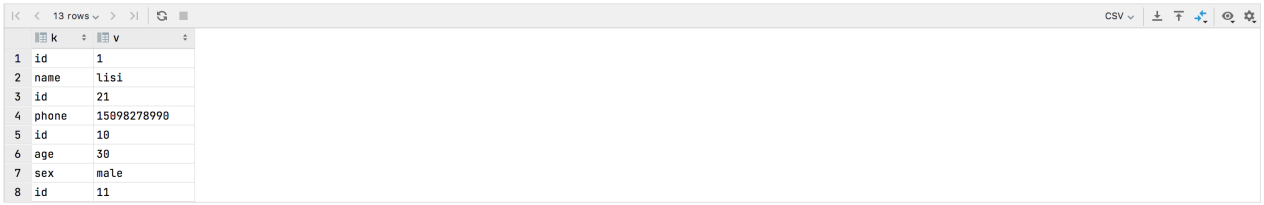


	cw
1	zhansan#lisi#wangwu#briup
2	hive#hbase#ELK
3	hbase#elk#hadoop
4	java#scala#python
5	hadoop#hive#hbase
6	spring#springMVC#Mybatis
7	Html#css#JavaScript#jquery

案例8:将map中的键值对返回两列

```
select explode(info) as (k,v) from tab_map;
```

```
select explode(info) as (k,v) from tab_map;
```

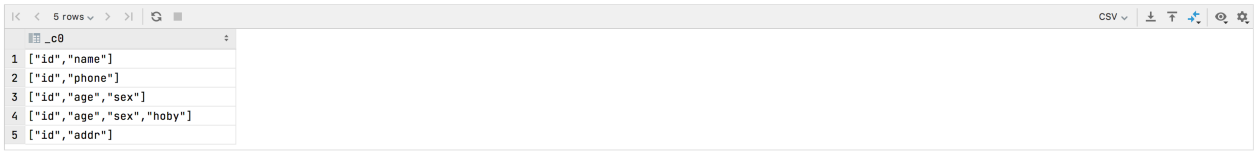


	k	v
1	id	1
2	name	lisi
3	id	21
4	phone	15098278990
5	id	10
6	age	30
7	sex	male
8	id	11

案例9:将map中的所有的键返回

```
select map_keys(info) as keys from tab_map;
```

```
select map_keys(info) as keys from tab_map;
```

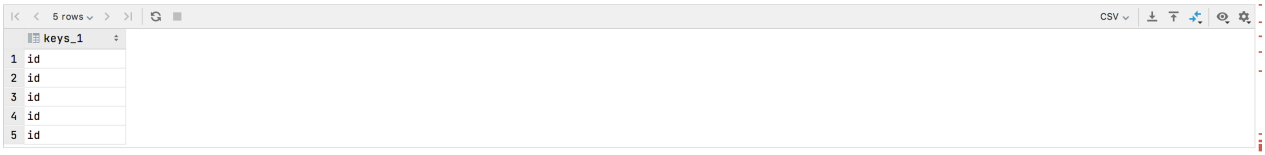


	_c0
1	["id","name"]
2	["id","phone"]
3	["id","age","sex"]
4	["id","age","sex","hoby"]
5	["id","addr"]

案例10:将map中的所有的键返回,并显示第一个键

```
select map_keys(info)[0] as keys_1 from tab_map;
```

```
select map_keys(info)[0] as keys_1 from tab_map;
```

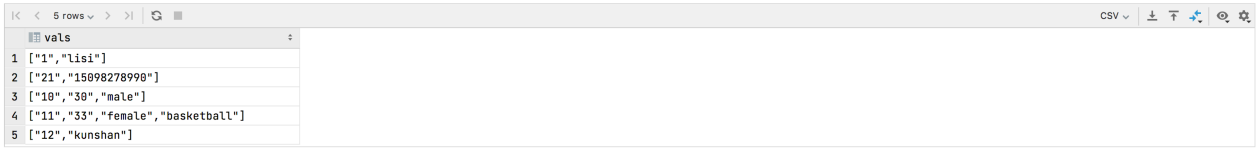


	keys_1
1	id
2	id
3	id
4	id
5	id

案例11:将map中的所有的值返回

```
select map_values(info) as vals from tab_map;
```

```
select map_values(info) as vals from tab_map;
```



	vals
1	["1","lisi"]
2	["21","15098278990"]
3	["10","30","male"]
4	["11","33","female","basketball"]
5	["12","kunshan"]

4.5.3 常用函数

常用	描述
corr(col1, col2)	相关系数
length(string)	字符串长度
concat(string A, string B,...)	它返回从A后串联B产生的字符串
concat_ws(sep,string A, string B,...)	指定分隔符合并
concat_ws(sep,array)	将array中的元素合并成字符
substr(string A, int start, int length)	取子集
upper(string A)/lower	
trim(string A)	删除两侧空格
ltrim(string A)/rtrim(string A)	删除左/右空格
regexp_replace(string A, string B, string C)	用C替换B
split(str,pat)	分割字符串，返回array
percentile_approx(col, p，B)	百分位数
`cast(<expr> as <type>)`	将表达式的结果转换类型
case when ... then...else...end	
between...and...	
if(condition,valueTrue,valueFalseorNull)	条件判断
nvl(value,default_value)	如果value值为NULL就返回default_value,否则返回value
coalesce(T v1, T v2, ...)	返回第一非null的值，如果全部都为NULL就返回NUL
isnull(a)/isnotnull(a)	判断返回boolean

案例1:获取字符串的长度

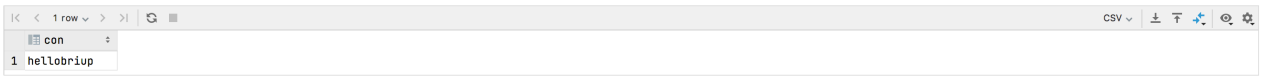
```
select length('hello briup') as len;
```

```
select length('hello briup') as len;
```

案例2:将两个字符串连在一起

```
select concat('hello','briup') as con;
```

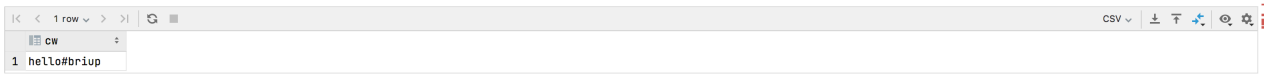
```
select concat('hello','briup') as con;
```



案例3:将两个字符串基于#连在一起

```
select concat_ws('#','hello','briup') as cw;
```

```
select concat_ws('#','hello','briup') as cw;
```

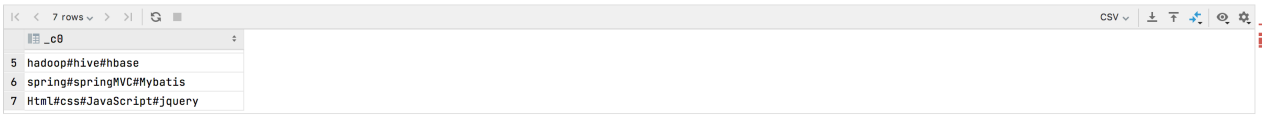


	cw
1	hello#briup

案例4:将数组btz中的元素基于#连在一起

```
select concat_ws('#',arr) from tab_array;
```

```
select concat_ws('#',arr) from tab_array;
```



	_c0
5	hadoop#hive#hbase
6	spring#springMVC#Mybatis
7	Html#css#JavaScript#jquery

案例5:将字符串变成大写

```
select upper('hello') as up;
```

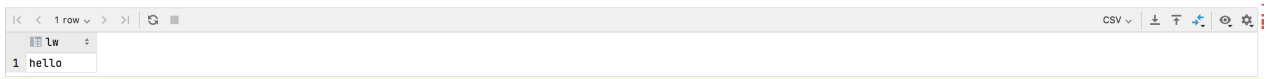
```
select upper('hello') as up;
```

	up
1	HELLO

案例6:将字符串变成小写

```
select lower('HELLO') as lw;
```

```
select lower('HELLO') as lw;
```

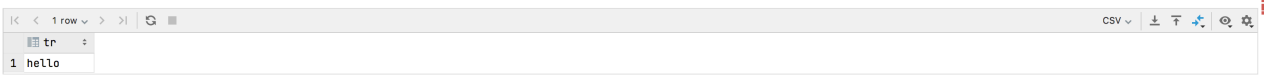


	lw
1	hello

案例7:将字符串前后空格去掉

```
select trim('    hello    ') as tr;
```

```
select trim('    hello    ') as tr;
```

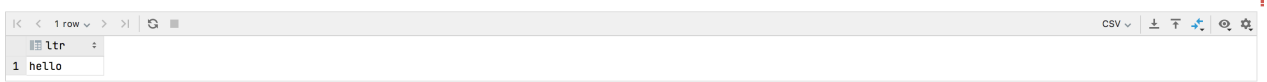


	tr
1	hello

案例8:将字符串中左边空格去掉

```
select ltrim('  hello    ') as ltr;
```

```
select ltrim('  hello    ') as ltr;
```

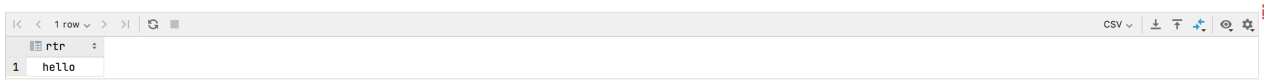


	ltr
1	hello

案例9:将字符串中右边空格去掉

```
select rtrim('  hello    ') as rtr;
```

```
select rtrim('  hello    ') as rtr;
```



	rtr
1	hello

案例10:将字符串中hello使用say替换

```
select regexp_replace('hello briup','hello','say') as rr;
```

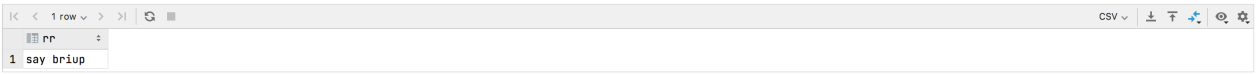
```
select regexp_replace('hello briup','hello','say') as rr;
```

	rr
1	say briup

案例11:将字符串中从字符串开始到o的内容使用say替换

```
select regexp_replace('hello briup','.{4}o','say') as rr;
```

```
select regexp_replace('hello briup','.*o','say') as rr;
```



	rr
1	say briup

案例12:将字符串基于逗号拆分成Arry数组类型

```
select split('java,hadoop,scala,hive,spark',',') as arr;
```

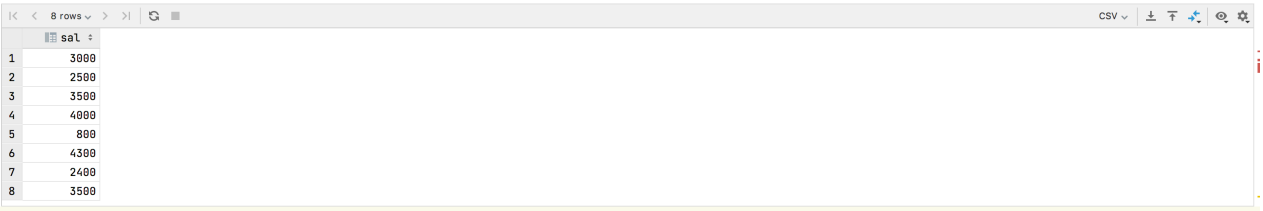
```
select split('java,hadoop,scala,hive,spark',',') as arr;
```

	arr
1	["java","hadoop","scala","hive","spark"]

案例13:将double类型的薪水转化为Int类型

```
select cast(salary as int) as sal from emp;
```

```
select cast(salary as int) as sal from emp;
```

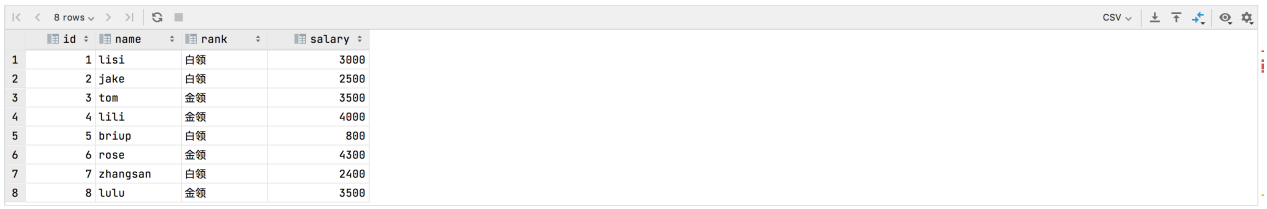


	sal
1	3000
2	2500
3	3500
4	4000
5	800
6	4300
7	2400
8	3500

案例14:员工薪水大于3000显示为金领，否则显示为白领

```
select id,name,case when salary>3000 then '金领' else '白领' end as rank,salary
from emp;
```

```
select id,name,case when salary>3000 then '金领' else '白领' end as rank,salary
from emp;
```



	id	name	rank	salary
1	1	lisi	白领	3000
2	2	jake	白领	2500
3	3	tom	金领	3500
4	4	lili	金领	4000
5	5	briup	白领	800
6	6	rose	金领	4300
7	7	zhangsan	白领	2400
8	8	lulu	金领	3500

案例15:员工薪水大于3000显示为金领，否则显示为白领

```
select id,name,if(salary>3000,'金领','白领') as rank
from emp;
```

```
select id,name,if(salary>3000,'金领','白领') as rank
from emp;
```

8 rows

	id	name	rank
1	1	lisi	白领
2	2	jake	白领
3	3	tom	金领
4	4	lili	金领
5	5	briup	白领
6	6	rose	金领
7	7	zhangsan	白领
8	8	lulu	金领

案例16:计算员工的年薪，如果薪水为NULL的，使用0替换

```
select nvl(salary,0)*13 as nsal from emp;
```

```
select nvl(salary,0)*13 as nsal from emp;
```

8 rows

	nsal
1	39000
2	32500
3	45500
4	52000
5	10400
6	55900
7	31200
8	45500

> null值参与计算都为null

4.5.4 日期函数

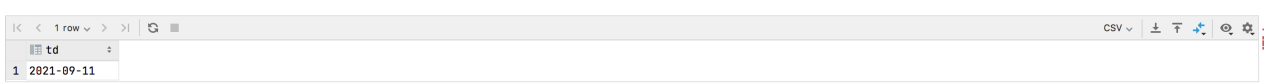
日期时间函数	说明
from_unixtime(int unixtime)	转换的秒数从Unix纪元(1970-01-0100:00:00 UTC)
to_date(string timestamp)	返回一个字符串时间戳的日期部分
date_format(date,format)	按制定格式返回日期 Format参考文献
year(string date)	
quarter(string date)	季度
month(string date)	
day(string date)	
current_date()	当前日期
weekofyear(string date)	
datediff(start，end)	返回相差天数
months_between(date,date)	相差月份
date_add()/date_sub()	日期加/减若干天数
add_months(date,int)	日期加若干月
last_day(date)	这个月最后一天
trunc(date,format)	年（“YY”）或月（“MM”）的第一天
next_day(date,string day_of_week)	下个周X对应的日期
get_json_object(string json_string, string path)	

案例1:获取指定时间戳的时间

```
select from_unixtime(1232323221) as utime;
```

案例2:将字符串转化为时间

```
select to_date('2021-09-11 12:22:23') as td;
```



案例3:按照指定格式返回日期

```
select date_format(to_date('2021-09-11 12:22:23'),'yyyy-MM-dd') as ds;
```

```
select date_format(to_date('2021-09-11 12:22:23'),'yyyy-MM-dd') as ds;
```

	ds
1	2021-09-11

格式表：

Date and Time Patterns

Date and time formats are specified by *date and time pattern* strings. Within date and time pattern strings, unquoted letters from `'A'` to `'Z'` and from `'a'` to `'z'` are interpreted as pattern letters representing the components of a date or time string. Text can be quoted using single quotes (`'`) to avoid interpretation. `"''"` represents a single quote. All other characters are not interpreted; they're simply copied into the output string during formatting or matched against the input string during parsing.

The following pattern letters are defined (all other characters from `'A'` to `'Z'` and from `'a'` to `'z'` are reserved):

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

案例4:从指定字符串中返回年份

```
select year('2021-09-11 12:22:23') as year;
```

```
select year('2021-09-11 12:22:23') as year;
```

	year
1	2021

案例5:从指定字符串中返回季度

```
select quarter('2021-09-11 12:22:23') as qurater;
```

```
select quarter('2021-09-11 12:22:23') as qurater;
```

	qurater
1	3

案例6:从指定字符串中返回月份

```
select month('2021-09-11 12:22:23') as month;
```

```
select month('2021-09-11 12:22:23') as month;
```

	month
1	9

案例7:从指定字符串中返回天数

```
select day('2021-09-11 12:22:23') as day;
```

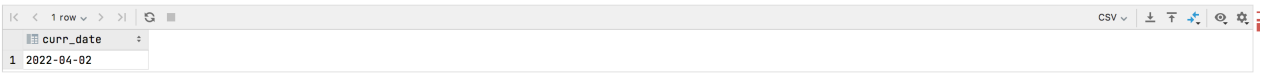
```
select day('2021-09-11 12:22:23') as day;
```

	day
1	11

案例8:获取当前时间

```
select current_date() as curr_date;
```

```
select current_date() as curr_date;
```



	curr_date
1	2022-04-02

案例9:获取一年中的第几周

```
select weekofyear('2021-09-11 12:22:23') as woy;
```

```
select weekofyear('2021-09-11 12:22:23') as woy;
```

	woy
1	36

案例10:计算两个时间相差多少天

```
select datediff('2021-09-11 12:22:23','2020-08-11 12:22:23') as df;
```

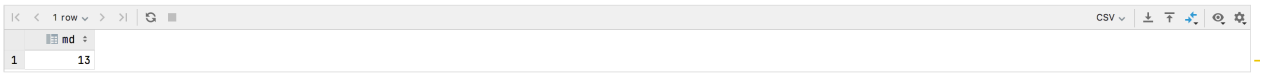
```
select datediff('2021-09-11 12:22:23','2020-08-11 12:22:23') as df;
```

	df
1	396

案例11:计算两个时间相差多少个月

```
select months_between('2021-09-11 12:22:23','2020-08-11 12:22:23') as md;
```

```
select months_between('2021-09-11 12:22:23','2020-08-11 12:22:23') as md;
```

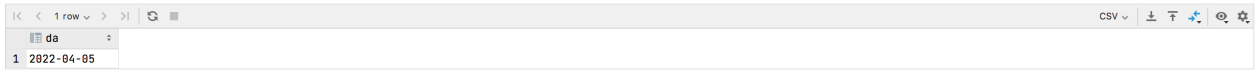


	md
1	13

案例12:当前时间向后推3天

```
select date_add(current_date(),3) as da;
```

```
select date_add(current_date(),3) as da;
```

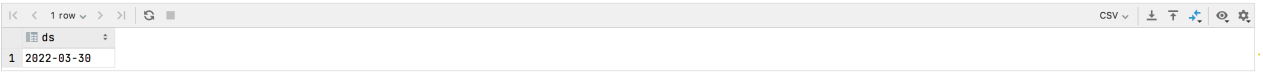


	da
1	2022-04-05

案例13:当前时间向前推3天

```
select date_sub(current_date(),3) as ds;
```

```
select date_sub(current_date(),3) as ds;
```

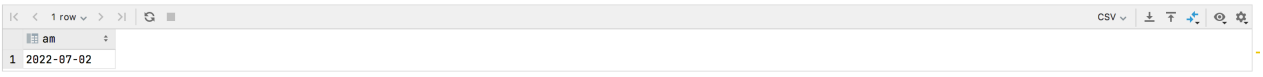


	ds
1	2022-03-30

案例14:当前时间向后添加3个月

```
select add_months(current_date(),3) as am;
```

```
select add_months(current_date(),3) as am;
```



	am
1	2022-07-02

案例15:获取指定月份的最后一天

```
select last_day('2021-09-11 12:22:23') as ld;
```

```
select last_day('2021-09-11 12:22:23') as ld;
```

ld
2021-09-30

案例16:基于日期按照天截取

```
select trunc(current_date(),'MM') as trunc_date;
```

```
select trunc(current_date(),'MM') as trunc_date;
```

trunc_date
2022-04-01

> MM按照天截取，获得指定月份的第一天
>
> YY 按照月份截取，获得指定年份第一个月的第一天

案例17:获取指定时间的下一个星期五是几号

```
select next_day(current_date(),'friday') as nd;
```

```
select next_day(current_date(),'friday') as nd;
```

nd
2022-04-08

案例18:获取json数据中的指定数据

```
select get_json_object('[{"name":"briup","sex":"男","age":"25"},
{"name":"jake","sex":"男","age":"47"}]','$[0].name') as uname;
```

```
select
    get_json_object('[{"name":"briup","sex":"男","age":"25"},{"name":"jake","sex":"男","age":"47"}]','$[0].name')
        as uname;
```

uname
briup

案例19:获取json数据中指定的数据

```
select get_json_object('{"id":1,"name":"lisi","age":30}',"$.name") as username
```

```
select get_json_object('{"id":1,"name":"lisi","age":30}',"$.name") as username;
```

username
lisi

案例20:获取json数据中的多个数据

```
select json_tuple('{"id":1,"name":"lisi","age":30}',"id","name","age") as
(id,name,age)
```

```
select json_tuple('{"id":1,"name":"lisi","age":30}',"id","name","age") as (id,name,age);
```

id	name	age
1	lisi	30

4.5.5 定义函数

Hive 自带了一些函数，比如：max/min等，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).用户自定义函数类别分为以下三种：

- UDF（User-Defined-Function）

  一进一出

- UDAF（User-Defined Aggregation Function）

  聚集函数，多进一出,类似于：count/max/min

- UDTF（User-Defined Table-Generating Functions）

  一进多出,如 、lateral、view、explore()

4.5.5.1 UDF

函数使用的时候传入一个数据返回一个结果，应用到数仓表中，传入某列数据，每一行该单元格输入给出一个结果

实现接口：

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.hadoop.hive.ql.exec;

import org.apache.hadoop.hive.ql.udf.UDFType;

/**
 * A User-defined function (UDF) for use with Hive.
 * <p>
 * New UDF classes need to inherit from this UDF class (or from {@link
 * org.apache.hadoop.hive.ql.udf.generic.GenericUDF GenericUDF} which provides more
flexibility at
 * the cost of more complexity).
 * <p>
 * Requirements for all classes extending this UDF are:
 * <ul>
```

```
 * <li>Implement one or more methods named {@code evaluate} which will be called by
Hive (the exact
 * way in which Hive resolves the method to call can be configured by setting a custom
{@link
 * UDFMethodResolver}). The following are some examples:
 * <ul>
 * <li>{@code public int evaluate();}</li>
 * <li>{@code public int evaluate(int a);}</li>
 * <li>{@code public double evaluate(int a, double b);}</li>
 * <li>{@code public String evaluate(String a, int b, Text c);}</li>
 * <li>{@code public Text evaluate(String a);}</li>
 * <li>{@code public String evaluate(List<Integer> a);} (Note that Hive Arrays are
represented as
 * {@link java.util.List Lists} in Hive.
 * So an {@code ARRAY<int>} column would be passed in as a {@code List<Integer>}.)
</li>
 * </ul>
 * </li>
 * <li>{@code evaluate} should never be a void method. However it can return {@code
null} if
 * needed.
 * <li>Return types as well as method arguments can be either Java primitives or the
corresponding
 * {@link org.apache.hadoop.io.Writable Writable} class.</li>
 * </ul>
 * One instance of this class will be instantiated per JVM and it will not be called
concurrently.
 *
 * @see Description
 * @see UDFType
 */
@UDFType(deterministic = true)
public class UDF {

  /**
   * The resolver to use for method resolution.
   */
  private UDFMethodResolver rslv;

  /**
   * The constructor.
   */
  public UDF() {
    rslv = new DefaultUDFMethodResolver(this.getClass());
  }

  /**
   * The constructor with user-provided {@link UDFMethodResolver}.
   */
  protected UDF(UDFMethodResolver rslv) {
    this.rslv = rslv;
  }
```

```
  /**
   * Sets the resolver.
   *
   * @param rslv The method resolver to use for method resolution.
   */
  public void setResolver(UDFMethodResolver rslv) {
    this.rslv = rslv;
  }

  /**
   * Get the method resolver.
   */
  public UDFMethodResolver getResolver() {
    return rslv;
  }

  /**
   * This can be overridden to include JARs required by this UDF.
   *
   * @see org.apache.hadoop.hive.ql.udf.generic.GenericUDF#getRequiredJars()
   *      GenericUDF.getRequiredJars()
   *
   * @return an array of paths to files to include, {@code null} by default.
   */
  public String[] getRequiredJars() {
    return null;
  }

  /**
   * This can be overridden to include files required by this UDF.
   *
   * @see org.apache.hadoop.hive.ql.udf.generic.GenericUDF#getRequiredFiles()
   *      GenericUDF.getRequiredFiles()
   *
   * @return an array of paths to files to include, {@code null} by default.
   */
  public String[] getRequiredFiles() {
    return null;
  }
}
```

案例1:基于手机号码确定归属地

1. 创建表

```
create table person(id int,name string,tel string)
row format delimited
fields terminated by ','
stored as textfile;
```

2. 构建数据文件

```
vi person.txt
```

3. 内容如下

```
1,张三,13834112233
2,李四,13994200987
3,王五,13302019922
4,jack,13211223344
```

4. 加载数据中

```
load data local inpath '/home/hdfs/person.txt' into table person;
```

5. 构建maven项目
6. pom.xml内容如下如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>Had</artifactId>
        <groupId>com.briup</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>Hive</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>3.0.3</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>3.0.3</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>2.3.5</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-jdbc</artifactId>
            <version>2.3.5</version>
        </dependency>
    </dependencies>
```

```
    <build>
        <finalName>hive</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

7. 构建java类基础UDF

```
package com.briup.DefineFun;
/*
自定义函数
构建evaluate方法
确定参数和返回值（sql需求决定）
 */
import org.apache.hadoop.hive.ql.exec.UDF;

public class GetArea extends UDF {
    public String evaluate(String phone){
        //[0,3)
        String ph=phone.substring(0,3);
        if("138".equals(ph)){
            return "上海";
        }else if("139".equals(ph)){
            return "苏州";
        }else if("133".equals(ph)){
            return "无锡";
        }else if("132".equals(ph)){
            return "南京";
        }else{
            return "未知";
        }
    }
}
```

8. 项目基于maven打成jar包，并把jar包上传到hive所在节点

```
scp /Users/huzhongliang/Documents/idea_work/Had/Hive/target/hive.jar
hdfs@192.168.43.8:~
```

9. 将jar拷贝到hive安装目录lib下

```
sudo cp hive.jar /opt/hive/lib/
```

> 当前操作也可以在hive命令运行窗口使用下面命令替换：
>
> add jar /home/hdfs/hive.jar

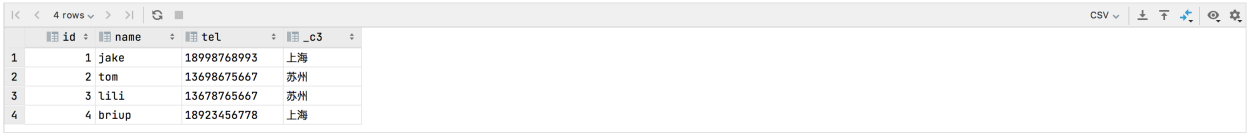
10. 关闭hive命令终端重启，设置函数与我们的自定义函数关联

```
create temporary function getArea as 'com.briup.DefineFun.GetArea';
```

11. 查询用户的基本信息和手机归属地

```
select id,name,tel,getArea(tel) from person;
```

```
select id,name,tel,getArea(tel) from person;
```



	id	name	tel	_c3
1	1	jake	18998768993	上海
2	2	tom	13698675667	苏州
3	3	lili	13678765667	苏州
4	4	briup	18923456778	上海

4.5.5.2 UDTF

函数使用的时候传入一行数据返回多行结果，应用到数仓表中，传入某列数据，每一行该单元格输入给出多行输出结果

实现接口：

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

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

import java.util.List;

import org.apache.hadoop.hive.ql.exec.MapredContext;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.StructField;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
```

```
/**
 * A Generic User-defined Table Generating Function (UDTF)
 *
 * Generates a variable number of output rows for a single input row. Useful for
 * explode(array)...
 */

public abstract class GenericUDTF {
  Collector collector = null;

  /**
   * Additionally setup GenericUDTF with MapredContext before initializing.
   * This is only called in runtime of MapRedTask.
   *
   * @param context context
   */
  public void configure(MapredContext mapredContext) {
  }

  public StructObjectInspector initialize(StructObjectInspector argOIs)
      throws UDFArgumentException {
    List<? extends StructField> inputFields = argOIs.getAllStructFieldRefs();
    ObjectInspector[] udtfInputOIs = new ObjectInspector[inputFields.size()];
    for (int i = 0; i < inputFields.size(); i++) {
      udtfInputOIs[i] = inputFields.get(i).getFieldObjectInspector();
    }
    return initialize(udtfInputOIs);
  }

  /**
   * Initialize this GenericUDTF. This will be called only once per instance.
   *
   * @param argOIs
   *          An array of ObjectInspectors for the arguments
   * @return A StructObjectInspector for output. The output struct represents a
   *         row of the table where the fields of the stuct are the columns. The
   *         field names are unimportant as they will be overridden by user
   *         supplied column aliases.
   */
  @Deprecated
  public StructObjectInspector initialize(ObjectInspector[] argOIs)
      throws UDFArgumentException {
    throw new IllegalStateException("Should not be called directly");
  }

  /**
   * Give a set of arguments for the UDTF to process.
   *
   * @param args
   *          object array of arguments
   */
```

```
  public abstract void process(Object[] args) throws HiveException;

  /**
   * Called to notify the UDTF that there are no more rows to process.
   * Clean up code or additional forward() calls can be made here.
   */
  public abstract void close() throws HiveException;

  /**
   * Associates a collector with this UDTF. Can't be specified in the
   * constructor as the UDTF may be initialized before the collector has been
   * constructed.
   *
   * @param collector
   */
  public final void setCollector(Collector collector) {
    this.collector = collector;
  }

  /**
   * Passes an output row to the collector.
   *
   * @param o
   * @throws HiveException
   */
  protected final void forward(Object o) throws HiveException {
    collector.collect(o);
  }

}
```

> 初始化方法需要重写，否则直接抛出错误

案例1：字段strs中包含多个字符串，按照逗号拆分输出多行

1. 创建表

```
create table MyUDTF(name string);
```

2. 构建数据文件

```
vi strs.txt
```

3. 内容如下

```
java,scala,python
hadoop,hive,hbase,zookeeper
spark,kafka,sqoop
```

4. 加载数据中

```
load data local inpath '/home/hdfs/strs.txt' into table MyUDTF;
```

5. 构建java类基础UDF

```
package com.briup.DefineFun;


import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import
org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactor
y;

import java.util.ArrayList;
import java.util.List;

public class EveryWord extends GenericUDTF {
    private List<String> list=new ArrayList<String>();
    @Override
    public StructObjectInspector initialize(StructObjectInspector argOIs) throws
UDFArgumentException {
        List<String> fieldNames=new ArrayList<String>();
        fieldNames.add("str");
        List<ObjectInspector> fieldOIs=new ArrayList<ObjectInspector>();
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames,
fieldOIs);
    }

    public void process(Object[] args) throws HiveException {
        String data=args[0].toString();
        String der=args[1].toString();
        String[] strs=data.split(der);
        for(String str:strs){
            list.clear();
            list.add(str);
            forward(list);
        }
    }

    public void close() throws HiveException {

    }
}
```

6. 项目基于maven打成jar包，并把jar包上传到hive所在节点

```
scp hd_hdfs-1.0-SNAPSHOT.jar hdfs@192.168.2.22:~
```

7. 将jar拷贝到hive安装目录lib下

```
sudo cp hive.jar /opt/hive/lib/
```

> 当前操作也可以在hive命令运行窗口使用下面命令替换：
>
> add jar /home/hdfs/hive.jar

8. 关闭hive命令终端重启，设置函数与我们的自定义函数关联

```
create temporary function getStrs as 'com.briup.DefineFun.EveryWord';
```

9. 查询结果

```
select getStrs(name,',') from  MyUDTF;
```

```
select getStrs(name,',') from  MyUDTF;
```

10 rows CSV

	str
1	java
2	scala
3	python
4	hadoop
5	hive
6	hbase
7	zookeeper

4.6 分组操作

4.6.1 group by

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。

创建表

```
create table order_1(
id int,
name string,
addr string,
salary double
)
row format delimited
fields terminated by ',';
```

构建数据文件:

```
vi order.txt
```

内容如下：

```
1,jake,昆山,3000
2,briup,上海,3500
3,tom,昆山,2600
4,kevin,上海,5000
5,lili,昆山,12000
6,rose,昆山,3400
7,lucy,上海,8000
8,vens,昆山,2700
```

加载数据:

```
load data local inpath '/home/hdfs/order.txt' into table order_1;
```

案例1:统计每个地区的订单总数

```
select addr,count(*)
from order_1
group by addr;
```

```
select addr,count(*)
from order_1
group by addr;
```

2 rows

	addr	_c1
1	上海	3
2	昆山	5

CSV

4.6.2 having

having对group by分组之后得到的结果进行筛选

案例1:统计订单总数超过3的地区及总数

```
select addr,count(*)
from order_1
group by addr
having count(*)>3;
```

```
select addr,count(*)
from order_1
group by addr
having count(*)>3;
```

1 row

	addr	_c1
1	昆山	5

CSV

> where和having的区别:
>
> where针对from查询的每一行数据过滤；having针对分组之后的结果过滤筛选数据。
>
> where后面不能写分组函数，而having后面可以使用分组函数。

4.7 连接操作

Hive中的Join可分为Common Join（Reduce阶段完成join）和Map Join（Map阶段完成join）。如果不指定MapJoin或者不符合MapJoin的条件（hive.auto.convert.join=true 对于小表启用mapjoin,hive.mapjoin.smalltable.filesize=25M 设置小表的阈值），那么Hive解析器会将Join操作转换成Common Join,即：在Reduce阶段完成join。

Hive中除了支持和传统数据库中一样的内关联、左关联、右关联、全关联，还支持LEFT SEMI JOIN和CROSS JOIN，但这两种JOIN类型，也可以用前面的代替。

测试数据：

创建文件

```
vi user_info
```

内容如下：

```
1,zhangsan
2,lisi
3,wangwu
```

创建文件

```
vi user_age
```

内容如下：

```
1,30
2,29
4,21
```

创建表：

```
create table user_info(id int,name string)
row format delimited
fields terminated by ','
stored as textfile;
create table user_age(id int,age int)
row format delimited
fields terminated by ','
stored as textfile;
```

加载数据中：

```
load data local inpath '/home/hdfs/user_info' into table user_info;
load data local inpath '/home/hdfs/user_age' into table user_age;
```

查询用户信息表

```
select * from user_info;
```

```
select * from user_info;
```

	id	name
1	1	zhangsan
2	2	lisi
3	3	wangwu

查询用户年龄表

```
select * from user_age;
```

```
select * from user_age;
```

3 rows

	id	age
1	1	30
2	2	29
3	4	31

4.7.1 等值连接

多张表查询数据的时候，在笛卡尔积德基础上基于表与表之间关联的列获取有效数据

案例1:查询用户信息及分数

```
select a.id,a.name,b.age
from user_info a,user_age b
where a.id=b.id;
```

```
select a.id,a.name,b.age
from user_info a,user_age b
where a.id=b.id;
```

2 rows

	id	name	age
1	1	zhangsan	30
2	2	lisi	29

4.7.2 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。等价于上面的等值连接

案例1:查询用户信息及分数

```
select a.id,a.name,b.age
from user_info a inner join user_age b
on a.id=b.id;
```

```
select a.id,a.name,b.age
from user_info a inner join user_age b
on a.id=b.id;
```

2 rows

	id	name	age
1	1	zhangsan	30
2	2	lisi	29

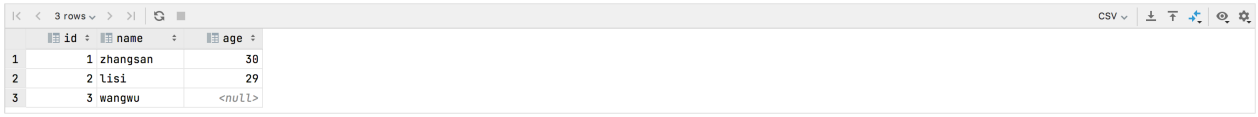
4.7.3 左外连接

左边的表为主表，右边的表为从表，在等值连接的基础之上，主表和从表匹配的数据显示出来，主表没有匹配的数据也显示出来

案例1:查询用户信息及分数，没有分数的用户也要显示出来

```
select a.id,a.name,b.age
from user_info a left join user_age b
on a.id=b.id;
```

```
select a.id,a.name,b.age
from user_info a left join user_age b
on a.id=b.id;
```



	id	name	age
1	1	zhangsan	30
2	2	lisi	29
3	3	wangwu	<null>

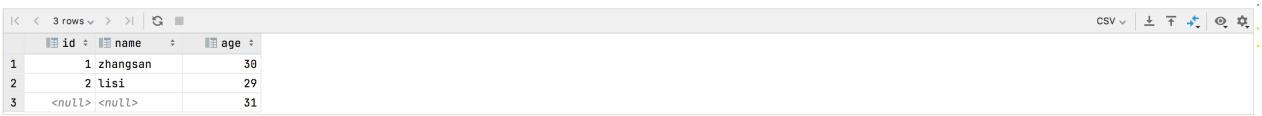
4.7.4 右外连接

左边的表为主表，右边的表为从表，在等值连接的基础之上，主表和从表匹配的数据显示出来,从表没有匹配的数据也显示出来

案例1:查询用户信息及分数，有分数没有用户对应的数据也显示出来

```
select a.id,a.name,b.age
from user_info a right join user_age b
on a.id=b.id;
```

```
select a.id,a.name,b.age
from user_info a right join user_age b
on a.id=b.id;
```



	id	name	age
1	1	zhangsan	30
2	2	lisi	29
3	<null>	<null>	31

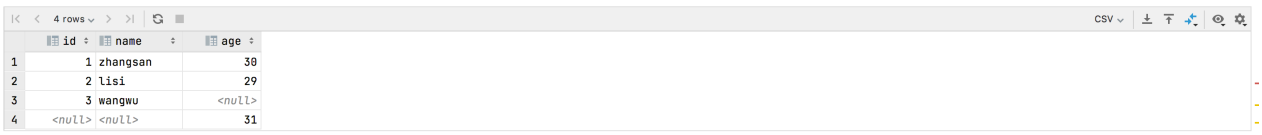
4.7.5 全连接

左边的表为主表，右边的表为从表，在等值连接的基础之上，主表和从表匹配的数据显示出来,主表和从表没有匹配的数据也显示出来

案例1:查询用户信息及分数，没有分数的用户也要显示出来和有分数没有用户对应的数据也显示出来

```
select a.id,a.name,b.age
from user_info a full join user_age b
on a.id=b.id;
```

```
select a.id,a.name,b.age
from user_info a full join user_age b
on a.id=b.id;
```



	id	name	age
1	1	zhangsan	30
2	2	lisi	29
3	3	wangwu	<null>
4	<null>	<null>	31

4.8 排序操作

4.8.1 全局排序

order by 全局排序，只有一个reduce

- ASC（ascend）:升序（默认）
- DESC（descend）:降序

> ORDER BY 子句在SELECT语句的结尾。

案例1:查询员工的薪水，并按照薪水升序排列

```
select id,name,salary
from emp
order by salary asc;
```

```
select id,name,salary
from emp
order by salary asc;
```

8 rows | DDL | CSV

	id	name	salary
1	5	briup	800
2	7	zhangsan	2400
3	2	jake	2500
4	1	lisi	3000
5	8	lulu	3500
6	3	tom	3500
7	4	lili	4000

等价于:

```
select id,name,salary
from emp
order by salary;
```

```
select id,name,salary
from emp
order by salary;
```

8 rows | DDL | CSV

	id	name	salary
1	5	briup	800
2	7	zhangsan	2400
3	2	jake	2500
4	1	lisi	3000
5	8	lulu	3500
6	3	tom	3500
7	4	lili	4000

等价于:

```
select id,name,salary
from emp
order by 3;
```

```
select id,name,salary
from emp
order by 3;
```

	id	name	salary
1	5	briup	800
2	7	zhangsan	2400
3	2	jake	2500
4	1	lisi	3000
5	8	lulu	3500
6	3	tom	3500
7	4	lili	4000

> order by后面跟数字，数字表示select后面的列，数字从1开始

案例2:查询员工的薪水，并按照薪水降序降序

```
select id,name,salary
from emp
order by salary desc;
```

```
select id,name,salary
from emp
order by salary desc;
```

	id	name	salary
1	6	rose	4300
2	4	lili	4000
3	8	lulu	3500
4	3	tom	3500
5	1	lisi	3000
6	2	jake	2500
7	7	zhangsan	2400
8	5	briup	800

案例3:查询员工的薪水，并按照薪水降序降序,薪水相同按照名字升序排序

```
select id,name,salary
from emp
order by salary desc,name asc;
```

```
select id,name,salary
from emp
order by salary desc,name asc;
```

	id	name	salary
1	6	rose	4300
2	4	lili	4000
3	8	lulu	3500
4	3	tom	3500
5	1	lisi	3000
6	2	jake	2500
7	7	zhangsan	2400
8	5	briup	800

4.8.2 局部排序

Sort By：每个MapReduce内部进行排序，对全局结果集来说不是排序。

1. 设置reduce个数

```
set mapreduce.job.reduces=3;
```

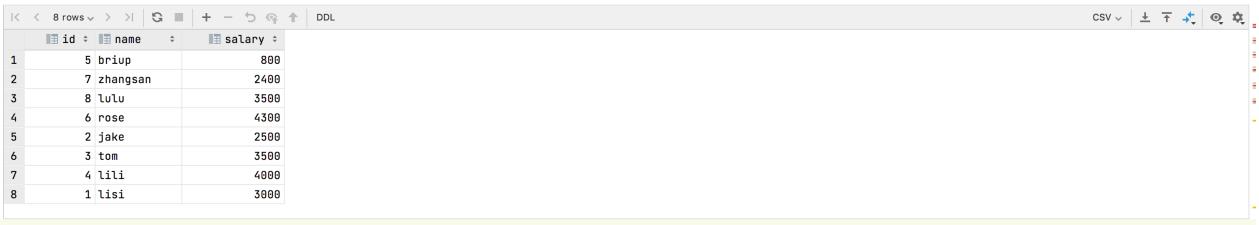
2. 查看设置reduce个数

```
set mapreduce.job.reduces;
```

案例1:查询员工的薪水，并按照薪水降序排列

```
select id,name,salary
from emp
sort by salary asc;
```

```
select id,name,salary
from emp
sort by salary asc;
```



	id	name	salary
1	5	briup	800
2	7	zhangsan	2400
3	8	lulu	3500
4	6	rose	4300
5	2	jake	2500
6	3	tom	3500
7	4	lili	4000
8	1	lisi	3000

案例2:查询员工的薪水，并按照薪水降序排列,将查询结果导入到文件中

```
insert overwrite local directory '/home/hdfs/sql'
select id,name,salary
from emp
sort by salary desc;
```

结果文件：

```
ls /home/hdfs/sql
```

```
hdfs@master:~$ ls /home/hdfs/sql/
000000_0  000001_0  000002_0
```

内容如下如下:

```
hdfs@master:~$ ls /home/hdfs/sql/
000000_0  000001_0  000002_0
hdfs@master:~$ cat /home/hdfs/sql/000000_0
6rose4300.0
8lulu3500.0
7zhangsan2400.0
5briup800.0
hdfs@master:~$ cat /home/hdfs/sql/000001_0
4lili4000.0
3tom3500.0
2jake2500.0
hdfs@master:~$ cat /home/hdfs/sql/000002_0
1lisi3000.0
```

> select前可以指定分隔符,row format delimited fields terminated by '#'

4.8.3 分区排序

Distribute By：类似MR中partition，进行分区，结合sort by使用。

> Hive要求DISTRIBUTE BY语句要写在SORT BY语句之前。
>
> 对于distribute by进行测试，一定要分配多reduce进行处理，否则无法看到distribute by的效果。

案例1: 基于员工编号分区，再按照薪水进行降序排序

1. 设置reduce的个数，将我们对应的id划分到对应的reduce当中去

```
set mapreduce.job.reduces=2;
```

2. 通过distribute by 进行数据的分区

```
insert overwrite local directory '/home/hdfs/sql1'
row format delimited fields terminated by '#'
select id,name,salary
from emp
distribute by id
sort by salary desc;
```

结果文件:

```
ls /home/hdfs/sql1/
```

```
hdfs@master:~$ ls /home/hdfs/sql1/
000000_0  000001_0
```

内容如下如下:

```
hdfs@master:~$ cat /home/hdfs/sql1/000000_0
6#rose#4300.0
4#lili#4000.0
8#lulu#3500.0
2#jake#2500.0
hdfs@master:~$ cat /home/hdfs/sql1/000001_0
3#tom#3500.0
1#lisi#3000.0
7#zhangsan#2400.0
5#briup#800.0
```

4.8.4 CLUSTER BY

当distribute by和sort by字段相同时，可以使用cluster by方式。

cluster by除了具有distribute by的功能外还兼具sort by的功能。但是排序只能是升序排序，不能指定排序规则为ASC或者DESC。

案例1: 基于员工编号分区，再按照薪水进行降序倒序

1. 设置reduce的个数，将我们对应的id划分到对应的reduce当中去

```
set mapreduce.job.reduces=2;
```

2. 通过distribute by 进行数据的分区

```
insert overwrite local directory '/home/hdfs/sql2'
select id,name,salary
from emp
distribute by id
sort by id;
```

结果文件:

```
ls /home/hdfs/sql2
```

```
hdfs@master:~$ ls /home/hdfs/sql2/
000000_0  000001_0
```

内容如下如下:

```
hdfs@master:~$ cat /home/hdfs/sql2/000000_0
2jake2500.0
4lili4000.0
6rose4300.0
8lulu3500.0
hdfs@master:~$ cat /home/hdfs/sql2/000001_0
1lisi3000.0
3tom3500.0
5briup800.0
7zhangsan2400.0_
```

等价于:

```
insert overwrite local directory '/home/hdfs/sql3'
select id,name,salary
from emp
cluster by id;
```

结果文件:

```
ls /home/hdfs/sql3
```

```
hdfs@master:~$ ls /home/hdfs/sql3
000000_0  000001_0
```

内容如下如下:

```
hdfs@master:~$ cat /home/hdfs/sql3/000000_0
2jake2500.0
4lili4000.0
6rose4300.0
8lulu3500.0
hdfs@master:~$ cat /home/hdfs/sql3/000001_0
1lisi3000.0
3tom3500.0
5briup800.0
7zhangsan2400.0
```

4.9 高阶函数

4.9.1 侧视图

Lateral View 是一种特殊的语法，用于搭配UDTF类型函数一起使用，用于解决UDTF函数的一些查询限制问题，

案例1:列出tab_array中的所有数据

```
select * from tab_array;
```

```
select * from tab_array;
```

7 rows | DDL | CSV

	id	arr
1	6	["zhansan","lisi","wangwu","briup"]
2	4	["hive","hbase","ELK"]
3	7	["hbase","elk","hadoop"]
4	1	["java","scala","python"]
5	2	["hadoop","hive","hbase"]
6	3	["spring","springMVC","Mybatis"]
7	4	["Html","css","JavaScript","jquery"]

案例2:列出tab_map中的所有数据

```
select * from tab_map;
```

```
select * from tab_map;
```

5 rows

	name	info
1	lili	{"id":"1","name":"lisi"}
2	briup	{"id":"21","phone":"15098278990"}
3	lisi	{"id":"10","age":"30","sex":"male"}
4	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}
5	lili	{"id":"12","addr":"kunshan"}

案例3:将tab_array中的array数组行转列

```
select explode(arr) as nw from tab_array;
```

```
select explode(arr) as nw from tab_array;
```

23 rows

	nw
1	zhansan
2	lisi
3	wangwu
4	briup
5	hive
6	hbase
7	ELK
8	hbase
9	elk
10	hadoop
11	java
12	scala
13	python
14	hadoop
15	hive
16	hbase
17	spring
18	springMVC
19	Mybatis
20	Html
21	css
22	JavaScript
23	jquery

案例4:将tab_map中的info键值数据转列

```
select explode(info) from tab_map;
```

```
select explode(info) from tab_map;
```

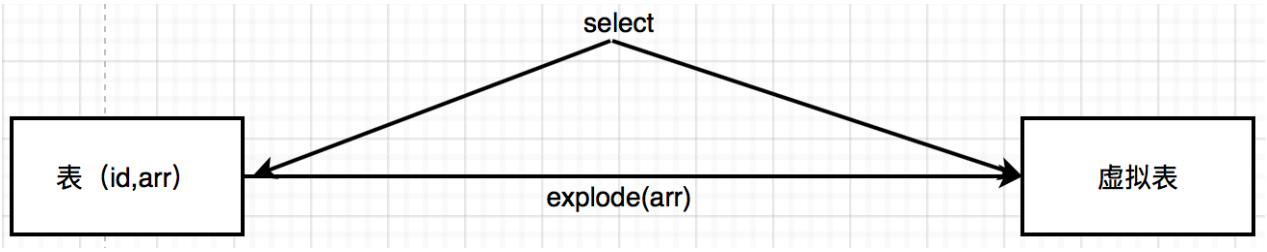
13 rows

	key	value
4	phone	15098278990
5	id	10
6	age	30
7	sex	male
8	id	11
9	age	33
10	sex	female

案例5:列出tab_array中的所有数据的同时显示id列

```
select id,explode(arr) as nw from tab_array;
```

```
Error while compiling statement: FAILED: SemanticException [Error 10081]: UDTF's are not supported outside the SELECT clause, nor nested in expressions
```



- explode函数属于UDTF表生成函数，explode执行返回的结果可以理解为一张虚拟表；
- select直接查询表中原始数据或查询explode生成的虚拟表页没有关系，但是不能继查原表和和虚拟

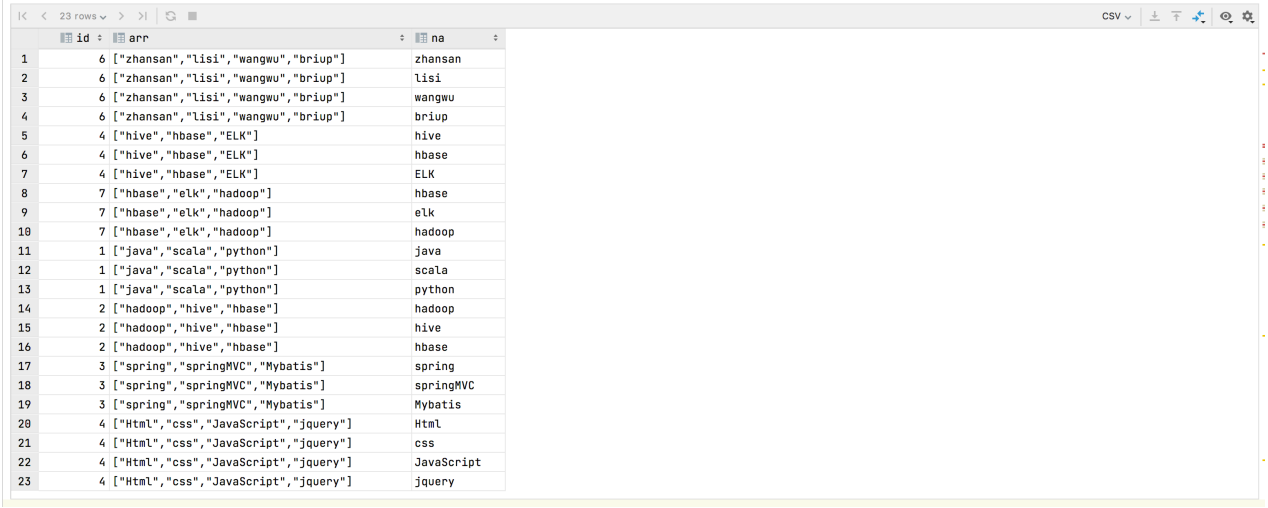
表

- select 查询数据的数据来源为原表和虚拟表，可以使用join关联查询，同时hive中提供了lateral view侧视图，等价于join操作

解决方案:

```
select a.id,a.arr,b.na
from tab_array a lateral view explode(arr) b as na;
```

```
select a.id,a.arr,b.na
from tab_array a lateral view explode(arr) b as na;
```



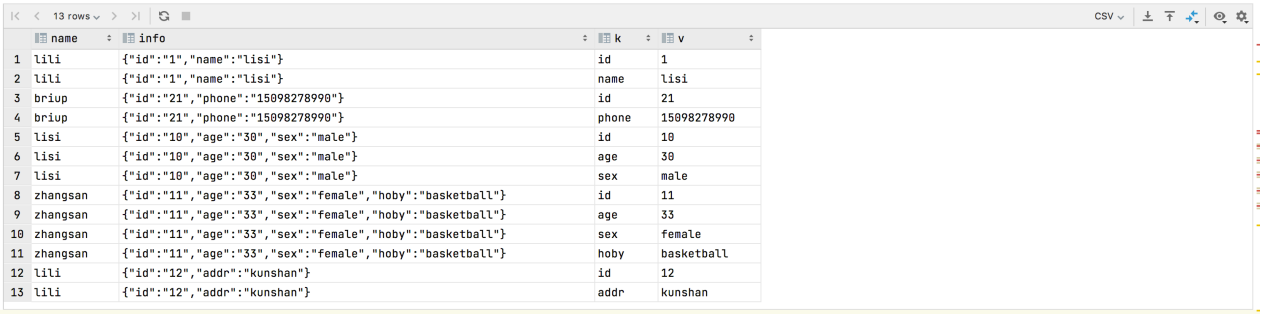
	id	arr	na
1	6	["zhansan","lisi","wangwu","briup"]	zhansan
2	6	["zhansan","lisi","wangwu","briup"]	lisi
3	6	["zhansan","lisi","wangwu","briup"]	wangwu
4	6	["zhansan","lisi","wangwu","briup"]	briup
5	4	["hive","hbase","ELK"]	hive
6	4	["hive","hbase","ELK"]	hbase
7	4	["hive","hbase","ELK"]	ELK
8	7	["hbase","elk","hadoop"]	hbase
9	7	["hbase","elk","hadoop"]	elk
10	7	["hbase","elk","hadoop"]	hadoop
11	1	["java","scala","python"]	java
12	1	["java","scala","python"]	scala
13	1	["java","scala","python"]	python
14	2	["hadoop","hive","hbase"]	hadoop
15	2	["hadoop","hive","hbase"]	hive
16	2	["hadoop","hive","hbase"]	hbase
17	3	["spring","springMVC","Mybatis"]	spring
18	3	["spring","springMVC","Mybatis"]	springMVC
19	3	["spring","springMVC","Mybatis"]	Mybatis
20	4	["Html","css","JavaScript","jquery"]	Html
21	4	["Html","css","JavaScript","jquery"]	css
22	4	["Html","css","JavaScript","jquery"]	JavaScript
23	4	["Html","css","JavaScript","jquery"]	jquery

> 通过lateral view 连接数据后可以继续使用group by、limit、order by 等

案例5：列出tab_map中的所有数据

```
select a.name,b.k,b.v
from tab_map a lateral view explode(info) b as k,v
```

```
select a.name,a.info,b.k,b.v
from tab_map a lateral view explode(info) b as k,v
```



	name	info	k	v
1	lili	{"id":"1","name":"lisi"}	id	1
2	lili	{"id":"1","name":"lisi"}	name	lisi
3	briup	{"id":"21","phone":"15098278990"}	id	21
4	briup	{"id":"21","phone":"15098278990"}	phone	15098278990
5	lisi	{"id":"10","age":"30","sex":"male"}	id	10
6	lisi	{"id":"10","age":"30","sex":"male"}	age	30
7	lisi	{"id":"10","age":"30","sex":"male"}	sex	male
8	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}	id	11
9	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}	age	33
10	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}	sex	female
11	zhangsan	{"id":"11","age":"33","sex":"female","hoby":"basketball"}	hoby	basketball
12	lili	{"id":"12","addr":"kunshan"}	id	12
13	lili	{"id":"12","addr":"kunshan"}	addr	kunshan

> k，v代表explode 转化之后行中多列的名字

案例6:查询device信息表，并把json字段中的数据提取出来

创建表

```
create table devices(
 id int,
 device string
)
row format delimited fields terminated by '#';
```

创建文件

```
vi devices.txt
```

内容如下:

```
1#{"dev_id":"YC-180425-
1007","dev_name":"Kingston","dev_exception":"324","dev_personLiable":"briup"}
1#{"dev_id":"YC-180425-
1008","dev_name":"Kingston","dev_exception":"329","dev_personLiable":"jake"}
```

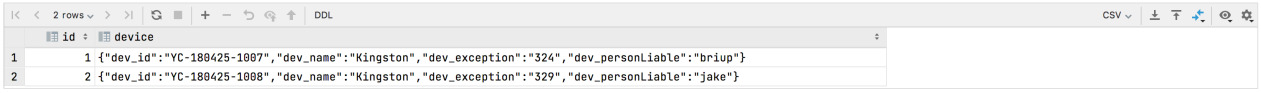
加载数据:

```
load data local inpath '/home/hdfs/devices.txt' into table devices;
```

查询设备信息:

```
select * from devices;
```

```
select * from devices;
```

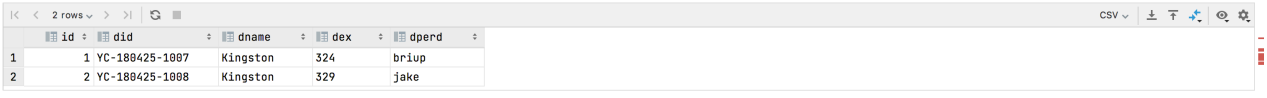


	id	device
1	1	{"dev_id":"YC-180425-1007","dev_name":"Kingston","dev_exception":"324","dev_personLiable":"briup"}
2	2	{"dev_id":"YC-180425-1008","dev_name":"Kingston","dev_exception":"329","dev_personLiable":"jake"}

查询device信息表，并把json字段中的数据提取出来

```
select d.id,b.did,b.dname,b.dex,b.dperd
from devices d
    lateral view
json_tuple(device,"dev_id","dev_name","dev_exception","dev_personLiable") b as
did,dname,dex,dperd;
```

```
select d.id,b.did,b.dname,b.dex,b.dperd
from devices d
    lateral view json_tuple(device,"dev_id","dev_name","dev_exception","dev_personLiable") b as did,dname,dex,dperd;
```



	id	did	dname	dex	dperd
1	1	YC-180425-1007	Kingston	324	briup
2	2	YC-180425-1008	Kingston	329	jake

案例7:如果侧视图为空，也要把tab_array数据查询出来

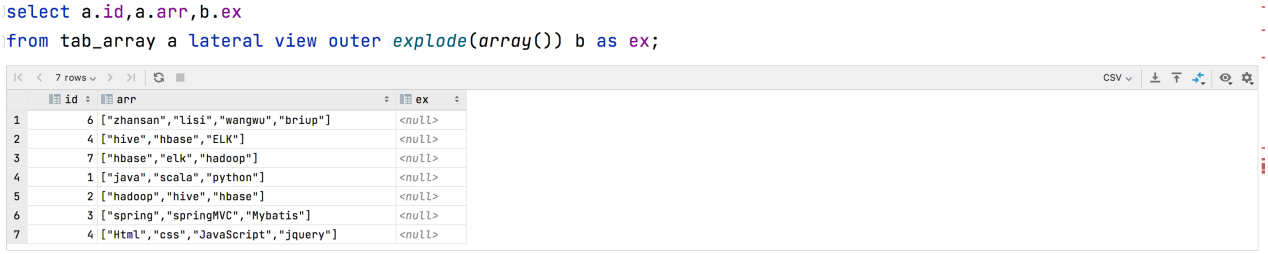
```
select a.id,a.arr,b.ex
from tab_array a lateral view explode(array()) b as ex;
```

```
select a.id,a.arr,b.ex
from tab_array a lateral view explode(array()) b as ex;
```

id	arr	ex

解决方案：如果侧视图中没有数据也要显示原表内容，使用lateral view outer

```
select a.id,a.arr,b.ex
from tab_array a lateral view outer explode(array()) b as ex;
```



	id	arr	ex
1	6	["zhansan","lisi","wangwu","briup"]	<null>
2	4	["hive","hbase","ELK"]	<null>
3	7	["hbase","elk","hadoop"]	<null>
4	1	["java","scala","python"]	<null>
5	2	["hadoop","hive","hbase"]	<null>
6	3	["spring","springMVC","Mybatis"]	<null>
7	4	["Html","css","JavaScript","jquery"]	<null>

4.9.2 增强聚合

增强聚合主要用于多维数据分析模式中，

其他多维度数据分析中的纬度指的是分析看待问题的纬度、角度。增强聚合包括:grouping_sets、cube、rollup

创建表

```
create table session_info(
  month string,
  day string,
  hour string,
  cookieid string
)
row format delimited fields terminated by ',';
```

创建文件

```
vi session.txt
```

内容如下:

```
3,3-3,2,cookie1
3,3-4,2,cookie
3,3-5,3,cookie4
3,3-6,7,cookie
3,3-7,10,cokie5
3,3-8,3,cookie
3,3-9,13,cookie6
3,3-10,5,cookie4
3,3-11,2,cookie5
3,3-12,6,cookie6
3,3-13,5,cookie
4,4-5,3,cookie4
4,4-6,7,cookie5
4,4-7,10,cokie6
4,4-8,3,cookie7
4,4-9,13,cookie1
3,3-3,2,cookie2
3,3-4,2,cookie2
```

```
3,3-5,3,cookie2
3,3-6,7,cookie2
4,4-7,10,cokie3
4,4-8,3,cookie7
4,4-9,13,cookie8
4,4-10,5,cookie9
4,4-11,2,cookie3
4,4-12,6,cookie10
```

加载数据

```
load data local inpath '/home/hdfs/session.txt' into table session_info;
```

案例1:查询用户访问信息

```
select * from session_info;
```

```
select * from session_info;
```

26 rows | DDL | CSV

	month	day	hour	cookieid
1	3	3-3	2	cookie1
2	3	3-4	2	cookie
3	3	3-5	3	cookie4
4	3	3-6	7	cookie
5	3	3-7	10	cokie5
6	3	3-8	3	cookie
7	3	3-9	13	cookie6
8	3	3-10	5	cookie4
9	3	3-11	2	cookie5
10	3	3-12	6	cookie6
11	3	3-13	5	cookie
12	4	4-5	3	cookie4
13	4	4-6	7	cookie5
14	4	4-7	10	cokie6
15	4	4-8	3	cookie7
16	4	4-9	13	cookie1
17	3	3-3	2	cookie2
18	3	3-4	2	cookie2
19	3	3-5	3	cookie2
20	3	3-6	7	cookie2
21	4	4-7	10	cokie3
22	4	4-8	3	cookie7
23	4	4-9	13	cookie8
24	4	4-10	5	cookie9
25	4	4-11	2	cookie3
26	4	4-12	6	cookie10

1. grouping sets是一种将多个group by写在一个sql语句的简易写法，相当于将多个不同纬度group by结果进行union all.

案列1:查询每个月访问的人数

```
select month,null as day,count(distinct cookieid) pv,1 as grouping_id
from session_info
group by month;
```

```
select month,null as day,count(distinct cookieid) pv,1 as grouping_id
from session_info
group by month;
```

2 rows | CSV

	month	day	pv	grouping_id
1	3	<null>	7	1
2	4	<null>	10	1

案例2:查询每天访问的人数

```
select null as month,day,count(distinct cookieid) pv,2 as grouping_id
from session_info
group by day;
```

```
select null as month,day,count(distinct cookieid) pv,2 as grouping_id
from session_info
group by day;
```

19 rows

	month	day	pv	grouping_id
1	<null>	3-10	1	2
2	<null>	3-11	1	2
3	<null>	3-12	1	2
4	<null>	3-13	1	2
5	<null>	3-3	2	2
6	<null>	3-4	2	2
7	<null>	3-5	2	2
8	<null>	3-6	2	2
9	<null>	3-7	1	2
10	<null>	3-8	1	2
11	<null>	3-9	1	2
12	<null>	4-10	1	2
13	<null>	4-11	1	2
14	<null>	4-12	1	2
15	<null>	4-5	1	2
16	<null>	4-6	1	2
17	<null>	4-7	2	2
18	<null>	4-8	1	2
19	<null>	4-9	2	2

案例3:查询每月及每天的访问人数

```
select month,day,count(distinct cookieid) pv,grouping__id
from session_info
group by month,day
grouping sets(month,day)
order by grouping_id;
```

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
grouping sets((month),(day))
order by GROUPING__ID;
```

21 rows

	month	day	pv	grouping__id
1	4	<null>	10	1
2	3	<null>	7	1
3	<null>	4-6	1	2
4	<null>	4-11	1	2
5	<null>	3-9	1	2
6	<null>	3-7	1	2
7	<null>	3-5	2	2
8	<null>	3-3	2	2
9	<null>	3-12	1	2
10	<null>	3-10	1	2
11	<null>	3-11	1	2
12	<null>	4-9	2	2
13	<null>	4-7	2	2
14	<null>	4-5	1	2
15	<null>	4-12	1	2
16	<null>	4-10	1	2
17	<null>	3-8	1	2
18	<null>	3-6	2	2
19	<null>	3-4	2	2
20	<null>	3-13	1	2
21	<null>	4-8	1	2

> grouping__id组的编号

等价于:

```
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day;
```

```
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day;
```

21 rows

	month	day	pv	grouping__id
1	<null>	3-10	1	2
2	<null>	3-11	1	2
3	<null>	3-12	1	2
4	<null>	3-13	1	2
5	<null>	3-3	2	2
6	<null>	3-4	2	2
7	<null>	3-5	2	2
8	<null>	3-6	2	2
9	<null>	3-7	1	2
10	<null>	3-8	1	2
11	<null>	3-9	1	2
12	<null>	4-10	1	2
13	<null>	4-11	1	2
14	<null>	4-12	1	2
15	<null>	4-5	1	2
16	<null>	4-6	1	2
17	<null>	4-7	2	2
18	<null>	4-8	1	2
19	<null>	4-9	2	2
20	3	<null>	7	1
21	4	<null>	10	1

案例4:查询每月、每天的访问人数及每月每天的访客人数。

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
grouping sets(month,day,(month,day))
order by GROUPING__ID;
```

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
grouping sets(month,day,(month,day))
order by GROUPING__ID;
```

40 rows

	month	day	pv	grouping__id
1	4	4-8	1	0
2	4	4-6	1	0
3	4	4-11	1	0
4	3	3-8	1	0
5	3	3-6	2	0
6	3	3-4	2	0
7	3	3-13	1	0
8	3	3-11	1	0
9	4	4-9	2	0
10	4	4-7	2	0
11	4	4-5	1	0
12	4	4-12	1	0
13	4	4-10	1	0
14	3	3-9	1	0
15	3	3-7	1	0
16	3	3-5	2	0
17	3	3-3	2	0
18	3	3-12	1	0
19	3	3-10	1	0
20	3	<null>	7	1
21	4	<null>	10	1

22	<null>	4-9	2	2
23	<null>	4-7	2	2
24	<null>	4-5	1	2
25	<null>	4-12	1	2
26	<null>	4-10	1	2
27	<null>	3-8	1	2
28	<null>	3-6	2	2
29	<null>	3-4	2	2
30	<null>	3-13	1	2
31	<null>	4-8	1	2
32	<null>	4-6	1	2
33	<null>	4-11	1	2
34	<null>	3-9	1	2
35	<null>	3-7	1	2
36	<null>	3-5	2	2
37	<null>	3-3	2	2
38	<null>	3-12	1	2
39	<null>	3-10	1	2
40	<null>	3-11	1	2

或

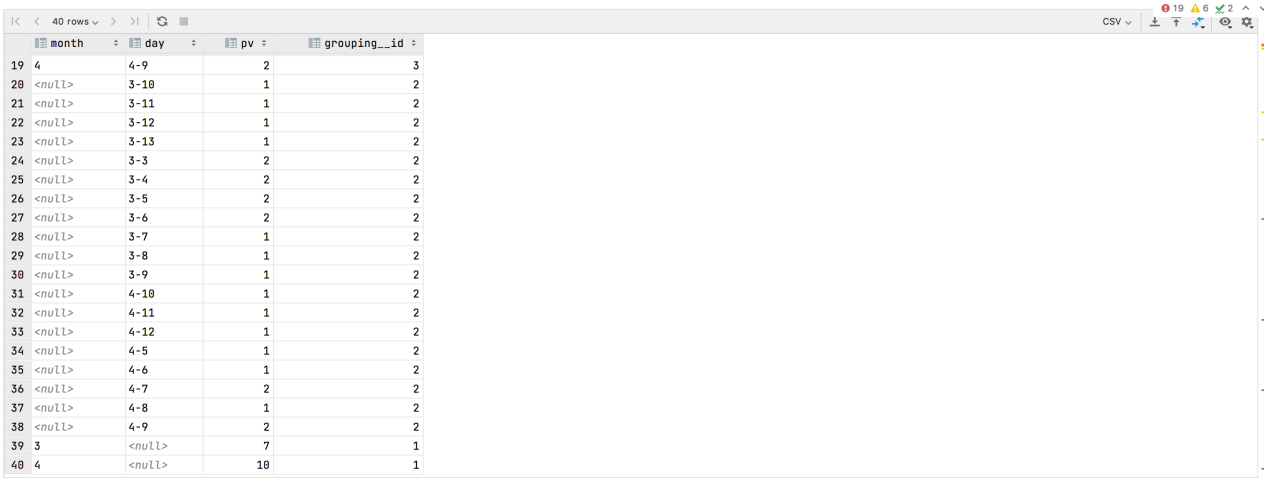
```sql
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day
union all
select month,day,count(distinct cookieid) pv,3 as grouping__id
from session_info
group by month,day;
```

```sql
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day
union all
select month,day,count(distinct cookieid) pv,3 as grouping__id
from session_info
group by month,day;
```

40 rows

CSV

	month	day	pv	grouping__id
1	3	3-10	1	3
2	3	3-11	1	3
3	3	3-12	1	3
4	3	3-13	1	3
5	3	3-3	2	3
6	3	3-4	2	3
7	3	3-5	2	3
8	3	3-6	2	3
9	3	3-7	1	3
10	3	3-8	1	3
11	3	3-9	1	3
12	4	4-10	1	3
13	4	4-11	1	3
14	4	4-12	1	3
15	4	4-5	1	3
16	4	4-6	1	3
17	4	4-7	2	3
18	4	4-8	1	3
19	4	4-9	2	3



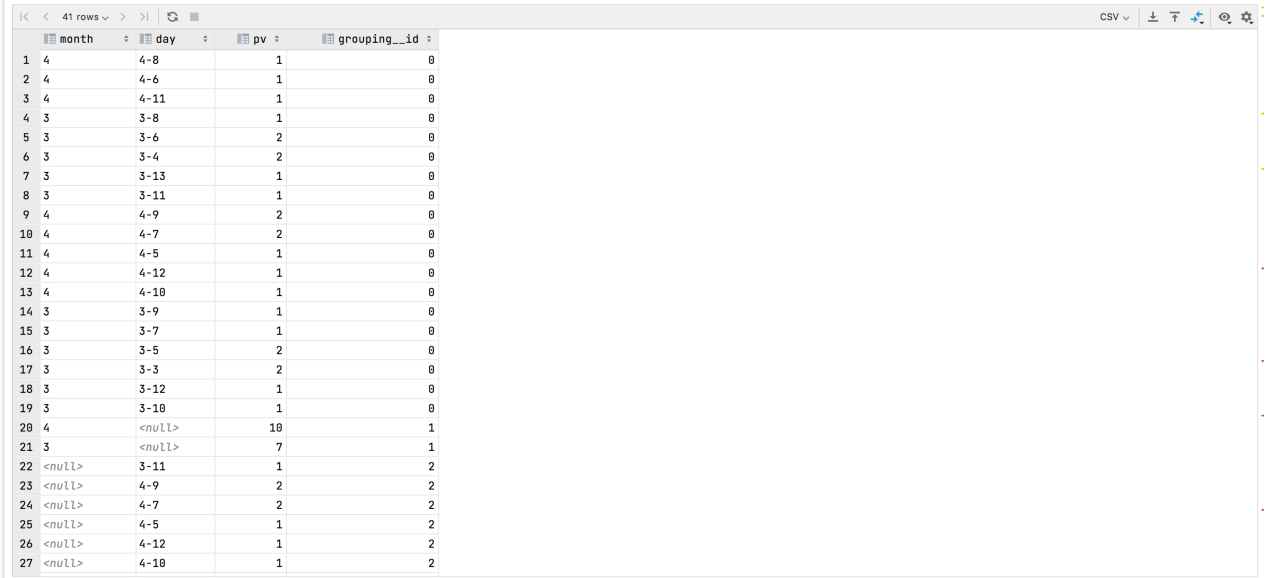
	month	day	pv	grouping__id
19	4	4-9	2	3
20	<null>	3-10	1	2
21	<null>	3-11	1	2
22	<null>	3-12	1	2
23	<null>	3-13	1	2
24	<null>	3-3	2	2
25	<null>	3-4	2	2
26	<null>	3-5	2	2
27	<null>	3-6	2	2
28	<null>	3-7	1	2
29	<null>	3-8	1	2
30	<null>	3-9	1	2
31	<null>	4-10	1	2
32	<null>	4-11	1	2
33	<null>	4-12	1	2
34	<null>	4-5	1	2
35	<null>	4-6	1	2
36	<null>	4-7	2	2
37	<null>	4-8	1	2
38	<null>	4-9	2	2
39	3	<null>	7	1
40	4	<null>	10	1

2. Cube函数表示根据group by 的纬度所有组合进行聚合，对于数据集来说，如果有n个纬度，则所有的组合总个数是2的n次方。例如：现有数据集3个a、b、c纬度,则组成的集合:（a、b、c）、（a、b）、（a、c）（b、c）、（a）、（b）、（c）、().

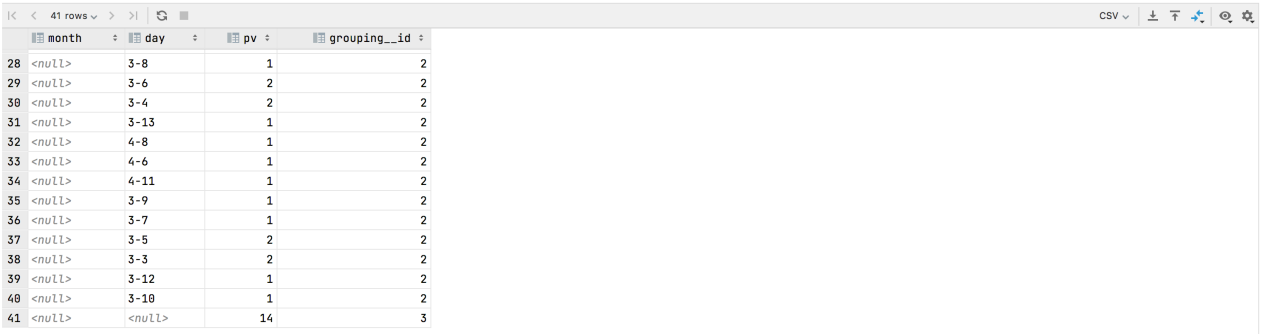
案例1:查询每月、每天各种组合的访问人数

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
with cube
order by GROUPING__ID;
```

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
with cube
order by GROUPING__ID;
```



	month	day	pv	grouping__id
1	4	4-8	1	0
2	4	4-6	1	0
3	4	4-11	1	0
4	3	3-8	1	0
5	3	3-6	2	0
6	3	3-4	2	0
7	3	3-13	1	0
8	3	3-11	1	0
9	4	4-9	2	0
10	4	4-7	2	0
11	4	4-5	1	0
12	4	4-12	1	0
13	4	4-10	1	0
14	3	3-9	1	0
15	3	3-7	1	0
16	3	3-5	2	0
17	3	3-3	2	0
18	3	3-12	1	0
19	3	3-10	1	0
20	4	<null>	10	1
21	3	<null>	7	1
22	<null>	3-11	1	2
23	<null>	4-9	2	2
24	<null>	4-7	2	2
25	<null>	4-5	1	2
26	<null>	4-12	1	2
27	<null>	4-10	1	2

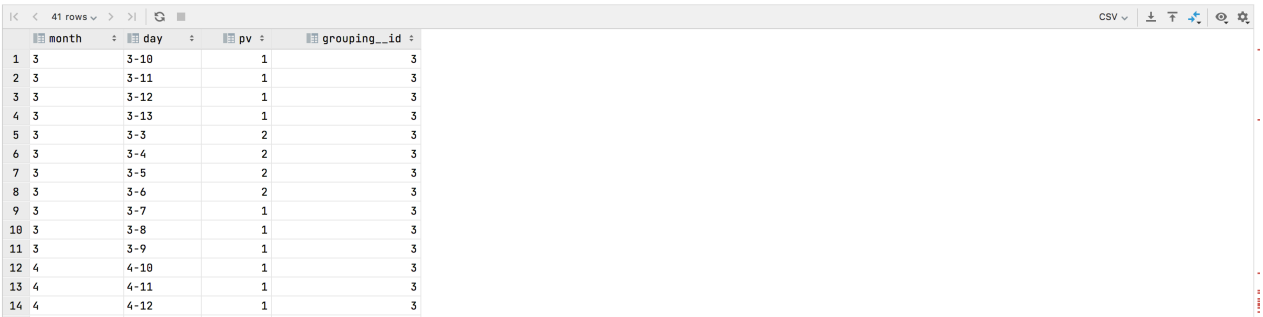


	month	day	pv	grouping__id
28	<null>	3-8	1	2
29	<null>	3-6	2	2
30	<null>	3-4	2	2
31	<null>	3-13	1	2
32	<null>	4-8	1	2
33	<null>	4-6	1	2
34	<null>	4-11	1	2
35	<null>	3-9	1	2
36	<null>	3-7	1	2
37	<null>	3-5	2	2
38	<null>	3-3	2	2
39	<null>	3-12	1	2
40	<null>	3-10	1	2
41	<null>	<null>	14	3

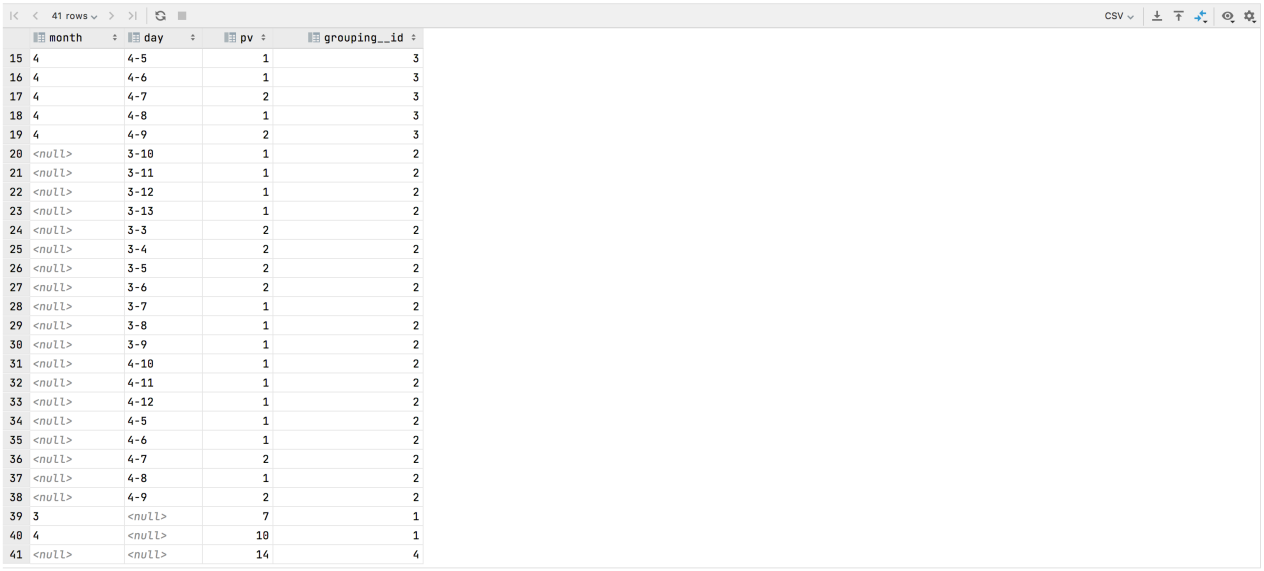
或

```
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day
union all
select month,day,count(distinct cookieid) pv,3 as grouping__id
from session_info
group by month,day
union all
select null as month ,null as day,count(distinct cookieid) pv,4 as grouping__id
from session_info;
```

```
select month,null as day,count(distinct cookieid) pv,1 as grouping__id
from session_info
group by month
union all
select null as month,day,count(distinct cookieid) pv,2 as grouping__id
from session_info
group by day
union all
select month,day,count(distinct cookieid) pv,3 as grouping__id
from session_info
group by month,day
union all
select null as month ,null as day,count(distinct cookieid) pv,4 as grouping__id
from session_info;
```



	month	day	pv	grouping__id
1	3	3-10	1	3
2	3	3-11	1	3
3	3	3-12	1	3
4	3	3-13	1	3
5	3	3-3	2	3
6	3	3-4	2	3
7	3	3-5	2	3
8	3	3-6	2	3
9	3	3-7	1	3
10	3	3-8	1	3
11	3	3-9	1	3
12	4	4-10	1	3
13	4	4-11	1	3
14	4	4-12	1	3



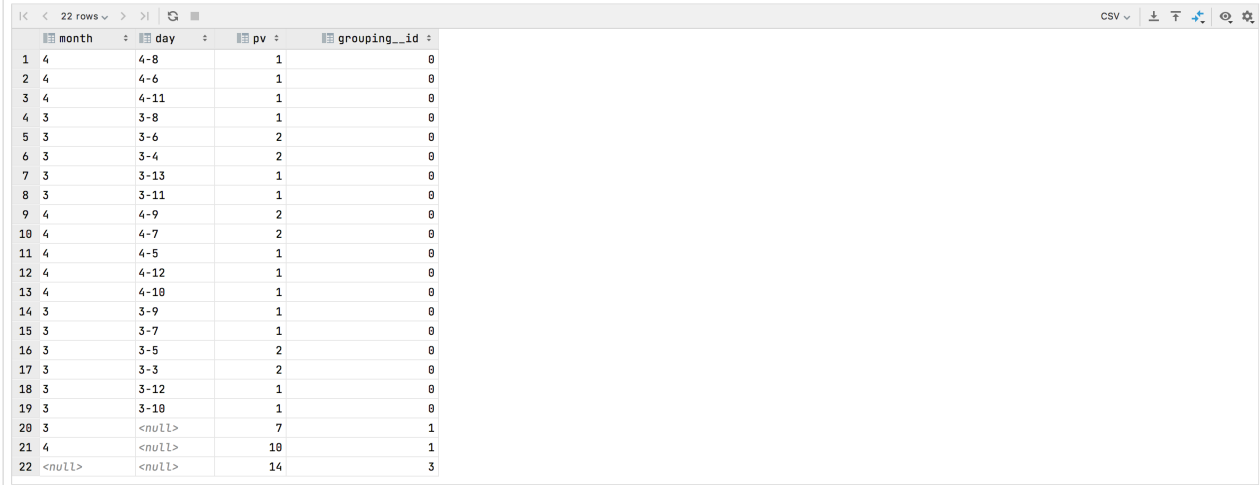
	month	day	pv	grouping__id
15	4	4-5	1	3
16	4	4-6	1	3
17	4	4-7	2	3
18	4	4-8	1	3
19	4	4-9	2	3
20	<null>	3-10	1	2
21	<null>	3-11	1	2
22	<null>	3-12	1	2
23	<null>	3-13	1	2
24	<null>	3-3	2	2
25	<null>	3-4	2	2
26	<null>	3-5	2	2
27	<null>	3-6	2	2
28	<null>	3-7	1	2
29	<null>	3-8	1	2
30	<null>	3-9	1	2
31	<null>	4-10	1	2
32	<null>	4-11	1	2
33	<null>	4-12	1	2
34	<null>	4-5	1	2
35	<null>	4-6	1	2
36	<null>	4-7	2	2
37	<null>	4-8	1	2
38	<null>	4-9	2	2
39	3	<null>	7	1
40	4	<null>	10	1
41	<null>	<null>	14	4

3. rollup是对group by的纬度的组合进行聚合，是cube的子集，以左边的纬度为主，例如：有a、b、c3纬度，则组合为:(a、b、c)、(a、b)、(a)、 ()

案例1:基于月、天按照月的组合查询出来数据

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
with rollup
order by GROUPING__ID;
```

```
select month,day,count(distinct cookieid) pv,GROUPING__ID
from session_info
group by month,day
with rollup
order by GROUPING__ID;
```



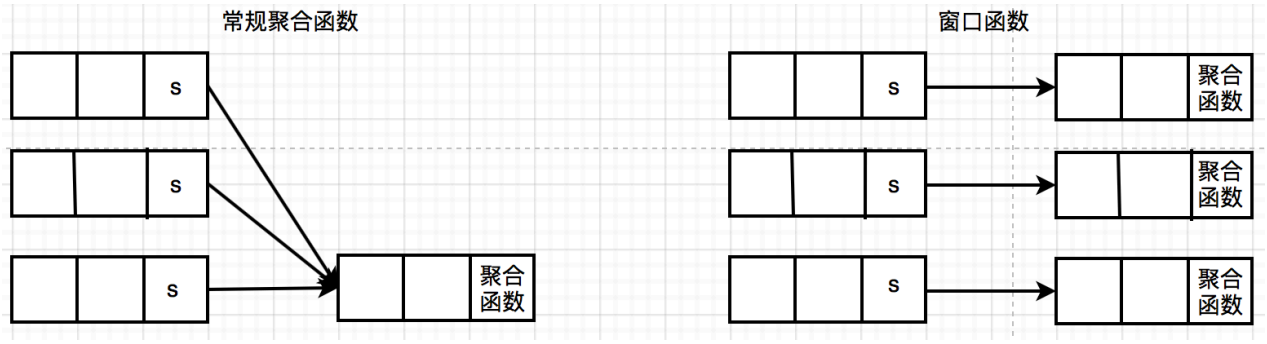
	month	day	pv	grouping__id
1	4	4-8	1	0
2	4	4-6	1	0
3	4	4-11	1	0
4	3	3-8	1	0
5	3	3-6	2	0
6	3	3-4	2	0
7	3	3-13	1	0
8	3	3-11	1	0
9	4	4-9	2	0
10	4	4-7	2	0
11	4	4-5	1	0
12	4	4-12	1	0
13	4	4-10	1	0
14	3	3-9	1	0
15	3	3-7	1	0
16	3	3-5	2	0
17	3	3-3	2	0
18	3	3-12	1	0
19	3	3-10	1	0
20	3	<null>	7	1
21	4	<null>	10	1
22	<null>	<null>	14	3

> 对于经常使用的列，放在左边

4.9.3 窗口函数

4.9.3.1 概念案例

窗口函数(Window functions)也叫做开窗函数OLAP，输入值是SELECT查询语句结果集中的一行或多行窗口中获取，select语句中具有over字句，则其是窗口函数。group by 子句组合的聚合函数会隐藏正在聚合的各个行，最终输出一行，窗口函数聚合之后可以访问当中的各个行，并且可以将这些行中的某些属性添加到结果集中。



窗口函数语法：

```
function(arg1...) over ([partition by (column_name...)][order by ....]
[<window_expression>])
```

- function(arg1...):

  聚合函数:sum、max、avg等

  排序函数:rank、row_number等

  分析函数:lead、lag等

- over partition by column_name... 指定按照分组的列，每个分组可以把它叫做窗口，partition by 没有，表示整张表
- order by... 指定每个分组数据排序规则，累积聚合，支持asc、desc
- window_expression 用于指定每个窗口中操作的数据范围，默认窗口中所有的行

案例1:查询所有订单表数据

```
select * from order_1;
```

```
select * from order_1;
```

8 rows | DDL | CSV

	id	name	addr	salary
1	1	jake	昆山	3000
2	2	briup	上海	3500
3	3	tom	昆山	2600
4	4	kevin	上海	5000
5	5	lili	昆山	12000
6	6	rose	昆山	3400
7	7	lucy	上海	8000
8	8	vens	昆山	2700

案例2:查询每个地区金额总和

```
select addr,sum(salary) from order_1 group by addr;
```

```
select addr,sum(salary) from order_1 group by addr;
```

2 rows | CSV

	addr	_c1
1	上海	16500
2	昆山	23700

案例3:查询每个地区金额总和

```
select id,name,addr,salary,sum(salary) over partition by addr from order_1;
```

```
select id,name,addr,salary,sum(salary) over(partition by addr) as sum_sal from order_1;
```

8 rows

	id	name	addr	salary	sum_sal
1	7	lucy	上海	8000	16500
2	4	kevin	上海	5000	16500
3	2	briup	上海	3500	16500
4	8	vens	昆山	2700	23700
5	6	rose	昆山	3400	23700
6	5	lili	昆山	12000	23700
7	3	tom	昆山	2600	23700

案例4:查询金额总和

```
select id,name,addr,salary,sum(salary) over() as sum_salary from order_1;
```

```
select id,name,addr,salary,sum(salary) over() as sum_salary from order_1;
```

8 rows

	id	name	addr	salary	sum_salary
1	8	vens	昆山	2700	40200
2	7	lucy	上海	8000	40200
3	6	rose	昆山	3400	40200
4	5	lili	昆山	12000	40200
5	4	kevin	上海	5000	40200
6	3	tom	昆山	2600	40200
7	2	briup	上海	3500	40200

案例5:基于名字排序，累加计算金额

```
select id,name,addr,salary,sum(salary) over(order by name) as sum_sal from order_1;
```

```
select id,name,addr,salary,sum(salary) over(order by name) as sum_sal from order_1;
```

8 rows

	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	3500
2	1	jake	昆山	3000	6500
3	4	kevin	上海	5000	11500
4	5	lili	昆山	12000	23500
5	7	lucy	上海	8000	31500
6	6	rose	昆山	3400	34900
7	3	tom	昆山	2600	37500
8	8	vens	昆山	2700	40200

案例6:基于名字排序，按照每个地区计算金额累加求和

```
select id,name,addr,salary,sum(salary) over(partition by addr order by name) as
sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary) over(partition by addr order by name) as sum_sal
from order_1;
```

8 rows

	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	3500
2	4	kevin	上海	5000	8500
3	7	lucy	上海	8000	16500
4	1	jake	昆山	3000	3000
5	5	lili	昆山	12000	15000
6	6	rose	昆山	3400	18400
7	3	tom	昆山	2600	21000
8	8	vens	昆山	2700	23700

4.9.3.2 窗口表达式

在一个聚合操作中，既有partition by同时也有order by的情况下进行累积聚合，默认累积聚合行为从第一行聚合到选中组的最后一行；Window expression窗口表达式给我们提供了一种控制行为范围的能力，如向后3行，向前3行；

语法：

```
rows between [选项]
```

选项：

- preceding 向前
- following 向后
- current row 当前行
- unbounded 边界
- unbounded preceding 从前面的起点，即从第一行开始
- unbounded following 到后面的终点，即最后一行

案列1:基于名字排序，按照每个地区计算金额累加求和，边界从第一行到当前行

```
select id,name,addr,salary,sum(salary)
over(partition by addr order by name rows between unbounded preceding and current row)
as sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between unbounded preceding and current row) as sum_sal
from order_1;
```

8 rows

	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	3500
2	4	kevin	上海	5000	8500
3	7	lucy	上海	8000	16500
4	1	jake	昆山	3000	3000
5	5	lili	昆山	12000	15000
6	6	rose	昆山	3400	18400
7	3	tom	昆山	2600	21000
8	8	vens	昆山	2700	23700

案列1:基于名字排序，按照每个地区计算金额累加求和，边界当前行前一行开始累计

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between 1 preceding and current row) as
sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between 1 preceding and current row) as sum_sal
from order_1;
```

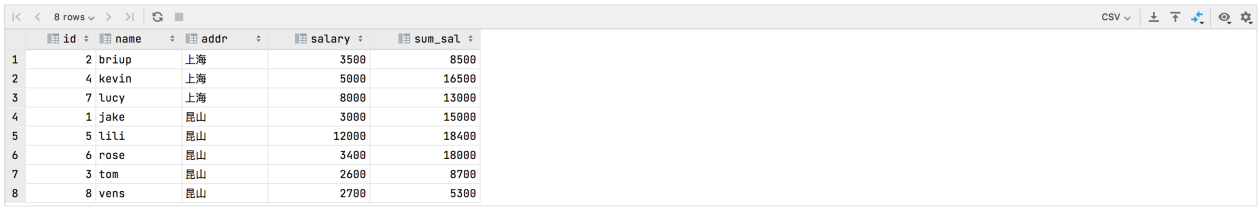
8 rows

	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	3500
2	4	kevin	上海	5000	8500
3	7	lucy	上海	8000	13000
4	1	jake	昆山	3000	3000
5	5	lili	昆山	12000	15000
6	6	rose	昆山	3400	15400
7	3	tom	昆山	2600	6000
8	8	vens	昆山	2700	5300

案列2:基于名字排序，按照每个地区计算金额累加求和，边界当前行前一行开始累计到当前后一行

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between 1 preceding and 1 following) as
sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between 1 preceding and 1 following) as sum_sal
from order_1;
```

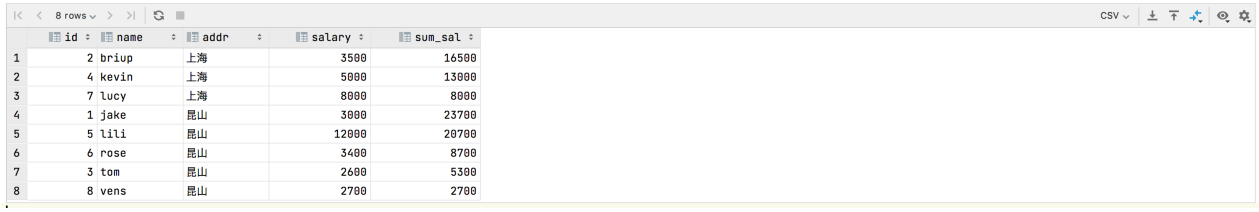


	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	8500
2	4	kevin	上海	5000	16500
3	7	lucy	上海	8000	13000
4	1	jake	昆山	3000	15000
5	5	lili	昆山	12000	18400
6	6	rose	昆山	3400	18000
7	3	tom	昆山	2600	8700
8	8	vens	昆山	2700	5300

案列3:基于名字排序，按照每个地区计算金额累加求和，边界当前行开始累计到最后一行

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between current row  and unbounded
following) as sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between current row  and unbounded following) as sum_sal
from order_1;
```

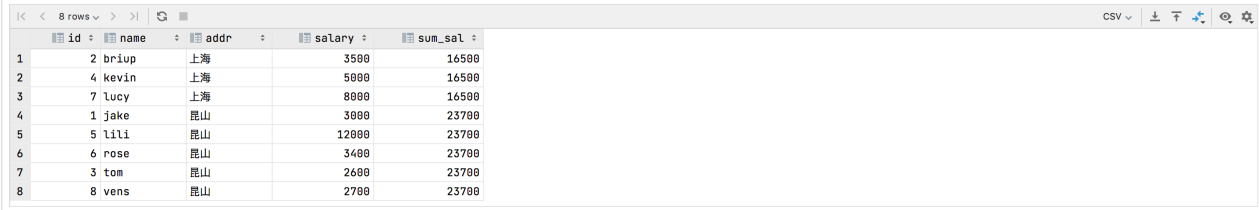


	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	16500
2	4	kevin	上海	5000	13000
3	7	lucy	上海	8000	8000
4	1	jake	昆山	3000	23700
5	5	lili	昆山	12000	20700
6	6	rose	昆山	3400	8700
7	3	tom	昆山	2600	5300
8	8	vens	昆山	2700	2700

案列4:基于名字排序，按照每个地区计算金额累加求和，边界所属组第一行开始累计到最后一行

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between unbounded preceding and
unbounded following) as sum_sal
from order_1;
```

```
select id,name,addr,salary,sum(salary)
    over(partition by addr order by name rows between unbounded preceding and unbounded following) as sum_sal
from order_1;
```



	id	name	addr	salary	sum_sal
1	2	briup	上海	3500	16500
2	4	kevin	上海	5000	16500
3	7	lucy	上海	8000	16500
4	1	jake	昆山	3000	23700
5	5	lili	昆山	12000	23700
6	6	rose	昆山	3400	23700
7	3	tom	昆山	2600	23700
8	8	vens	昆山	2700	23700

4.9.3.3 窗口排序

row_number():每个分组中，为每行分配一个从1开始的唯一序列号，递增，不考虑重复；

rank()：每个分组中，为每行分配一个从1开始的序列号，考虑重复，重复值之后挤占后续位置；

dense_rank():每个分组中，为每行分配一个从1开始的序列号，考虑重复，不挤占后续位置；

ntile(number):将每个分组内的数据分为若干个部分（每一个部分类似于一个桶），且每一个桶分配一个桶编号，如果不能平均分配，则优先分配较小编号的桶，每一个桶能放的行数最多相差1，在计算过程中可以选择桶中的一部分。

创建表

```
create table web_info(
  cookieid string,
  ct string,
  pv int,
  url string
)
row format delimited fields terminated by ',';
```

创建文件

```
vi web.txt
```

内容如下:

```
cookie1,2022-03-10,1,url1
cookie1,2022-03-11,5,url2
cookie1,2022-03-12,7,url3
cookie1,2022-03-13,3,url4
cookie1,2022-03-14,2,url5
cookie1,2022-03-15,4,url6
cookie1,2022-03-16,4,url7
cookie2,2022-03-10,2,url2
cookie2,2022-03-11,3,url1
cookie2,2022-03-12,5,url3
cookie2,2022-03-13,6,url4
cookie2,2022-03-14,3,url6
cookie2,2022-03-15,9,url5
cookie2,2022-03-16,7,url7
```

加载数据

```
load data local inpath '/home/hdfs/web.txt' into table web_info;
```

案例1:查询所有会话信息。

```
select * from web_info;
```

```
select * from web_info;
```

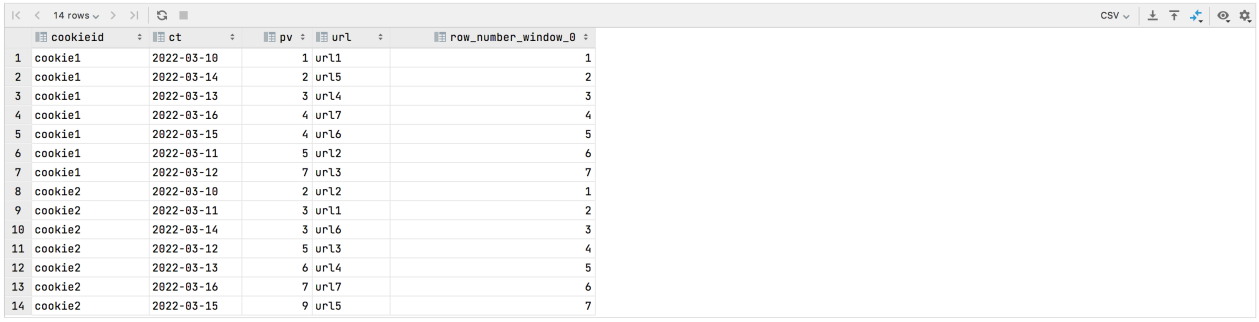
14 rows | DDL | CSV

	cookieid	ct	pv	url
1	cookie1	2022-03-10	1	url1
2	cookie1	2022-03-11	5	url2
3	cookie1	2022-03-12	7	url3
4	cookie1	2022-03-13	3	url4
5	cookie1	2022-03-14	2	url5
6	cookie1	2022-03-15	4	url6
7	cookie1	2022-03-16	4	url7
8	cookie2	2022-03-10	2	url2
9	cookie2	2022-03-11	3	url1
10	cookie2	2022-03-12	5	url3
11	cookie2	2022-03-13	6	url4
12	cookie2	2022-03-14	3	url6
13	cookie2	2022-03-15	9	url5
14	cookie2	2022-03-16	7	url7

案例2:按照会话分组，基于访问次数排序,不考虑重复；

```
select cookieid,ct,pv,url,row_number() over(partition by cookieid order by pv)
from web_info;
```

```
select cookieid,ct,pv,url,row_number() over(partition by cookieid order by pv)
from web_info;
```

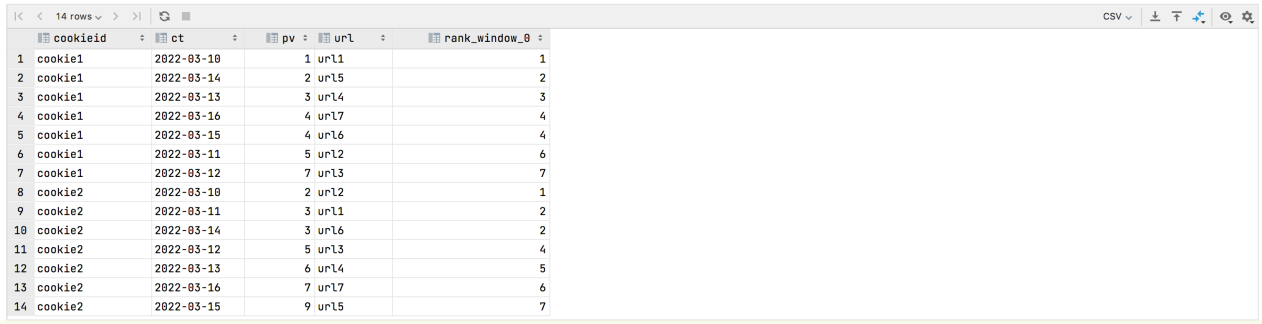


	cookieid	ct	pv	url	row_number_window_0
1	cookie1	2022-03-10	1	url1	1
2	cookie1	2022-03-14	2	url5	2
3	cookie1	2022-03-13	3	url4	3
4	cookie1	2022-03-16	4	url7	4
5	cookie1	2022-03-15	4	url6	5
6	cookie1	2022-03-11	5	url2	6
7	cookie1	2022-03-12	7	url3	7
8	cookie2	2022-03-10	2	url2	1
9	cookie2	2022-03-11	3	url1	2
10	cookie2	2022-03-14	3	url6	3
11	cookie2	2022-03-12	5	url3	4
12	cookie2	2022-03-13	6	url4	5
13	cookie2	2022-03-16	7	url7	6
14	cookie2	2022-03-15	9	url5	7

案例3:按照会话分组，基于访问次数排序,考虑重复，重复计出后面序号；

```
select cookieid,ct,pv,url,rank() over(partition by cookieid order by pv)
from web_info;
```

```
select cookieid,ct,pv,url,rank() over(partition by cookieid order by pv)
from web_info;
```

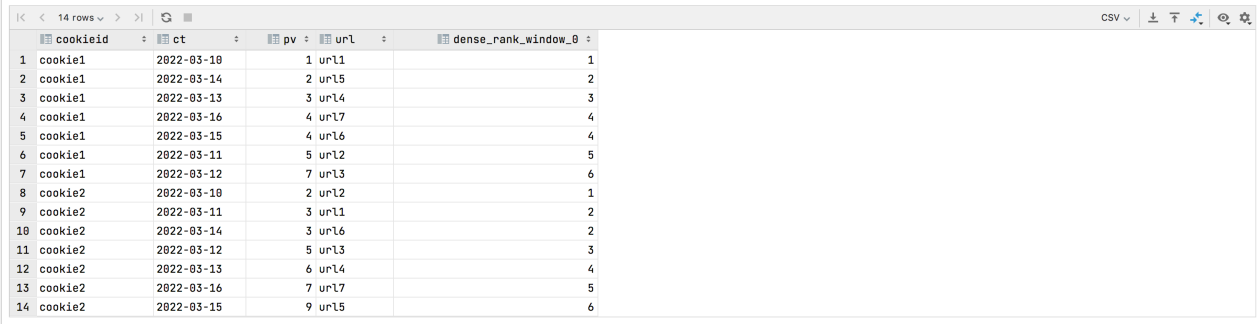


	cookieid	ct	pv	url	rank_window_0
1	cookie1	2022-03-10	1	url1	1
2	cookie1	2022-03-14	2	url5	2
3	cookie1	2022-03-13	3	url4	3
4	cookie1	2022-03-16	4	url7	4
5	cookie1	2022-03-15	4	url6	4
6	cookie1	2022-03-11	5	url2	6
7	cookie1	2022-03-12	7	url3	7
8	cookie2	2022-03-10	2	url2	1
9	cookie2	2022-03-11	3	url1	2
10	cookie2	2022-03-14	3	url6	2
11	cookie2	2022-03-12	5	url3	4
12	cookie2	2022-03-13	6	url4	5
13	cookie2	2022-03-16	7	url7	6
14	cookie2	2022-03-15	9	url5	7

案例4:按照会话分组，基于访问次数排序,考虑重复；重复不计出后面序号；

```
select cookieid,ct,pv,url,dense_rank() over(partition by cookieid order by pv)
from web_info;
```

```
select cookieid,ct,pv,url,dense_rank() over(partition by cookieid order by pv)
from web_info;
```

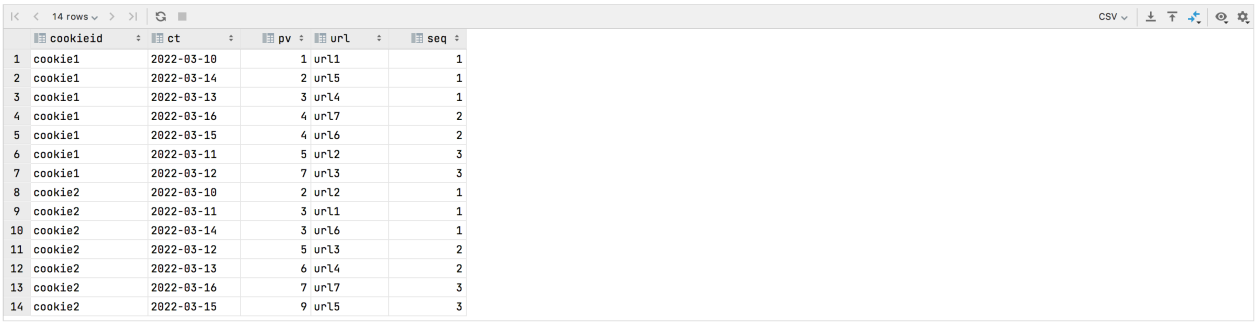


	cookieid	ct	pv	url	dense_rank_window_0
1	cookie1	2022-03-10	1	url1	1
2	cookie1	2022-03-14	2	url5	2
3	cookie1	2022-03-13	3	url4	3
4	cookie1	2022-03-16	4	url7	4
5	cookie1	2022-03-15	4	url6	4
6	cookie1	2022-03-11	5	url2	5
7	cookie1	2022-03-12	7	url3	6
8	cookie2	2022-03-10	2	url2	1
9	cookie2	2022-03-11	3	url1	2
10	cookie2	2022-03-14	3	url6	2
11	cookie2	2022-03-12	5	url3	3
12	cookie2	2022-03-13	6	url4	4
13	cookie2	2022-03-16	7	url7	5
14	cookie2	2022-03-15	9	url5	6

案例5：:按照会话分组，基于访问次数排序，并把每个组的数据分为3个桶数据；

```
select cookieid,ct,pv,url,ntile(3) over(partition by cookieid order by pv) as seq
from web_info;
```

```
select cookieid,ct,pv,url,ntile(3) over(partition by cookieid order by pv) as seq
from web_info;
```



	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	1
2	cookie1	2022-03-14	2	url5	1
3	cookie1	2022-03-13	3	url4	1
4	cookie1	2022-03-16	4	url7	2
5	cookie1	2022-03-15	4	url6	2
6	cookie1	2022-03-11	5	url2	3
7	cookie1	2022-03-12	7	url3	3
8	cookie2	2022-03-10	2	url2	1
9	cookie2	2022-03-11	3	url1	1
10	cookie2	2022-03-14	3	url6	1
11	cookie2	2022-03-12	5	url3	2
12	cookie2	2022-03-13	6	url4	2
13	cookie2	2022-03-16	7	url7	3
14	cookie2	2022-03-15	9	url5	3

4.9.3.4 窗口分析

LAG(col,n,default) 用于统计窗口内往上第n行值

- 第一个参数列名
- 第二个参数往上第n行（可选，默认1）
- 第三个参数为默认值（往上第n行为null值，取默认值，不指定，默认null）

LEAD(col,n,default)用于统计窗口内往下第n行值

- 第一个参数列名
- 第二个参数往下第n行（可选，默认1）
- 第三个参数为默认值（往下第n行为null值，取默认值，不指定，默认null）

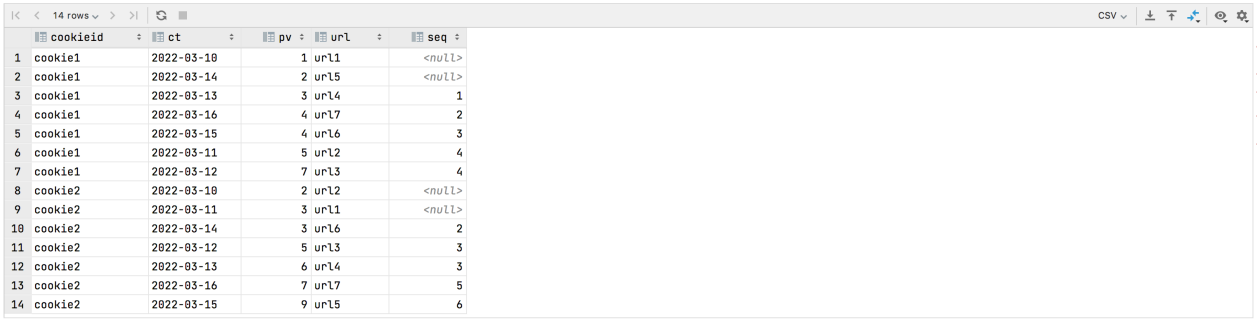
FIRST_VALUE(col)取分组内排序后，截止到当前行，第一个值

LAST_VALUE(col)取分组内排序后，截止到当前行，最后一个值

案例1:按照会话分组，基于访问次数排序，访问次数向上提取2行值；

```
select cookieid,ct,pv,url,LAG(pv,2) over(partition by cookieid order by pv) as seq
from web_info;
```

```
select cookieid,ct,pv,url,lag(pv,2) over(partition by cookieid order by pv) as seq
from web_info;
```

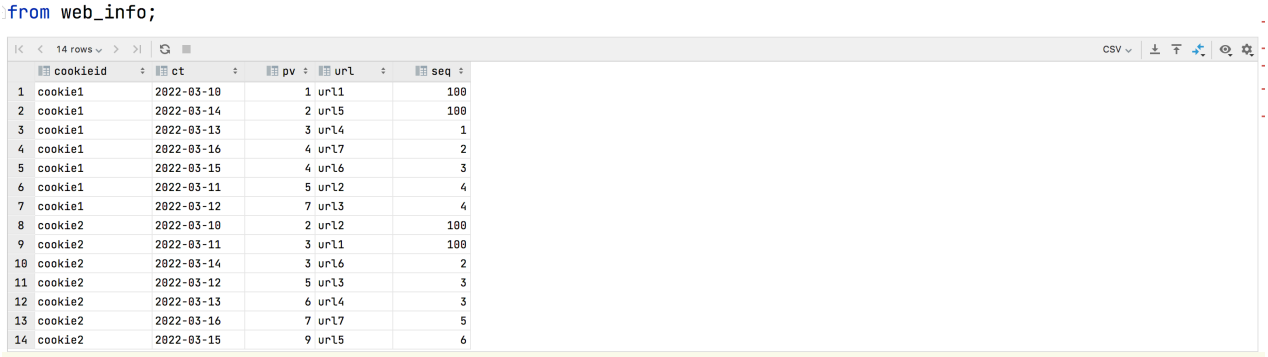


	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	<null>
2	cookie1	2022-03-14	2	url5	<null>
3	cookie1	2022-03-13	3	url4	1
4	cookie1	2022-03-16	4	url7	2
5	cookie1	2022-03-15	4	url6	3
6	cookie1	2022-03-11	5	url2	4
7	cookie1	2022-03-12	7	url3	4
8	cookie2	2022-03-10	2	url2	<null>
9	cookie2	2022-03-11	3	url1	<null>
10	cookie2	2022-03-14	3	url6	2
11	cookie2	2022-03-12	5	url3	3
12	cookie2	2022-03-13	6	url4	3
13	cookie2	2022-03-16	7	url7	5
14	cookie2	2022-03-15	9	url5	6

案例2:按照会话分组，基于访问次数排序，访问次数向上提取2行值，值为null使用100替换；

```
select cookieid,ct,pv,url,LAG(pv,2,100) over(partition by cookieid order by pv) as seq
from web_info;
```

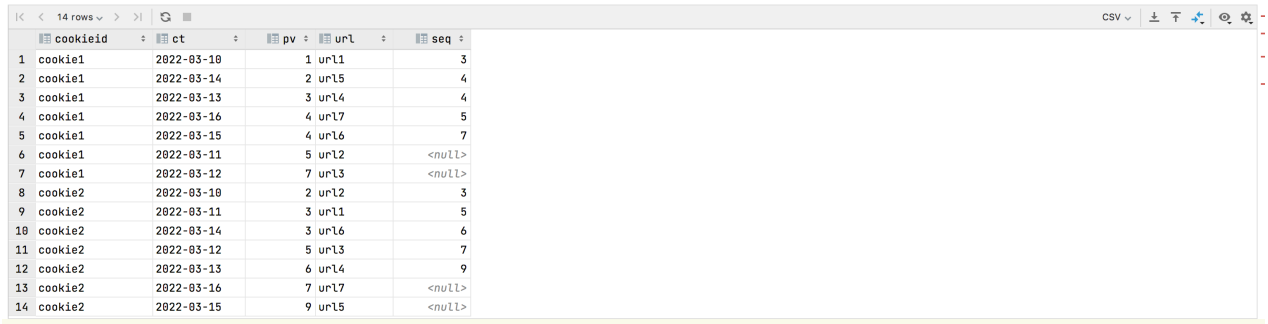
```
select cookieid,ct,pv,url,LAG(pv,2,100) over(partition by cookieid order by pv) as seq
from web_info;
```



	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	100
2	cookie1	2022-03-14	2	url5	100
3	cookie1	2022-03-13	3	url4	1
4	cookie1	2022-03-16	4	url7	2
5	cookie1	2022-03-15	4	url6	3
6	cookie1	2022-03-11	5	url2	4
7	cookie1	2022-03-12	7	url3	4
8	cookie2	2022-03-10	2	url2	100
9	cookie2	2022-03-11	3	url1	100
10	cookie2	2022-03-14	3	url6	2
11	cookie2	2022-03-12	5	url3	3
12	cookie2	2022-03-13	6	url4	3
13	cookie2	2022-03-16	7	url7	5
14	cookie2	2022-03-15	9	url5	6

案例3:按照会话分组，基于访问次数排序，访问次数向下提取2行值；

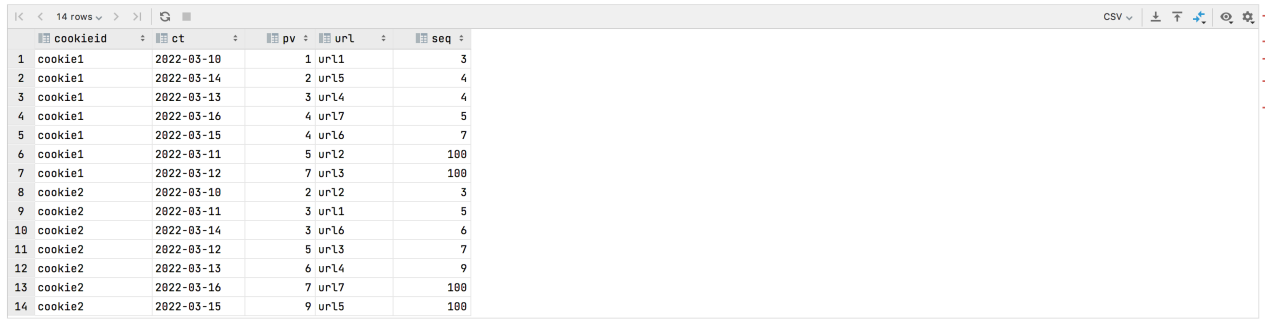
```
select cookieid,ct,pv,url,lead(pv,2) over(partition by cookieid order by pv) as seq
from web_info;
```



	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	3
2	cookie1	2022-03-14	2	url5	4
3	cookie1	2022-03-13	3	url4	4
4	cookie1	2022-03-16	4	url7	5
5	cookie1	2022-03-15	4	url6	7
6	cookie1	2022-03-11	5	url2	<null>
7	cookie1	2022-03-12	7	url3	<null>
8	cookie2	2022-03-10	2	url2	3
9	cookie2	2022-03-11	3	url1	5
10	cookie2	2022-03-14	3	url6	6
11	cookie2	2022-03-12	5	url3	7
12	cookie2	2022-03-13	6	url4	9
13	cookie2	2022-03-16	7	url7	<null>
14	cookie2	2022-03-15	9	url5	<null>

案例4:按照会话分组，基于访问次数排序，访问次数向下提取2行值，值为null使用100替换；

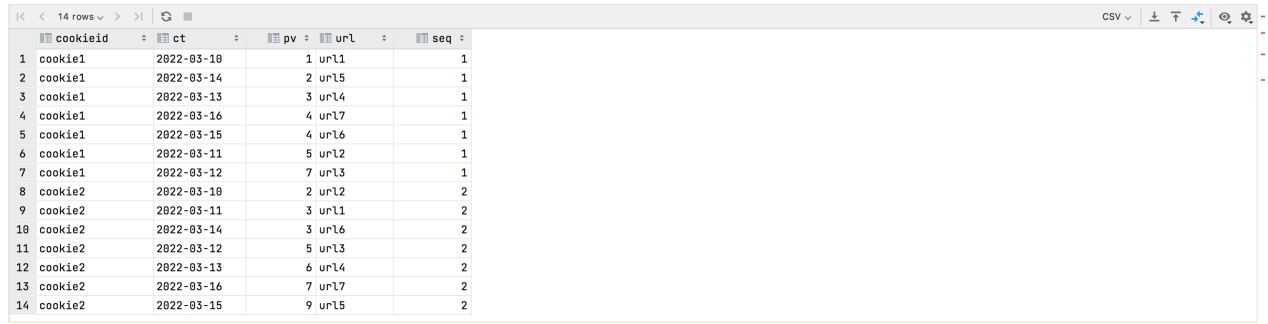
```
select cookieid,ct,pv,url,lead(pv,2,100) over(partition by cookieid order by pv) as seq
from web_info;
```



	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	3
2	cookie1	2022-03-14	2	url5	4
3	cookie1	2022-03-13	3	url4	4
4	cookie1	2022-03-16	4	url7	5
5	cookie1	2022-03-15	4	url6	7
6	cookie1	2022-03-11	5	url2	100
7	cookie1	2022-03-12	7	url3	100
8	cookie2	2022-03-10	2	url2	3
9	cookie2	2022-03-11	3	url1	5
10	cookie2	2022-03-14	3	url6	6
11	cookie2	2022-03-12	5	url3	7
12	cookie2	2022-03-13	6	url4	9
13	cookie2	2022-03-16	7	url7	100
14	cookie2	2022-03-15	9	url5	100

案例5:按照会话分组，基于访问次数排序，取出会话中第一个访问次数拼接给组内其他行；

```
select cookieid,ct,pv,url,first_value(pv) over(partition by cookieid order by pv) as seq
from web_info;
```



	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	1
2	cookie1	2022-03-14	2	url5	1
3	cookie1	2022-03-13	3	url4	1
4	cookie1	2022-03-16	4	url7	1
5	cookie1	2022-03-15	4	url6	1
6	cookie1	2022-03-11	5	url2	1
7	cookie1	2022-03-12	7	url3	1
8	cookie2	2022-03-10	2	url2	2
9	cookie2	2022-03-11	3	url1	2
10	cookie2	2022-03-14	3	url6	2
11	cookie2	2022-03-12	5	url3	2
12	cookie2	2022-03-13	6	url4	2
13	cookie2	2022-03-16	7	url7	2
14	cookie2	2022-03-15	9	url5	2

案例6:按照会话分组，基于访问次数排序，取分组内排序后，截止到当前行，最后一个值；

```
select cookieid,ct,pv,url,last_value(url) over(partition by cookieid order by pv) as
seq
from web_info;
```

```
select cookieid,ct,pv,url,last_value(url) over(partition by cookieid order by pv) as seq
from web_info;
```

14 rows

	cookieid	ct	pv	url	seq
1	cookie1	2022-03-10	1	url1	url1
2	cookie1	2022-03-14	2	url5	url5
3	cookie1	2022-03-13	3	url4	url4
4	cookie1	2022-03-16	4	url7	url6
5	cookie1	2022-03-15	4	url6	url6
6	cookie1	2022-03-11	5	url2	url2
7	cookie1	2022-03-12	7	url3	url3
8	cookie2	2022-03-10	2	url2	url2
9	cookie2	2022-03-11	3	url1	url6
10	cookie2	2022-03-14	3	url6	url6
11	cookie2	2022-03-12	5	url3	url3
12	cookie2	2022-03-13	6	url4	url4
13	cookie2	2022-03-16	7	url7	url7
14	cookie2	2022-03-15	9	url5	url5

4.9.4 抽样函数

在数据分析过程，可能我们不需要所有的数据只需要查询部分数据以便于快速得到数据分析处理结果，获取全部数据中的一部分就叫抽样或采样，方式：

- 随机抽样
- 块采集

4.9.4.1 随机抽样

随机抽样rand()函数随机抽取数据，配合limit来限制抽取数据的个数，优点随机，缺点速度不快，特别是表数据多的时候。

order by语句也可以用于随机获取数据，但是数据量大的情况，可能造成数据热点问题，因为order by全局排序，只能启动一个reducer

案例1:查询员工表中的数据

```
select * from emp;
```

```
select * from emp;
```

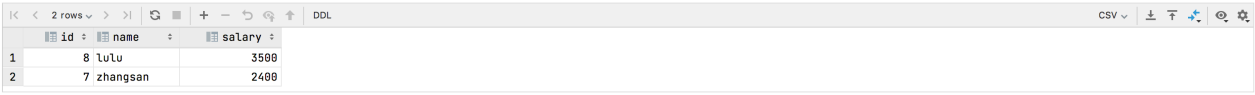
8 rows

	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500
4	4	lili	4000
5	5	briup	800
6	6	rose	4300
7	7	zhangsan	2400
8	8	lulu	3500

案例2:随机抽取两名员工信息

```
select *
from emp
distribute by rand()
sort by rand()
limit 2;
```

```
select *
from emp
distribute by rand()
sort by rand()
limit 2;
```



	id	name	salary
1	8	lulu	3500
2	7	zhangsan	2400

> 推荐使用distribute +sort ， 可以保证数据随机分布在mapper和reducer之间

案例3:随机抽取两名员工信息

```
select *
from emp
order by rand()
limit 2;
```

```
select *
from emp
order by rand()
limit 2;
```

	id	name	salary
1	7	zhangsan	2400
2	2	jake	2500

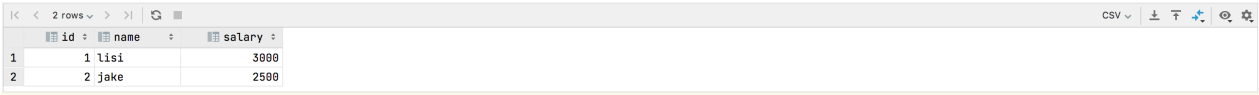
4.9.4.2 块采集

块采样基于表在hdfs文件中存储文件块大小采样，允许按照数据块中行数据、百分比数据、块大小采样，优点速度快，缺点数据不是随机的。

案例1:基于块从员工表中抽取二行数据

```
select * from emp tablesample(2 rows);
```

```
select * from emp tablesample (2 rows);
```

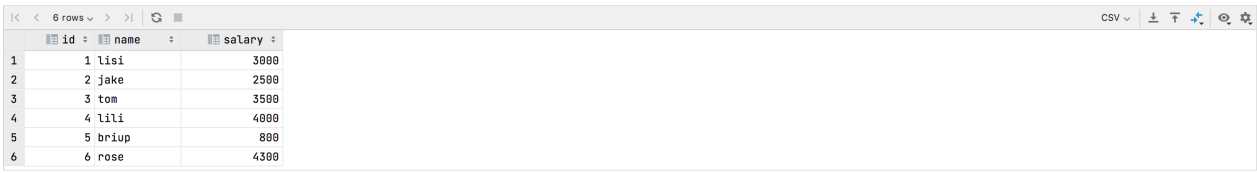


	id	name	salary
1	1	lisi	3000
2	2	jake	2500

案例2:抽取员工表对应数据块的70%

```
select * from emp tablesample (70 percent );
```

```
select * from emp tablesample (70 percent );
```

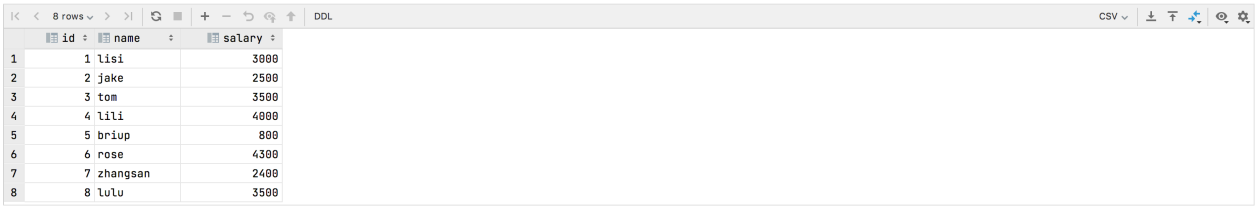


	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500
4	4	lili	4000
5	5	briup	800
6	6	rose	4300

案例3:基于数据块抽取1k员工信息

```
select * from emp tablesample (1k);
```

```
select * from emp tablesample (1K);
```



	id	name	salary
1	1	lisi	3000
2	2	jake	2500
3	3	tom	3500
4	4	lili	4000
5	5	briup	800
6	6	rose	4300
7	7	zhangsan	2400
8	8	lulu	3500

> 支持数据单位：b、k、m、g.....,不区分大小写

4.9.5 URL数据解析

URL的基本组成部分:

```
protolol://Host:path?query
```

- protocol 协议类型
- host 域名
- path 访问路径
- query 参数数据

URL数据案例:

```
http://www.briup.com/index.php/bigdata.html?userid=1&name=tom
```

Hive中为了实现对URL的解析，专门提供了解析URL的函数parse_url、parse_url_tuple;

- parse_url(string urlString, string partToExtract [, string keyToExtract])

  第一个参数URL地址值

  第二个参数查询内容部分，HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE

- parse_url_tuple(string urlStr,string p1,...,string pn)

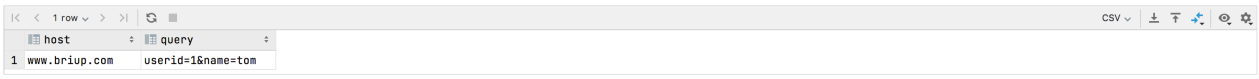
  第一个参数URL地址值

  第二个参数查询内容部分，HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, USERINFO, QUERY

案例1:提取URL中域名、查询条件;

```
select parse_url('http://www.briup.com/index.php/bigdata.html?
userid=1&name=tom','HOST') as host,
    parse_url('http://www.briup.com/index.php/bigdata.html?userid=1&name=tom','QUERY')
as query;
```

```
select parse_url('http://www.briup.com/index.php/bigdata.html?userid=1&name=tom','HOST') as host,
    parse_url('http://www.briup.com/index.php/bigdata.html?userid=1&name=tom','QUERY') as query;
```

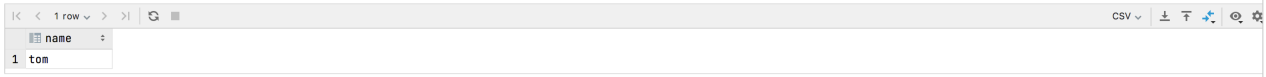


	host	query
1	www.briup.com	userid=1&name=tom

案例2:提取URL中查询条件name;

```
select parse_url('http://www.briup.com/index.php/bigdata.html?
userid=1&name=tom','QUERY','name') as name;
```

```
select parse_url('http://www.briup.com/index.php/bigdata.html?userid=1&name=tom','QUERY','name') as name;
```

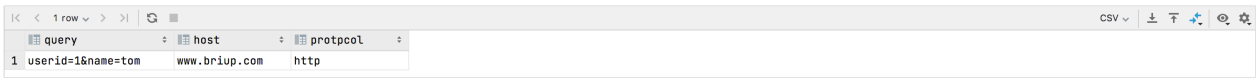


	name
1	tom

案例3:提取URL中查询条件、域名、协议

```
select parse_url_tuple('http://www.briup.com/index.php/bigdata.html?
userid=1&name=tom',
    'QUERY','HOST','PROTOCOL') as(query,host,protpcol);
```

```
select parse_url_tuple('http://www.briup.com/index.php/bigdata.html?userid=1&name=tom',
    'QUERY','HOST','PROTOCOL') as(query,host,protpcol);
```



	query	host	protpcol
1	userid=1&name=tom	www.briup.com	http

> parse_url_tuple是UDTF函数

案例4:查询出tb_url中的所有数据，并提取URL中的查询条件、域名、协议。

创建表

```
create table tb_url(
cookieid string,
pv int,
url string
)
row format delimited fields terminated by ',';
```

创建文件

```
vi url.txt
```

内容如下:

```
cookie,1,http://www.briup.com/index.php/bigdata.html?userid=1&name=tom
cookie1,2,http://www.briup.com/index.php/login.html?userid=2&name=jake
cookie2,5,http://www.briup.com/index.php/register.html?userid=3&name=briup
cookie3,4,http://www.briup.com/index.php/web.html?userid=4&name=wangwu
```

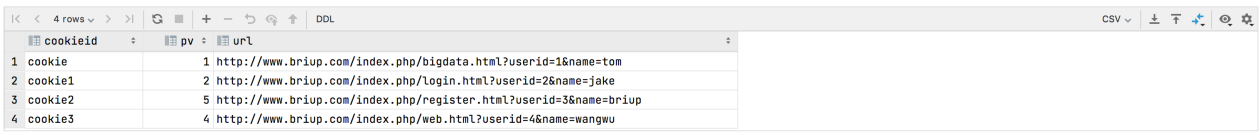
加载数据

```
load data local inpath '/home/hdfs/url.txt' into table tb_url;
```

查询表结果:

```
select * from tb_url;
```

```
select * from tb_url;
```



	cookieid	pv	url
1	cookie	1	http://www.briup.com/index.php/bigdata.html?userid=1&name=tom
2	cookie1	2	http://www.briup.com/index.php/login.html?userid=2&name=jake
3	cookie2	5	http://www.briup.com/index.php/register.html?userid=3&name=briup
4	cookie3	4	http://www.briup.com/index.php/web.html?userid=4&name=wangwu

查询出tb_url中的所有数据，并提取URL中的查询条件、域名、协议

```
select t.cookieid,t.pv,t.url,b.query,b.host,b.protocol
from tb_url t lateral view parse_url_tuple(url,'QUERY','HOST','PROTOCOL') b as
query,host,protocol;
```

```
select t.cookieid,t.pv,t.url,b.query,b.host,b.protocol
from tb_url t lateral view parse_url_tuple(url,'QUERY','HOST','PROTOCOL') b as query,host,protocol;
```

	cookieid	pv	url	query	host	protocol
1	cookie	1	http://www.briup.com/index.php/bigdata.html?userid=1&name=tom	userid=1&name=tom	www.briup.com	http
2	cookie1	2	http://www.briup.com/index.php/login.html?userid=2&name=jake	userid=2&name=jake	www.briup.com	http
3	cookie2	5	http://www.briup.com/index.php/register.html?userid=3&name=briup	userid=3&name=briup	www.briup.com	http
4	cookie3	4	http://www.briup.com/index.php/web.html?userid=4&name=wangwu	userid=4&name=wangwu	www.briup.com	http

5 Hive shell参数

5.1 命令行

语法：

```
hive [-hiveconf x=y]* [-i filename]* [-f filename|-e query-string][-S]
```

-i 从文件初始化HQL

-e 从命令行执行指定的HQL

-f 执行HQL脚本

-v 冗余模式，额外打印出执行的HQL语句

-S 静默模式，指定后不显示执行进度信息，最后只显示结果

-p 连接远程Hive Server的端口号

–hiveconf x=y 在命令行中设置Hive的运行时配置参数，优先级高于hive-site.xml,但低于Hive交互Shell中使用Set命令设置。

案例1:使用sql语句或者sql脚本进行交互

```
hive -e "create database if not exists briup"
```

5.2 参数配置

设定Hive的参数可以调优HQL代码的执行效率，参数设置，有三种设定方式：

- 配置文件
- 命令行参数
- 参数声明

> 优先级依次递增，参数声明>命令行参数>配置文件
>
> 特殊设置必须用配置文件 、命令行参数设定，因为参数的读取在Session建立以前已经完成了，如log4j

5.2.1 配置文件

Hive的配置文件包括：

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

> 用户自定义配置会覆盖默认配置
>
> 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

5.2.2 命令行参数

启动Hive（客户端或Server方式）时，可以在命令行添加-hiveconf param=value来设定参数

案例1:

```
hive -hiveconf hive.root.logger=INFO,console
```

> 设定对本次启动的Session（对于Server方式启动，则是所有请求的Sessions）有效

5.2.3 参数声明

在hive命令进入的终端中使用SET关键字设定参数

案例1:

```
set mapred.reduce.tasks=100;
```

> 作用域也是session级的

6 数据压缩

6.1 压缩编码

Mapreduce支持的压缩编码

压缩格式	工具	算法	文件扩展名	是否可切分
DEFAULT	无	DEFAULT	.deflate	否
Gzip	gzip	DEFAULT	.gz	否
bzip2	bzip2	bzip2	.bz2	是
LZO	lzop	LZO	.lzo	否
LZ4	无	LZ4	.lz4	否
Snappy	无	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

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
LZ4	org.apache.hadoop.io.compress.Lz4Codec
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

> 性能压缩参考 **http://google.github.io/snappy/**

6.2 压缩参数

在Hadoop中启用压缩，可以配置如下参数（mapred-site.xml文件中）

参数	默认值	阶段	建议
io.compression.codecs （在core-site.xml中配置）	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec,org.apache.hadoop.io.compress.Lz4Codec	输入压缩	Hadoop使用文件扩展名判断是否支持某种编解码器
mapreduce.map.output.compress	false	mapper输出	这个参数设为true启用压缩
mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.DefaultCodec	mapper输出	使用LZO、LZ4或snappy编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress	false	reducer输出	这个参数设为true启用压缩
mapreduce.output.fileoutputformat.compress.codec	org.apache.hadoop.io.compress. DefaultCodec	reducer输出	使用标准工具或者编解码器，如gzip和bzip2
mapreduce.output.fileoutputformat.compress.type	RECORD	reducer输出	SequenceFile输出使用的压缩类型：NONE和BLOCK

6.3 Map输出压缩

开启map输出阶段压缩可以减少job中map和Reduce task间数据传输量

1. 开启hive中间传输数据压缩功能

```
set hive.exec.compress.intermediate=true;
```

2. 开启mapreduce中map输出压缩功能

```
set mapreduce.map.output.compress=true;
```

3. 设置mapreduce中map输出数据的压缩方式

```
set mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.BZip2Codec;
```

4. 执行sql语句测试

```
select count(*) as count from stu;
```

```
set hive.exec.compress.intermediate=true;
set mapreduce.map.output.compress=true;
set mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.BZip2Codec;
select count(*) as count from stu;
```

6.4 Reduce输出压缩

Hive将输出写入到表中时，输出内容同样可以进行压缩。属性hive.exec.compress.output控制着这个功能。用户可能需要保持默认设置文件中的默认值false，这样默认的输出就是非压缩的纯文本文件了。用户可以通过在查询语句或执行脚本中设置这个值为true，来开启输出结果压缩功能。

1. 开启hive最终输出数据压缩功能

```
set hive.exec.compress.output=true;
```

2. 开启mapreduce最终输出数据压缩

```
set mapreduce.output.fileoutputformat.compress=true;
```

3. 设置mapreduce最终数据输出压缩方式

```
set mapreduce.output.fileoutputformat.compress.codec =
org.apache.hadoop.io.compress.BZip2Codec;
```

4. 设置mapreduce最终数据输出压缩为块压缩

```
set mapreduce.output.fileoutputformat.compress.type=BLOCK;
```

5. 测试一下输出结果是否是压缩文件

```
insert overwrite local directory '/home/hdfs/sql5'
select id,name,salary
from emp
distribute by id
sort by id;
```

> 使用压缩输出的时候，注意当前hadoop是否对压缩格式支持

6. 查看结果文件

```
ls /home/hdfs/sql5
```

```
hdfs@master:~$ ls /home/hdfs/sql5
000000_0.bz2
```

7 调优

7.1 Fetch抓取

Hive中对某些情况的查询可以不必使用MapReduce计算。如：select * from stu;在这种情况下，Hive可以简单地读取stu对应的存储目录下的文件，然后输出查询结果到控制台。

通过设置hive.fetch.task.conversion参数,可以控制查询语句是否走MapReduce.

- hive.fetch.task.conversion设置成none，然后执行查询语句，都会执行mapreduce程序。
- hive.fetch.task.conversion设置成more，然后执行查询语句，不会执行mapreduce程序。

案例1:执行mapreduce程序

```
set hive.fetch.task.conversion=none;
select * from stu;
```

```
hive> set hive.fetch.task.conversion=none;
hive> select * from stu;
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a d
ifferent execution engine (i.e. spark, tez) or using Hive 1.X releases.
Query ID = hdfs_20220402221915_59e1e171-2516-4932-a6e3-ef398d19c777
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_1648780209911_0019, Tracking URL = http://master:8088/proxy/application_1648780209911_0019/
Kill Command = /opt/hadoop-3.0.3/bin/hadoop job  -kill job_1648780209911_0019
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0
2022-04-02 22:19:34,530 Stage-1 map = 0%,  reduce = 0%
2022-04-02 22:19:45,935 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 1.73 sec
MapReduce Total cumulative CPU time: 1 seconds 730 msec
Ended Job = job_1648780209911_0019
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1   Cumulative CPU: 1.73 sec   HDFS Read: 4083 HDFS Write: 346 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 730 msec
OK
6       zhansan,lisi,wangwu,briup
1       lisi
2       wangwu
Time taken: 31.201 seconds, Fetched: 3 row(s)
```

案例2:不会执行mapreduce程序

```
set hive.fetch.task.conversion=more;
select * from stu;
```

```
hive> set hive.fetch.task.conversion=more;
hive> select * from stu;
OK
6       zhansan,lisi,wangwu,briup
1       lisi
2       wangwu
Time taken: 0.449 seconds, Fetched: 3 row(s)
```

7.2 本地模式

大多数的Hadoop Job是需要Hadoop提供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时消耗可能会比实际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。

用户可以通过设置hive.exec.mode.local.auto的值为true，来让Hive在适当的时候自动启动这个优化。

案例1:开启本地模式

```
#开启本地模式
set hive.exec.mode.local.auto=true;
#设置local mr的最大输入数据量，当输入数据量小于这个值时采用local  mr的方式，默认为134217728，即128M
set hive.exec.mode.local.auto.inputbytes.max=51234560;
#设置local mr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 mr的方式，默认为4
set hive.exec.mode.local.auto.input.files.max=10;
select * from stu cluster by id;
```

案例2:关闭本地模式

```
set hive.exec.mode.local.auto=false;
select * from stu cluster by id;
```

7.3 MapJoin

如果不指定MapJoin或者不符合MapJoin的条件，那么Hive解析器会在Reduce阶段完成join,容易发生数据倾斜。可以用MapJoin把小表全部加载到内存在map端进行join，避免reducer处理。

开启MapJoin参数设置

1. 设置自动选择Mapjoin

```
set hive.auto.convert.join = true;
```

2. 大表小表的阈值设置（默认25M以下认为是小表）：

```
set hive.mapjoin.smalltable.filesize=25123456;
```

3. 有数据倾斜的时候进行负载均衡（默认是false）

```
set hive.groupby.skewindata = true;
```

> 当选项设定为 true，生成的查询计划会有两个MR Job。第一个MR Job中，Map的输出结果会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的Group By Key有可能被分发到不同的Reduce中，从而达到负载均衡的目的；第二个MR Job再根据预处理的数据结果按照Group By Key分布到Reduce中（这个过程可以保证相同的Group By Key被分布到同一个Reduce中），最后完成最终的聚合操作。

7.4 Group By

默认情况下，Map阶段同一Key数据分发给一个reduce，当一个key数据过大时就倾斜了。并不是所有的聚合操作都需要在Reduce端完成，很多聚合操作都可以先在Map端进行部分聚合，最后在Reduce端得出最终结果.

开启Map端聚合参数设置:

1. Map端进行聚合

```
set hive.map.aggr = true;
```

2. 在Map端进行聚合操作的条目数目

```
set hive.groupby.mapaggr.checkinterval = 100000;
```

3. 数据倾斜的时候进行负载均衡（默认是false）

```
set hive.groupby.skewindata = true;
```

> 设定为 true，生成的查询计划会有两个MR Job
>
> 第一个MR Job中，Map的输出结果会随机分布到Reduce中，每个Reduce做部分聚合操作，并输出结果，这样处理的结果是相同的Group By Key有可能被分发到不同的Reduce中，从而达到负载均衡的目的；
>
> 第二个MR Job再根据预处理的数据结果按照Group By Key分布到Reduce中（这个过程可以保证相同的Group By Key被分布到同一个Reduce中），最后完成最终的聚合操作。

7.5 Count(distinct)

数据量小的时候无所谓，数据量大的情况下，由于COUNT DISTINCT操作需要用一个Reduce Task来完成，这一个Reduce需要处理的数据量太大，就会导致整个Job很难完成，一般COUNT DISTINCT使用先GROUP BY再COUNT的方式替换：

```
select count(distinct id) from stu;
select count(id) from (select id from stu group by id) a;
```

> 虽然会多用一个Job来完成，但在数据量大的情况下，这个绝对是值得的。

7.6 笛卡尔积

尽量避免笛卡尔积，即避免join的时候不加on条件，或者无效的on条件，Hive只能使用1个reducer来完成笛卡尔积。

7.7 动态分区调整

往hive分区表中插入数据时，hive提供了一个动态分区功能，其可以基于查询参数的位置去推断分区的名称，从而建立分区。使用Hive的动态分区，需要进行相应的配置。

Hive的动态分区是以第一个表的分区规则，来对应第二个表的分区规则，将第一个表的所有分区，全部拷贝到第二个表中来，第二个表在加载数据的时候，不需要指定分区了，直接用第一个表的分区即可

7.7.1 动态分区参数

1. 开启动态分区功能（默认true，开启）

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

2. 设置为非严格模式（动态分区的模式，默认strict，表示必须指定至少一个分区为静态分区，nonstrict模式表示允许所有的分区字段都可以使用动态分区。）

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

3. 在所有执行MR的节点上，最大一共可以创建多少个动态分区。

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

4. 在每个执行MR的节点上，最大可以创建多少个动态分区。该参数需要根据实际的数据来设定。

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

5. 整个MR Job中，最大可以创建多少个HDFS文件。在linux系统当中，每个linux用户最多可以开启1024个进程，每一个进程最多可以打开2048个文件，即持有2048个文件句柄，下面这个值越大，就可以打开文件句柄越大

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

6. 当有空分区生成时，是否抛出异常。一般不需要设置。

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

案例1:

1. 创建文件内容

```
vi stu_data.txt
```

2. 文件内容如下

```
1,tom
2,jake
3,lisi
4,rose
```

3. 分区表准备

```
create table ori(id int,name string)
partitioned by (day int)
row format delimited fields
terminated by ',';
```

4. 导入ori数据

```
load data local inpath '/home/hdfs/stu_data.txt' into  table ori partition (day=1);
load data local inpath '/home/hdfs/stu_data.txt' into  table ori partition (day=2);
```

5. 创建目标分区表

```
create table ori_target(id int,name string)
partitioned by (day int)
row format delimited fields
terminated by ',';
```

6. 向目标分区放入数据

如果按照之前介绍的往指定一个分区中Insert数据，那么这个需求很不容易实现。这时候就需要使用动态分区来实现。

```
insert overwrite table ori_target partition (day)
select id,name,day
from ori;
```

7. 查看分区

```
show partitions ori_target;
```

```
show partitions ori_target;
```

2 rows CSV

	partition
1	day=1
2	day=2

7.8 并行执行

Hive会将一个查询转化成一个或者多个阶段。这样的阶段可以是MapReduce阶段、抽样阶段、合并阶段、limit阶段。或者Hive执行过程中可能需要的其他阶段。默认情况下，Hive一次只会执行一个阶段。不过，某个特定的job可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个job的执行时间缩短。不过，如果有更多的阶段可以并行执行，那么job可能就越快完成。

通过设置参数hive.exec.parallel值为true，就可以开启并发执行。不过，在共享集群中，需要注意下，如果job中并行阶段增多，那么集群利用率就会增加。

```
#开启任务并行执行
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=16;
```

> 在系统资源比较空闲的时候才有优势，否则，没资源，并行也起不来

7.9 严格模式

Hive提供了一个严格模式，可以防止用户执行那些可能意想不到的不好的影响的查询。

```
#开启严格模式
set hive.mapred.mode = strict;
#开启非严格模式
set hive.mapred.mode = nostrict;
```

开启严格模式可以禁止3种类型的查询。

- 对于分区表，

  在where语句中必须含有分区字段作为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。

- 对于使用了order by语句的查询，要求必须使用limit语句

  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Reducer中进行处理，强制要求用户增加这个LIMIT语句可以防止Reducer额外执行很长一段时间。

- 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

7.10 JVM重用

JVM重用是Hadoop调优参数的内容，其对Hive的性能具有非常大的影响，特别是对于很难避免小文件的场景或task特别多的场景，这类场景大多数执行时间都很短。

Hadoop的默认配置通常是使用派生JVM来执行map和Reduce任务的。这时JVM的启动过程可能会造成相当大的开销，尤其是执行的job包含有成百上千task任务的情况。JVM重用可以使得JVM实例在同一个job中重新使用N次。N的值可以在Hadoop的mapred-site.xml文件中进行配置。通常在10-20之间，具体多少需要根据具体业务场景测试得出。

我们也可以在hive当中通过

```
set  mapred.job.reuse.jvm.num.tasks=10;
```

这个设置来设置我们的jvm重用

这个功能的缺点是，开启JVM重用将一直占用使用到的task插槽，以便进行重用，直到任务完成后才能释放。如果某个“不平衡的”job中有某几个reduce task执行的时间要比其他Reduce task消耗的时间多的多的话，那么保留的插槽就会一直空闲着却无法被其他的job使用，直到所有的task都结束了才会释放。

7.11 推测执行

在分布式集群环境下，因为程序Bug（包括Hadoop本身的bug），负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致，有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop采用了推测执行（Speculative Execution）机制.

根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

设置开启推测执行参数：

```
set mapred.map.tasks.speculative.execution=true
set mapred.reduce.tasks.speculative.execution=true
set hive.mapred.reduce.tasks.speculative.execution=true;
```

关于调优这些推测执行变量，还很难给一个具体的建议。如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的map或者Reduce task的话，那么启动推测执行造成的浪费是非常巨大。

7.12 explain

Hive提供了EXPLAIN命令来展示一个查询的执行计划,这个执行计划对于我们了解底层原理，hive 调优，排查 **数据倾斜** 等很有帮助.

使用语法如下：

```
EXPLAIN [EXTENDED|CBO|AST|DEPENDENCY|AUTHORIZATION|LOCKS|VECTORIZATION|ANALYZE] query
```

explain 后面可以跟以下可选参数，注意：这几个可选参数不是 hive 每个版本都支持的

- EXTENDED：加上 extended 可以输出有关计划的额外信息。这通常是物理信息，例如文件名。这些额外信息对我们用处不大
- CBO：输出由Calcite优化器生成的计划。CBO 从 hive 4.0.0 版本开始支持
- AST：输出查询的抽象语法树。AST 在hive 2.1.0 版本删除了，存在bug，转储AST可能会导致OOM错误，将在4.0.0版本修复
- DEPENDENCY：dependency在EXPLAIN语句中使用会产生有关计划中输入的额外信息。它显示了输

入的各种属性
- AUTHORIZATION：显示所有的实体需要被授权执行（如果存在）的查询和授权失败
- LOCKS：这对于了解系统将获得哪些锁以运行指定的查询很有用。LOCKS 从 hive 3.2.0 开始支持
- VECTORIZATION：将详细信息添加到EXPLAIN输出中，以显示为什么未对Map和Reduce进行矢量化。从 Hive 2.3.0 开始支持
- ANALYZE：用实际的行数注释计划。从 Hive 2.2.0 开始支持

在 hive cli 中输入以下命令：

```
explain select sum(id) from emp;
```

得到结果（**请逐行看完，即使看不懂也要每行都看**）：

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 depends on stages: Stage-1

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Map Operator Tree:
          TableScan
            alias: emp
            Statistics: Num rows: 8 Data size: 107 Basic stats: COMPLETE Column stats: NONE
            Select Operator
              expressions: id (type: int)
              outputColumnNames: id
              Statistics: Num rows: 8 Data size: 107 Basic stats: COMPLETE Column stats: NONE
              Group By Operator
                aggregations: sum(id)
                mode: hash
                outputColumnNames: _col0
                Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE
                Reduce Output Operator
                  sort order:
                  Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE
                  value expressions: _col0 (type: bigint)
      Execution mode: vectorized
      Reduce Operator Tree:
        Group By Operator
          aggregations: sum(VALUE._col0)
          mode: mergepartial
          outputColumnNames: _col0
          Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE
          File Output Operator
            compressed: false
            Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats: NONE
            table:
                input format: org.apache.hadoop.mapred.SequenceFileInputFormat
                output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
                serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        ListSink
```

我们将上述结果拆分看，先从最外层开始，包含两个大的部分：

- stage dependencies： 各个stage之间的依赖性
- stage plan： 各个stage的执行计划

先看第一部分 stage dependencies ，包含两个 stage，Stage-1 是根stage，说明这是开始的stage，Stage-0 依赖 Stage-1，Stage-1执行完成后执行Stage-0。

再看第二部分 stage plan，里面有一个 Map Reduce，一个MR的执行计划分为两个部分：

- Map Operator Tree： MAP端的执行计划树
- Reduce Operator Tree： Reduce端的执行计划树

这两个执行计划树里面包含这条sql语句的 operator：

1. map端第一个操作肯定是加载表，所以就是 TableScan 表扫描操作，常见的属性：

（1）alias： 表名称

（2）Statistics： 表统计信息，包含表中数据条数，数据大小等

2. Select Operator： 选取操作**，常见的属性 ：

（1）expressions：需要的字段名称及字段类型

（2）outputColumnNames：输出的列名称

（3）Statistics：表统计信息，包含表中数据条数，数据大小等

3. Group By Operator：分组聚合操作，常见的属性：

（1）aggregations：显示聚合函数信息

（2）mode：聚合模式，值有 hash：随机聚合，就是hash partition；partial：局部聚合；final：最终聚合

（3）keys：分组的字段，如果没有分组，则没有此字段

（4）outputColumnNames：聚合之后输出列名

（5）Statistics： 表统计信息，包含分组聚合之后的数据条数，数据大小等

4. Reduce Output Operator：输出到reduce操作，常见属性：

   sort order：值为空 不排序；值为 + 正序排序，值为 - 倒序排序；值为 +- 排序的列为两列，第一列为正序，第二列为倒序

5. Filter Operator：过滤操作，常见的属性：

   predicate：过滤条件，如sql语句中的where id>=1，则此处显示(id >= 1)

6. Map Join Operator：join 操作，常见的属性：

（1）condition map：join方式 ，如Inner Join 0 to 1 Left Outer Join0 to 2

（2）keys: join 的条件字段

（3）outputColumnNames： join 完成之后输出的字段

（4）Statistics： join 完成之后生成的数据条数，大小等

7. File Output Operator：文件输出操作，常见的属性

（1）compressed：是否压缩

（2）table：表的信息，包含输入输出文件格式化方式，序列化方式等

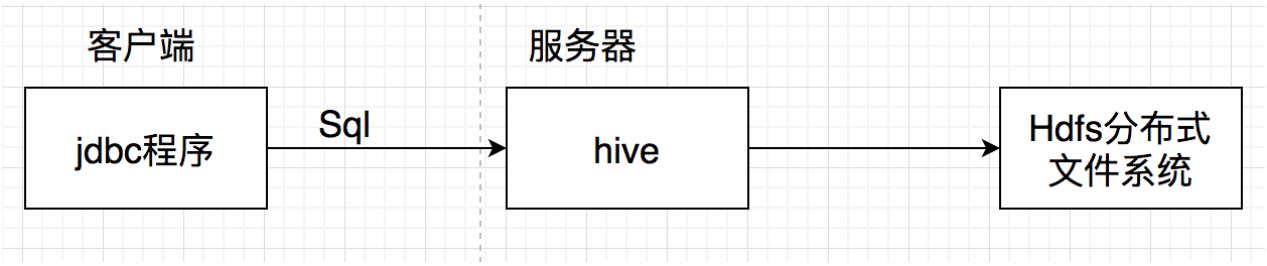
8. Fetch Operator 客户端获取数据操作，常见的属性：

   limit，值为 -1 表示不限制条数，其他值为限制的条数

8 jdbc连接Hive

8.1 概述

jdbc连接hive过程，jdbc是客户端，hive是服务器端，jdbc执行相应的sql语句，sql语句发送给hive，hive基于元数据执行sql返回结果集给jdbc客户端；



> jdbc程序连接hive的时候，默认在hive中的身份是计算机的用户名，通过whoami可以查询
>
> 需要指定用户身份，环境变量中配置：
>
> export HADOOP_USER_NAME=briup

8.2 配置

1. 在hdfs分布式集群中主节点hdfs-site.xml文件中添加如下内容

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
```

2. 在hdfs分布式集群中主节点core-site.xml文件中添加如下内容

```
<property>
  <name>hadoop.proxyuser.hdfs.hosts</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hdfs.groups</name>
  <value>*</value>
</property>
```

3. 重启hadoop集群，并开启hive服务

```
hive --service hiveserver2 &
```

> 服务开启默认端口10000

4. 在pom.xml文件中加入hive对jdbc的支持

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>Had</artifactId>
        <groupId>com.briup</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>Hive</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>3.1.2</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-jdbc</artifactId>
            <version>3.1.2</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>hive</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>7</source>
                    <target>7</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

4. 编写程序，查询stu表中的数据

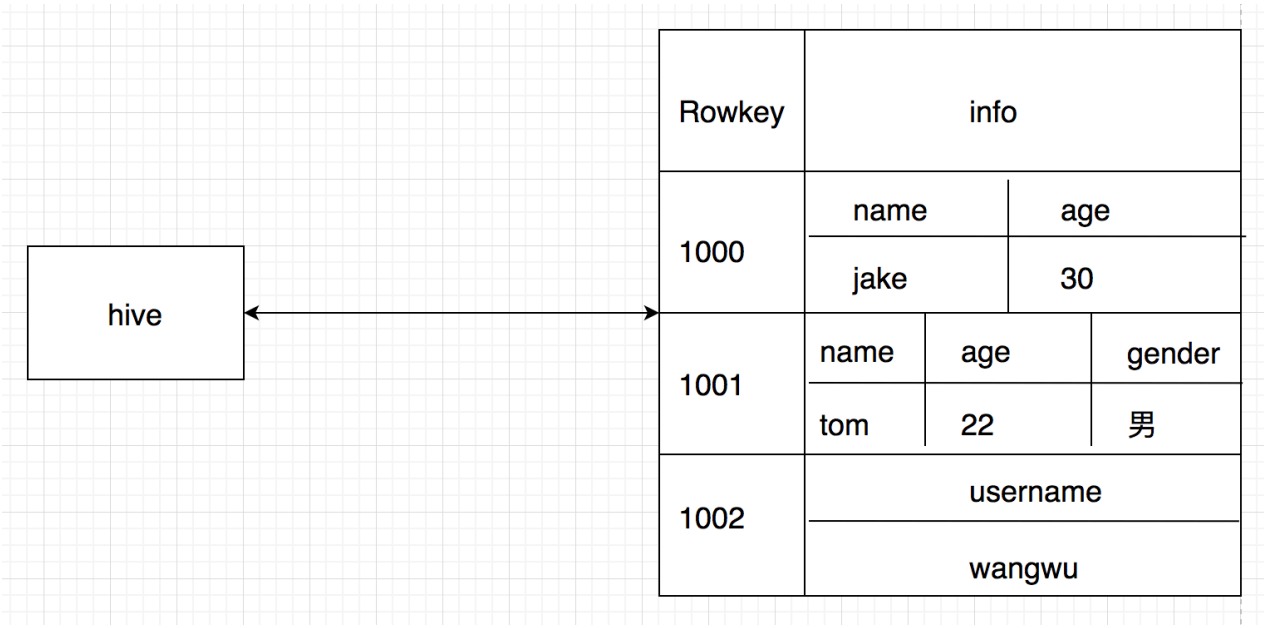
```java
package com.briup.jdbc_test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class FirstJdbc {
    public static void main(String[] args)
            throws Exception {
```

```
        Class.forName("org.apache.hive.jdbc.HiveDriver");
        Connection conn=
        DriverManager.getConnection(
                "jdbc:hive2://192.168.43.8:10000/briup",
                "hdfs","");
        Statement sts=
                conn.createStatement();
        String sql="select * from stu";
        ResultSet rs=sts.executeQuery(sql);
        while(rs.next()){
            int id=rs.getInt(1);
            String name=rs.getString(2);
            System.out.println(id+"--"+name);
        }
    }
}
```

9 hive集成hbase

9.1 概述

Hbase数据库的缺点在于语法格式异类，没有类sql的查询方式，因此在实际的业务当中操作和计算数据非常不方便，但是Hive就不一样了，Hive支持标准的sql语法，于是我们就希望通过Hive这个客户端工具对Hbase中的数据进行操作与查询，进行相应的数据挖掘，这就是所谓Hive与hbase整合的含义



hive和hbase对应关系

hive	hbase
表名	表名
列名	列族
值	Map<列标识，值>

9.2 配置

1. 在hbase命令行中构建表

```
create 'briup:person',{NAME => 'f1',VERSIONS => 1},{NAME => 'f2',VERSIONS => 1},{NAME
=> 'f3',VERSIONS => 1}
```

2. 向hbase表中插入数据

```
put 'briup:person','1001','f1:name','jack'
put 'briup:person','1001','f2:age','18'
put 'briup:person','1002','f1:name','jack'
put 'briup:person','1003','f3:position','ceo'
```

3. 进入hive命令行

```
SET hbase.zookeeper.quorum=master:2181;
SET zookeeper.znode.parent=/hbase;
```

> hbase.zookeeper.quorum：指定HBase使用的zookeeper集群，默认端口是2181，可以不指定，如果指定，格式为zkNode1:2222,zkNode2:2222,zkNode3:2222
>
> zookeeper.znode.parent:指定HBase在zookeeper中使用的根目录

4.在hive命令行创建表

```
CREATE EXTERNAL TABLE person (
rowkey string,
f1 map<STRING,STRING>,
f2 map<STRING,STRING>,
f3 map<STRING,STRING>
) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,f1:,f2:,f3:")
TBLPROPERTIES ("hbase.table.name" = "briup:person");
```

> hbase.columns.mapping：Hive表和HBase表的字段映射关系，分别为：Hive表中第一个字段映射:key(rowkey)，第二个字段映射列族f1，第三个字段映射列族f2,第四个字段映射列族f3
>
> hbase.table.name：HBase中表的名字

5. 测试查询数据

```
select rowkey,f1,f2,f3
from person;
```