目录

[课程大纲（HIVE增强） 3](#_Toc439081657)

[1. Hive基本概念 4](#_Toc439081658)

[1.1 Hive简介 4](#_Toc439081659)

[1.1.1 什么是Hive 4](#_Toc439081660)

[1.1.2 为什么使用Hive 4](#_Toc439081661)

[1.1.3 Hive的特点 4](#_Toc439081662)

[1.2 Hive架构 5](#_Toc439081663)

[1.2.1 架构图 5](#_Toc439081664)

[1.2.2 基本组成 5](#_Toc439081665)

[1.2.3 各组件的基本功能 5](#_Toc439081666)

[1.3 Hive与Hadoop的关系 6](#_Toc439081667)

[1.4 Hive与传统数据库对比 6](#_Toc439081668)

[1.5 Hive的数据存储 6](#_Toc439081669)

[2. Hive基本操作 7](#_Toc439081670)

[2.1 DDL操作 7](#_Toc439081671)

[2.1.1 创建表 7](#_Toc439081672)

[2.1.2 修改表 9](#_Toc439081673)

[2.1.3 显示命令 11](#_Toc439081674)

[2.2 DML操作 11](#_Toc439081675)

[2.2.1 Load 11](#_Toc439081676)

[2.2.2 Insert 13](#_Toc439081677)

[2.2.3 SELECT 15](#_Toc439081678)

[2.3 Hive Join 18](#_Toc439081679)

[2.5 Hive Shell使用进阶 21](#_Toc439081680)

[2.5.1 Hive命令行 21](#_Toc439081681)

[2.5.2. Hive参数配置方式 23](#_Toc439081682)

[4. Hive函数 24](#_Toc439081683)

[4.1 内置运算符 24](#_Toc439081684)

[4.2 内置函数 24](#_Toc439081685)

[4.3 Hive自定义函数和Transform 24](#_Toc439081686)

[4.3.1 自定义函数类别 24](#_Toc439081687)

[4.3.2 UDF开发实例 24](#_Toc439081688)

[4.3.3 Transform实现 25](#_Toc439081689)

[5. Hive执行过程实例分析 26](#_Toc439081690)

[5.1 JOIN 26](#_Toc439081691)

[5.2 GROUP BY 26](#_Toc439081692)

[5.3 DISTINCT 27](#_Toc439081693)

[6. Hive使用注意点（各种小细节） 27](#_Toc439081694)

[6.1 字符集 27](#_Toc439081695)

[6.2 压缩 28](#_Toc439081696)

[6.3 count(distinct) 28](#_Toc439081697)

[6.4 子查询 28](#_Toc439081698)

[6.5 Join中处理null值的语义区别 28](#_Toc439081699)

[6.6 分号字符 29](#_Toc439081700)

[6.7 Insert 29](#_Toc439081701)

[6.7.1新增数据 29](#_Toc439081702)

[6.7.2 插入次序 30](#_Toc439081703)

[6.7.3 初始值 30](#_Toc439081704)

[7. Hive优化技巧 31](#_Toc439081705)

[7.1 HADOOP计算框架特性 31](#_Toc439081706)

[7.2 优化的常用手段概述 31](#_Toc439081707)

[7.3 全排序 32](#_Toc439081708)

[7.3.1 例1 32](#_Toc439081709)

[7.3.2 例2 34](#_Toc439081710)

[7.4 怎样写exist/in子句 36](#_Toc439081711)

[7.5 怎样决定reducer个数 36](#_Toc439081712)

[7.6 合并MapReduce操作 36](#_Toc439081713)

[7.7 Bucket 与 Sampling 37](#_Toc439081714)

[7.8 Partition优化 38](#_Toc439081715)

[7.9 JOIN优化 39](#_Toc439081716)

[7.9.1 JOIN原则 39](#_Toc439081717)

[7.9.2 Map Join 39](#_Toc439081718)

[7.10 数据倾斜 40](#_Toc439081719)

[7.10.1 空值数据倾斜 40](#_Toc439081720)

[7.10.2 不同数据类型关联产生数据倾斜 41](#_Toc439081721)

[7.10.3 大表Join的数据偏斜 41](#_Toc439081722)

[7.11 合并小文件 42](#_Toc439081723)

[7.12 Group By 优化 43](#_Toc439081724)

[7.12.1 Map端部分聚合： 43](#_Toc439081725)

[7.12.2 有数据倾斜的时候进行负载均衡 43](#_Toc439081726)

[8. Hive实战 44](#_Toc439081727)

[Hive 实战案例1——数据ETL 44](#_Toc439081728)

[需求： 44](#_Toc439081729)

[数据示例： 44](#_Toc439081730)

[实现步骤： 45](#_Toc439081731)

[Hive 实战案例2——访问时长统计 47](#_Toc439081732)

[需求： 47](#_Toc439081733)

[实现步骤： 47](#_Toc439081734)

[Hive实战案例3——级联求和 48](#_Toc439081735)

[需求： 48](#_Toc439081736)

[实现步骤 48](#_Toc439081737)

# 课程大纲（HIVE增强）

|  |  |
| --- | --- |
| Hive增强 | HIVE基本概念 |
| HIVE架构及运行机制 |
| HQL-DDL基本语法 |
| HQL-DML基本语法 |
| HIVE的join |
| HIVE UDF函数 |
| HIVE shell基本操作 |
| HIVE 参数配置 |
| HIVE 自定义函数和Transform |
| HIVE 执行HQL的实例分析 |
| HIVE最佳实践注意点 |
| HIVE优化策略 |
| HIVE实战案例1 |
| HIVE实战案例2 |
| HIVE实战案例3 |

学习目标：

1、熟练掌握hive的使用

2、熟练掌握hql的编写

3、理解hive的工作原理

4、具备hive应用实战能力

# Hive基本概念

## Hive简介

### 什么是Hive

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。

### 为什么使用Hive

* 直接使用hadoop所面临的问题

人员学习成本太高

项目周期要求太短

MapReduce实现复杂查询逻辑开发难度太大

* 为什么要使用Hive

操作接口采用类SQL语法，提供快速开发的能力。

避免了去写MapReduce，减少开发人员的学习成本。

扩展功能很方便。

### Hive的特点

* 可扩展

Hive可以自由的扩展集群的规模，一般情况下不需要重启服务。

* 延展性

Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

* 容错

良好的容错性，节点出现问题SQL仍可完成执行。

## Hive架构

### 架构图



Jobtracker是hadoop1.x中的组件，它的功能相当于： **Resourcemanager+AppMaster**

TaskTracker 相当于： **Nodemanager + yarnchild**

### 基本组成

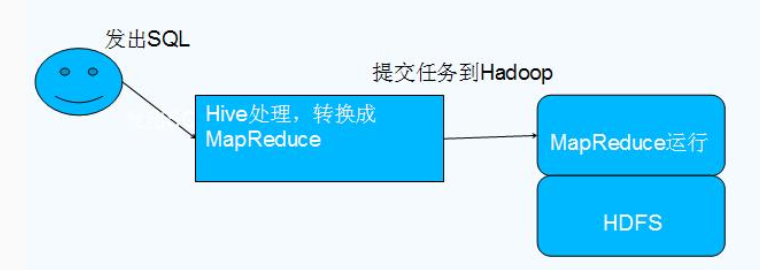
* 用户接口：包括 CLI、JDBC/ODBC、WebGUI。
* 元数据存储：通常是存储在关系数据库如 mysql , derby中。
* 解释器、编译器、优化器、执行器。

### 各组件的基本功能

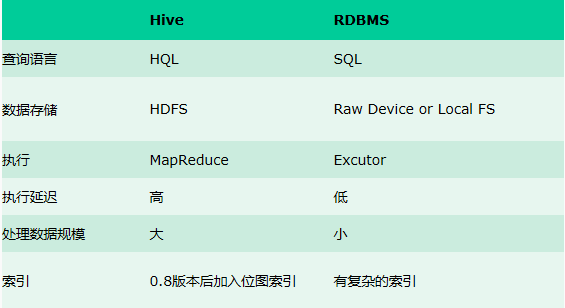
* 用户接口主要由三个：**CLI、JDBC/ODBC和WebGUI**。其中，CLI为shell命令行；JDBC/ODBC是Hive的JAVA实现，与传统数据库JDBC类似；WebGUI是通过浏览器访问Hive。
* 元数据存储：Hive 将元数据存储在数据库中。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
* 解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后有 MapReduce 调用执行。

## Hive与Hadoop的关系

Hive利用HDFS存储数据，利用MapReduce查询数据



## Hive与传统数据库对比



*总结：hive具有sql数据库的外表，但应用场景完全不同，hive只适合用来做批量数据统计分析*

## Hive的数据存储

1、Hive中所有的数据都存储在 HDFS 中，没有专门的数据存储格式（可支持Text，SequenceFile，ParquetFile，RCFILE等）

2、只需要在创建表的时候告诉 Hive 数据中的**列分隔符和行分隔符**，Hive 就可以解析数据。

3、Hive 中包含以下数据模型：DB、Table，External Table，Partition，Bucket。

* db：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db目录下一个文件夹
* external table：外部表, 与table类似，不过其数据存放位置可以在任意指定路径

普通表: 删除表后, hdfs上的文件都删了

External外部表删除后, hdfs上的文件没有删除, 只是把文件删除了

* partition：在hdfs中表现为table目录下的子目录
* bucket：桶, 在hdfs中表现为同一个表目录下根据hash散列之后的多个文件, 会根据不同的文件把数据放到不同的文件中

## 1.6 HIVE的安装部署

### 1.6.1 安装

单机版：

元数据库mysql版：

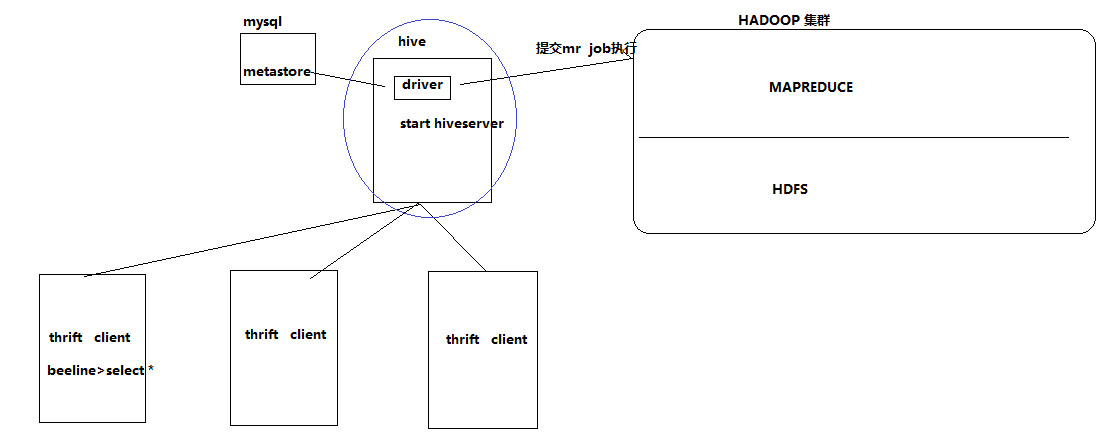
### 1.6.2 使用方式

#### Hive交互shell

bin/hive

#### Hive thrift服务

将hive启动为服务，该服务是使用thrift协议，只要客户端满足thrift协议就能连上hive服务



启动方式，（假如是在hadoop01上）：

启动为前台：**bin/hiveserver2**

启动为后台：nohup bin/hiveserver2 1>/var/log/hiveserver.log 2>/var/log/hiveserver.err **&**

启动成功后，可以在别的节点(该节点要满足thrift协议)上用beeline去连接

* 方式（1）

**hive/bin/beeline** 回车，进入beeline的命令界面

输入命令连接hiveserver2

beeline> !connect jdbc:hive2://mini1:10000

（hadoop01是hiveserver2所启动的那台主机名，端口默认是10000）

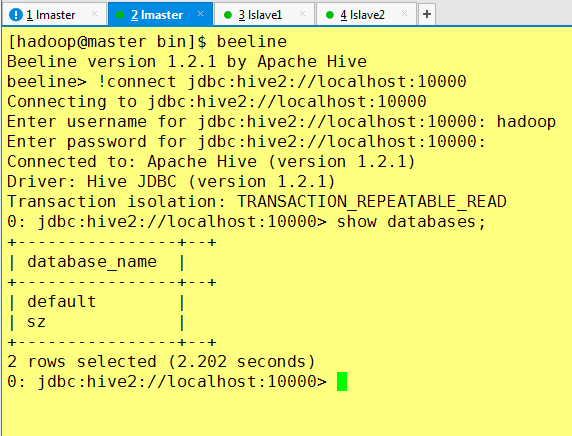
* 方式（2）

或者启动就连接：

**bin/beeline -u jdbc:hive2://mini1:10000 -n hadoop**

默认用户名是启动hive服务的用户名，密码无，这些可以再hive-site.xml里面配置

接下来就可以做正常sql查询了，而且界面更好



#### Hive命令

[hadoop@hdp-node-02 ~]$ hive -e ‘sql’

# Hive基本操作

## DDL操作

### 创建表

#### 建表语法

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table\_name

[(col\_name data\_type [COMMENT col\_comment], ...)]

[COMMENT table\_comment]

[PARTITIONED BY (col\_name data\_type [COMMENT col\_comment], ...)]

[CLUSTERED BY (col\_name, col\_name, ...)

[SORTED BY (col\_name [ASC|DESC], ...)] INTO num\_buckets BUCKETS]

[ROW FORMAT row\_format]

[STORED AS file\_format]

[LOCATION hdfs\_path]

说明：

1. 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。
2. EXTERNAL关键字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。
3. **LIKE 允许用户复制现有的表结构，但是不复制数据。**
4. **ROW FORMAT**

DELIMITED [FIELDS TERMINATED BY char] [COLLECTION ITEMS TERMINATED BY char]

[MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]

| SERDE serde\_name [WITH SERDEPROPERTIES (property\_name=property\_value, property\_name=property\_value, ...)]

用户在建表的时候可以自定义 SerDe 或者使用自带的 SerDe。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用自带的 SerDe。在建表的时候，用户还需要为表指定列，用户在指定表的列的同时也会指定自定义的 SerDe，Hive通过 **SerDe** 确定表的具体的列的数据。

1. STORED AS

SEQUENCEFILE|TEXTFILE|RCFILE

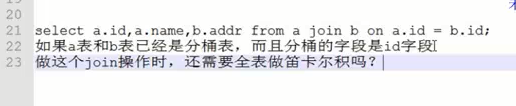
如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCEFILE。

6、CLUSTERED BY

对于每一个表（table）或者分区， 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。**Hive采用对列值哈希，然后除以桶的个数求余**的方式决定该条记录存放在哪个桶当中。

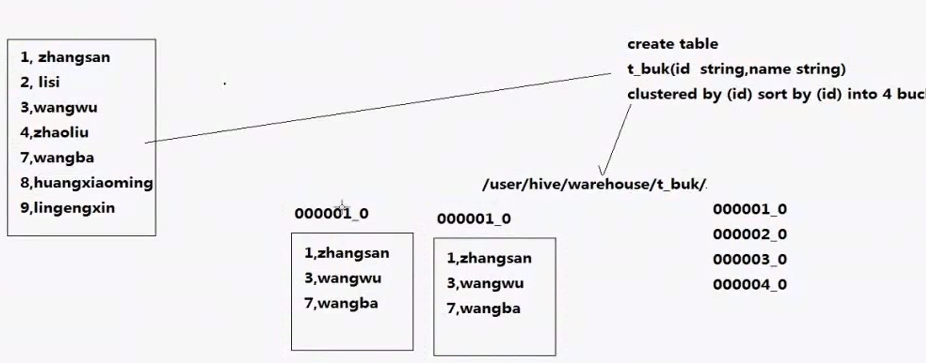
把表（或者分区）组织成桶（Bucket）有两个理由：

* 1. 获得更高的查询处理效率。桶为表加上了额外的结构，Hive 在处理有些查询时能利用这个结构。**具体而言，连接两个在（包含连接列的）相同列上划分了桶的表，可以使用 Map 端连接 （Map-side join）高效的实现**。比如**JOIN**操作。对于JOIN操作两个表有一个相同的列，**如果对这两个表都进行了桶操作**。那么将保存相同列值的桶进行JOIN操作就可以，可以大大较少JOIN的数据量。



（2）使取样（sampling）更高效。在处理大规模数据集时，在开发和修改查询的阶段，如果能在数据集的一小部分数据上试运行查询，会带来很多方便。

**创建分桶表后，把数据load到表中后表是不会分区的（创建表只是说这个表示一个分桶排序的表，你要把我想要的数据放里面），只有你把别的表中的数据select到分桶表中后，表中数据才是分区的：**



insert into table t\_buck select \*from t\_p; 但是这样还是分桶的，但是没有分区排序，是没用的，当你那这个表中的数据和别的表join时，会误认为你这个表是分好桶排好序的，这时就会报错。

**设置开启分桶, 并设置reduce数量是分桶的数量个数，而且select的数据要是分区且排序后的才能插入到分桶表中去：**

set hive.enforce.bucketing = true;

set mapreduce.job.reduces=4;

**所以一个完整步骤是：**

#创建分桶表

create table stu\_buck(sno int,sname string,sex string,sage int,sdept string)

**clustered** by(sno) /\*clustered by(sno)= distributeby（sno）+sort by（sno）这里不能写distribute by\*/

**sorted** by(sno DESC)

into 4 buckets

row format delimited

fields terminated by ',';

#设置变量,设置分桶为true, 设置reduce数量是分桶的数量个数

set hive.enforce.bucketing = true;

set mapreduce.job.reduces=4;

#开始往创建的分通表插入数据(插入数据需要是已分桶, 且排序的)

#可以使用distribute by(sno) sort by(sno asc) 或是排序和分桶的字段相同的时候使用**Cluster by(字段)** *cluster by（id） = distribute by（id） + sort by（id）*

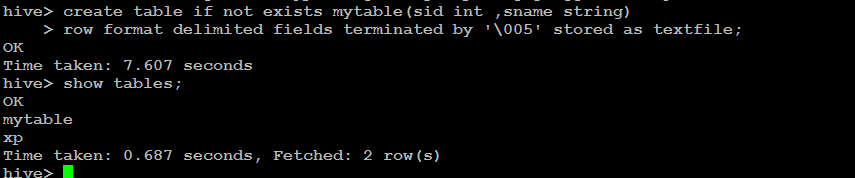
#注意使用cluster by 就等同于分桶+排序(sort)

insert into table stu\_buck

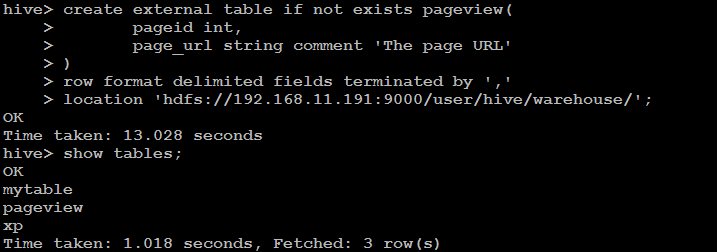
select sno,sname,sex,sage,sdept from student distribute by(sno) sort by(sno asc);

#### 具体实例

1. 创建内部表mytable。



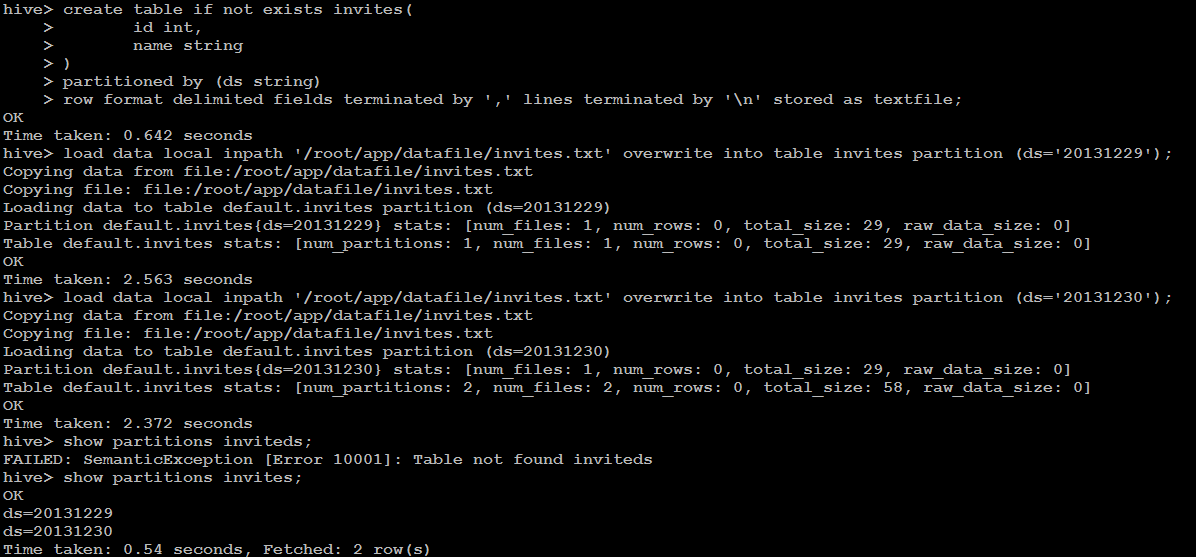
1. 创建外部表pageview。

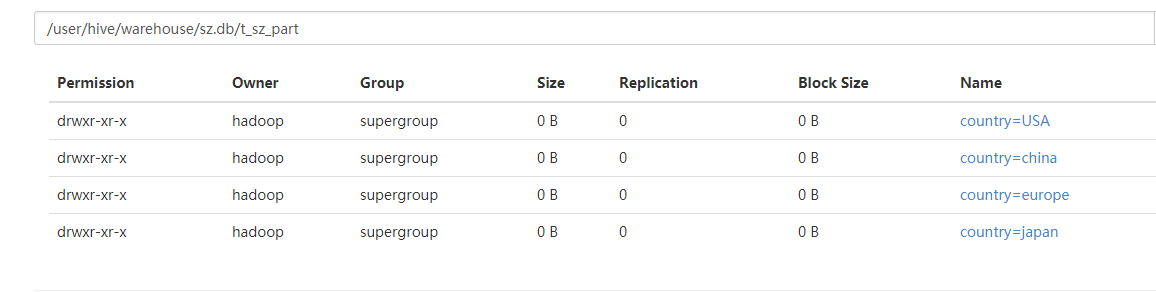


1. 创建分区表invites。

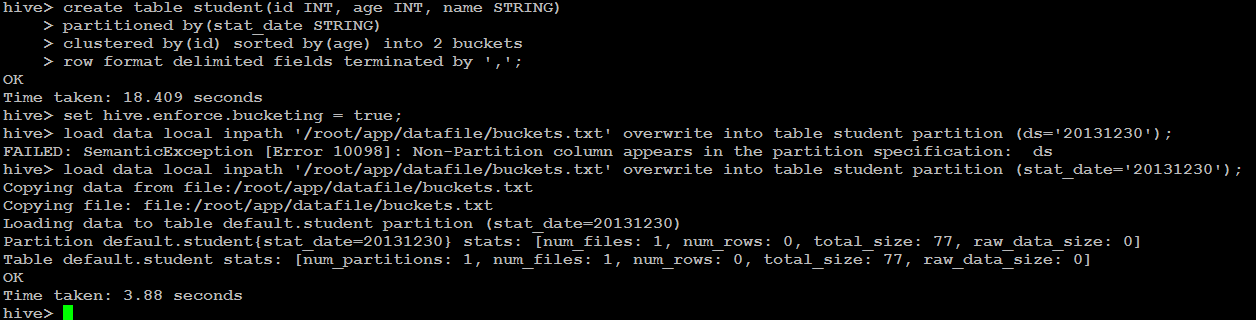
|  |
| --- |
| create table student\_p(Sno int,Sname string,Sex string,Sage int,Sdept string) partitioned by(part string) row format delimited fields terminated by ','stored as textfile; |

**注意分区的列名不能和表中的字段名相同**，其实分区指定的字段也是表的字段。





1. 创建带桶的表student。



### 修改表

#### 增加/删除分区

* 语法结构

ALTER TABLE table\_name ADD [IF NOT EXISTS] partition\_spec [ LOCATION 'location1' ] partition\_spec [ LOCATION 'location2' ] ...

partition\_spec:

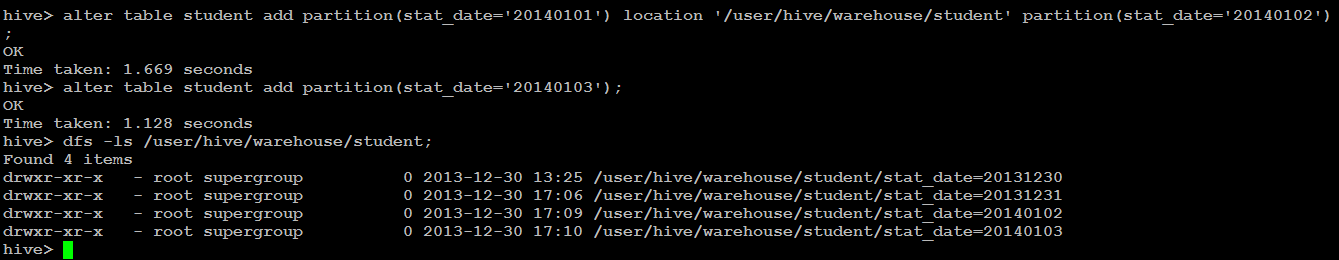
: PARTITION (partition\_col = partition\_col\_value, partition\_col = partiton\_col\_value, ...)

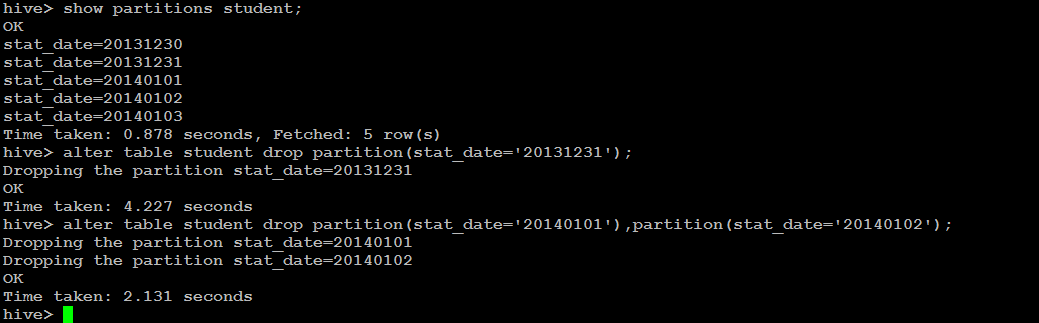
ALTER TABLE table\_name DROP partition\_spec, partition\_spec,...

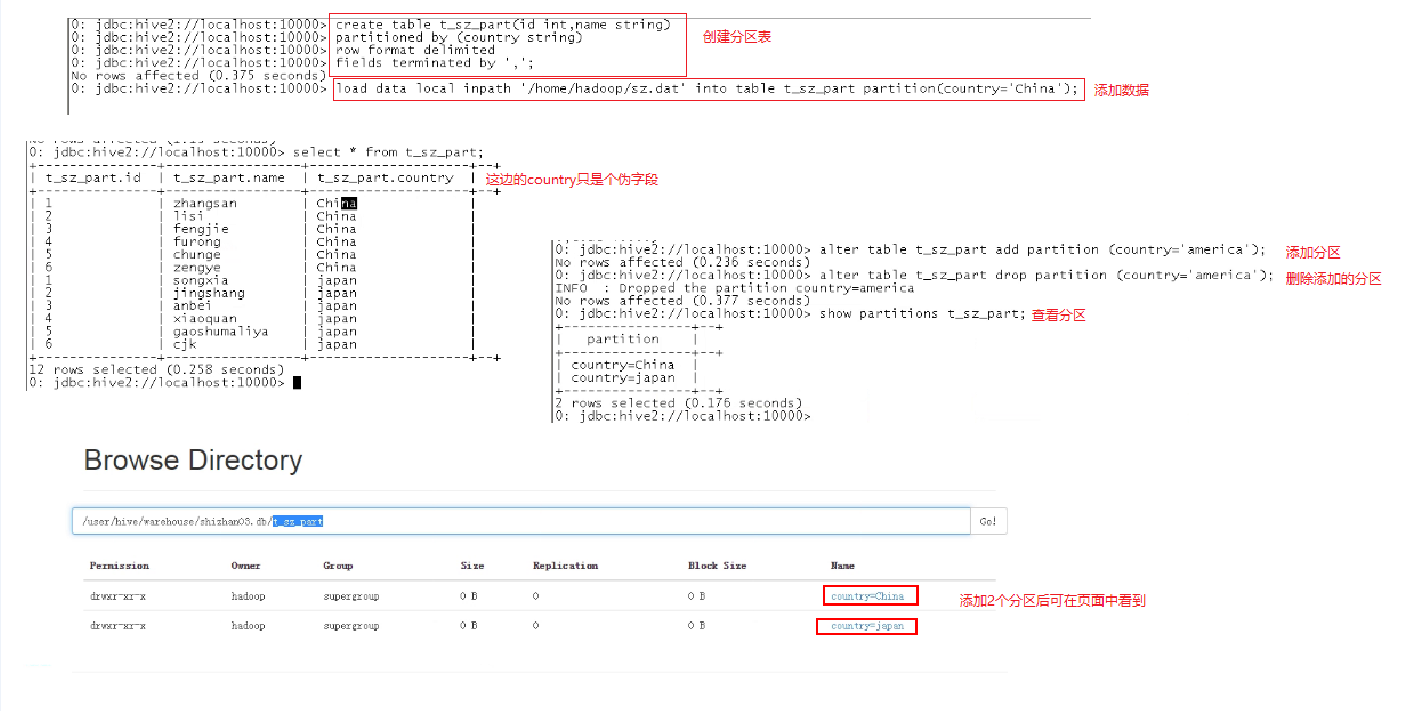
* 具体实例

|  |
| --- |
| alter table student\_p add partition(part='a') partition(part='b'); |

**建表时指定的分区字段是part**





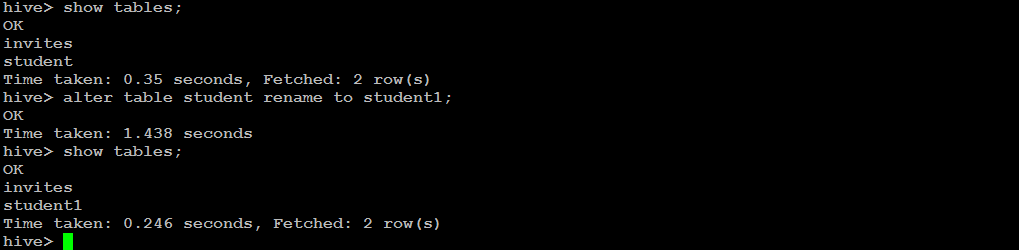


#### 重命名表

* 语法结构

ALTER TABLE table\_name RENAME TO new\_table\_name

* 具体实例



#### 增加/更新列

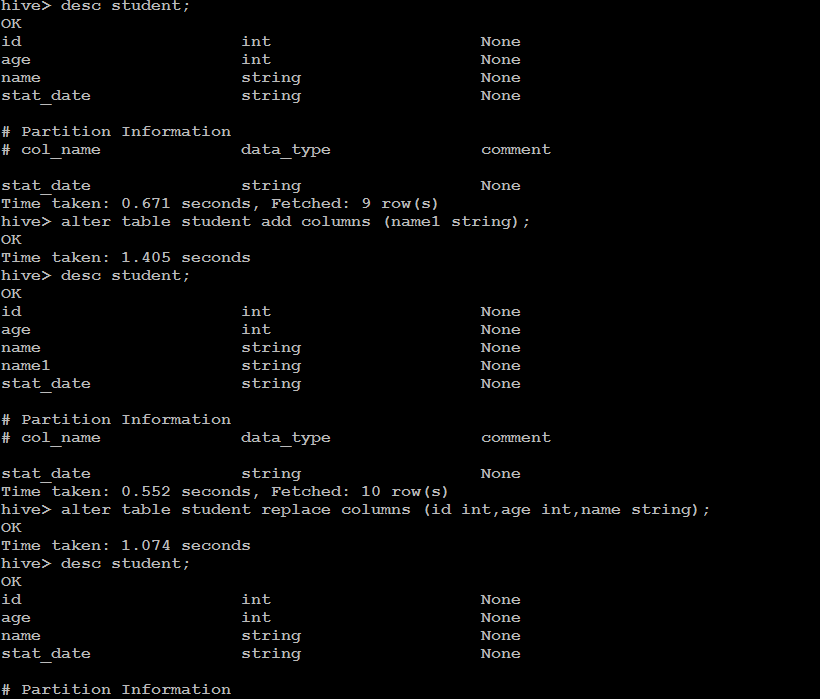
* 语法结构

ALTER TABLE table\_name ADD|REPLACE COLUMNS (col\_name data\_type [COMMENT col\_comment], ...)

*注：ADD是代表新增一字段，字段位置在所有列后面(partition列前)，REPLACE则是表示替换表中所有字段。*

ALTER TABLE table\_name CHANGE [COLUMN] col\_old\_name col\_new\_name column\_type [COMMENT col\_comment] [FIRST|AFTER column\_name]

* 具体实例



### 显示命令

show tables

show databases

show partitions table\_name;查看表的分区

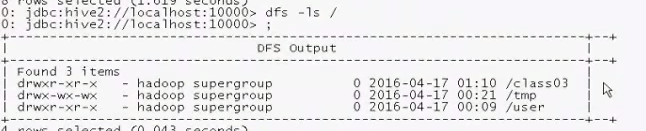
show functions

desc extended t\_name;

desc formatted table\_name;

show create table table\_name;

在hive命令行可以使用hdfs命令，不用加hadoop了：



truncate table person; 清空表中的数据

## DML操作

### Load

* 语法结构

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO

TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

说明：

1. Load 操作只是单纯的复制/移动操作，将数据文件移动到 Hive 表对应的位置。
2. filepath：

相对路径，例如：project/data1

绝对路径，例如：/user/hive/project/data1

包含模式的完整 URI，列如：

hdfs://namenode:9000/user/hive/project/data1

file:///home

1. LOCAL关键字

如果指定了 LOCAL， load 命令会去查找本地文件系统中的 filepath。

如果没有指定 LOCAL 关键字，则根据inpath中的uri查找文件

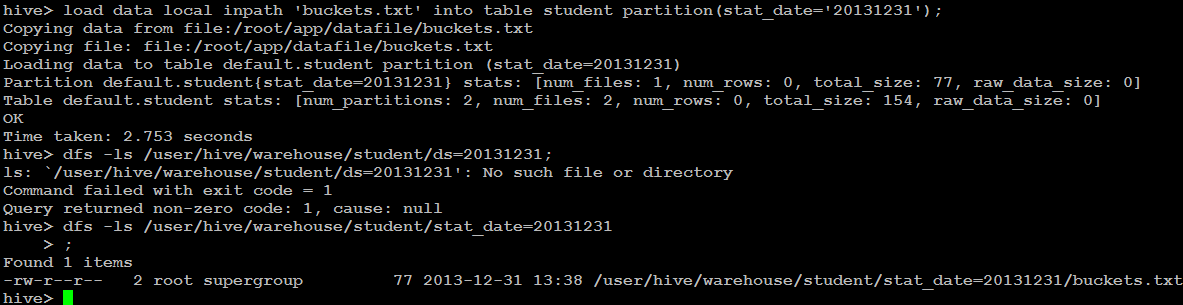
1. OVERWRITE 关键字

如果使用了 OVERWRITE 关键字，则目标表（或者分区）中的内容会被删除，然后再将 filepath 指向的文件/目录中的内容添加到表/分区中。

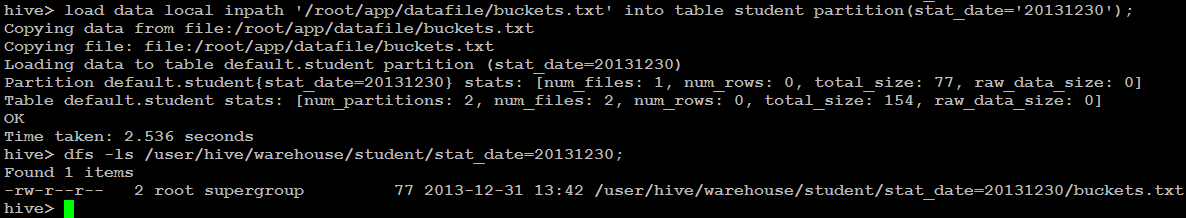
如果目标表（分区）已经有一个文件，并且文件名和 filepath 中的文件名冲突，那么现有的文件会被新文件所替代。

* 具体实例

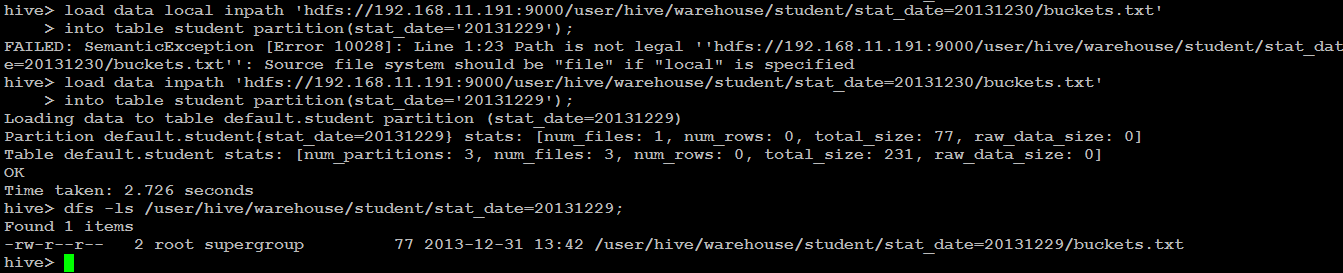
1. 加载相对路径数据。



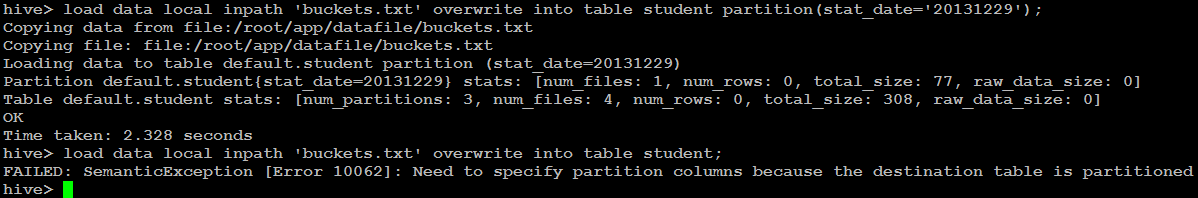
1. 加载绝对路径数据。



1. 加载包含模式数据。



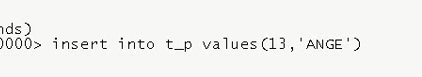
1. OVERWRITE关键字使用。



### Insert

可以这样插入，也会去执行mr，这样会在t\_p的目录下**新加一个文件00000\_0**，不会将数据放到之前的文件下，因为hdfs不支持修改文件。但是这样做效率不高，一般是批量从其它表查询再插入进来。

**Insert into t\_p values(13,’ange’);**



* 将查询结果插入Hive表
* 语法结构

INSERT OVERWRITE/into TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1 FROM from\_statement

Multiple inserts:

FROM from\_statement

INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1

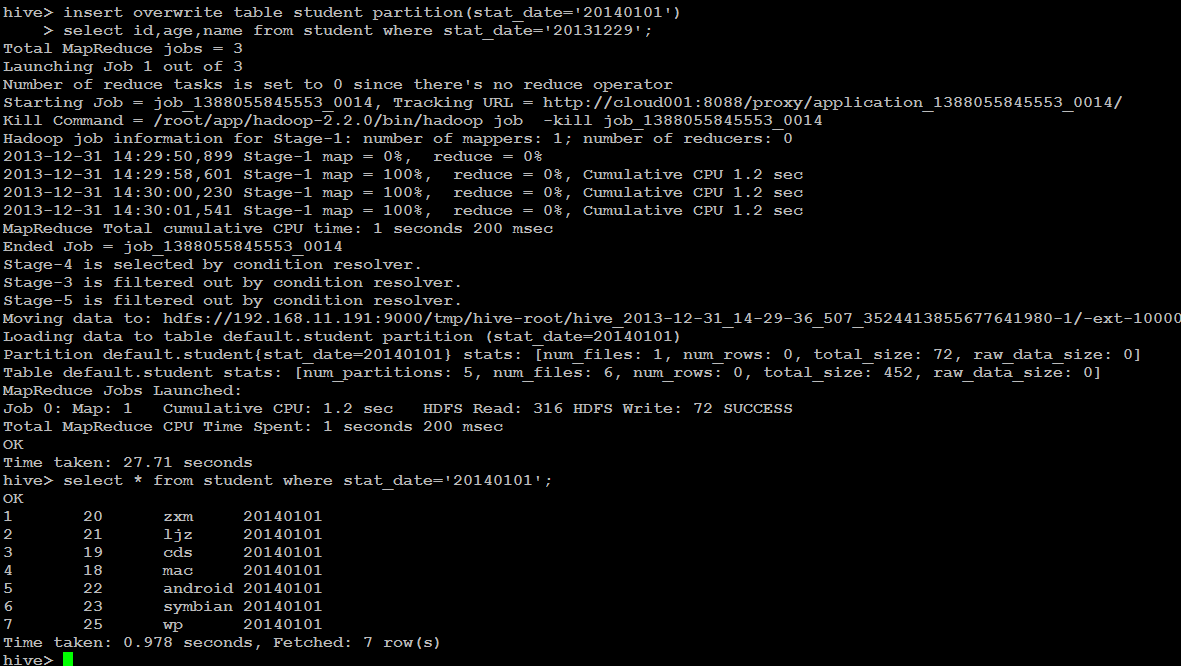
[INSERT OVERWRITE TABLE tablename2 [PARTITION ...] select\_statement2] ...

Dynamic partition inserts:

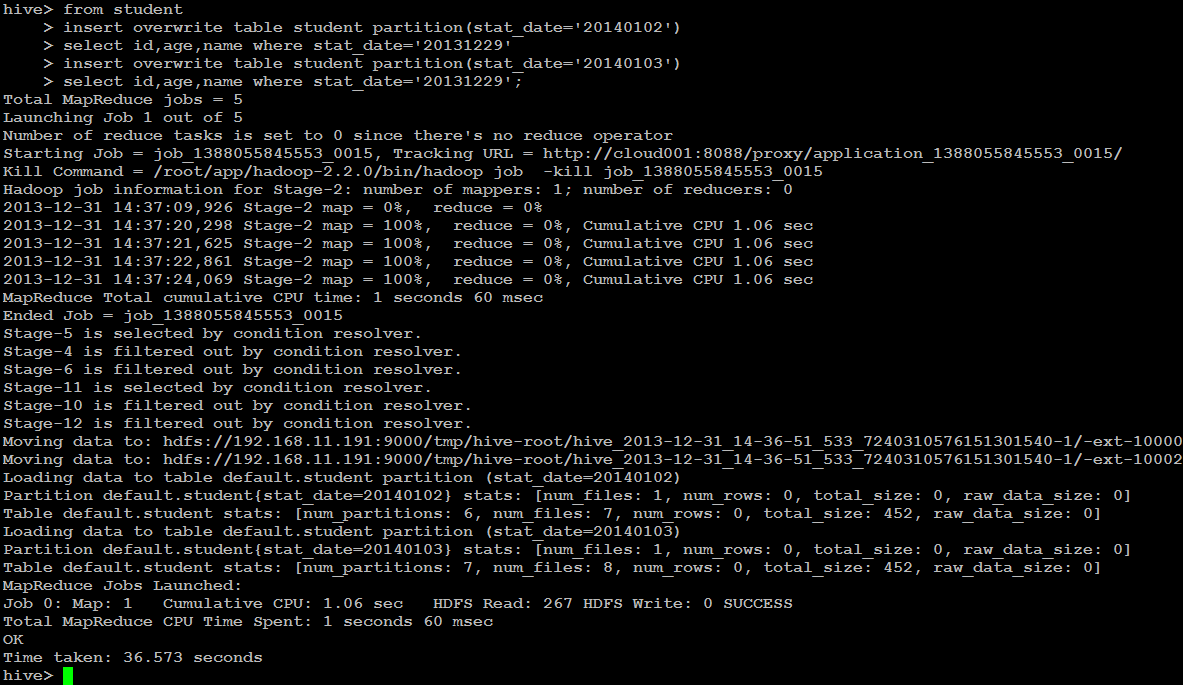
INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select\_statement FROM from\_statement

* 具体实例

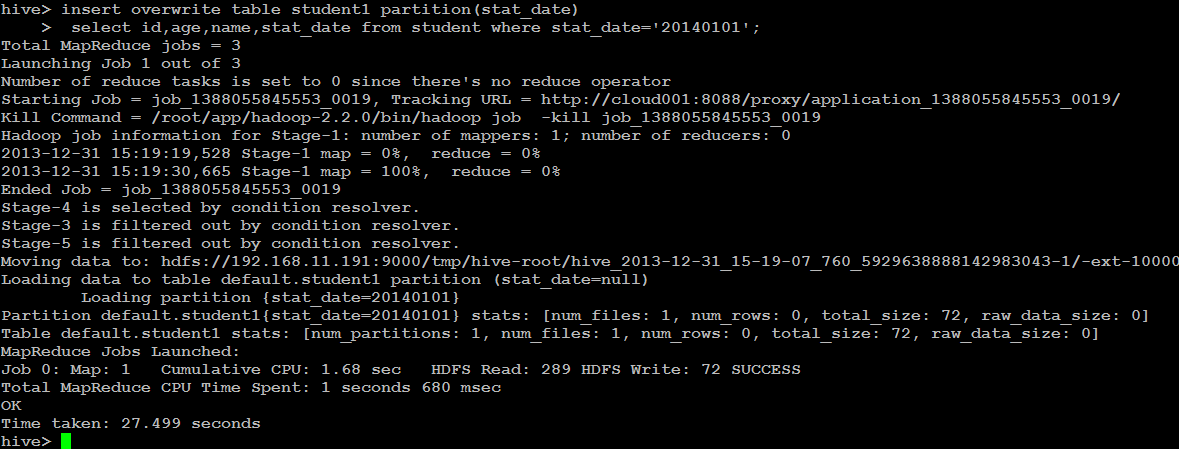
1、基本模式插入。



2、多插入模式。



3、自动分区模式。



* 导出表数据
* 语法结构

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 SELECT ... FROM ...

multiple inserts:

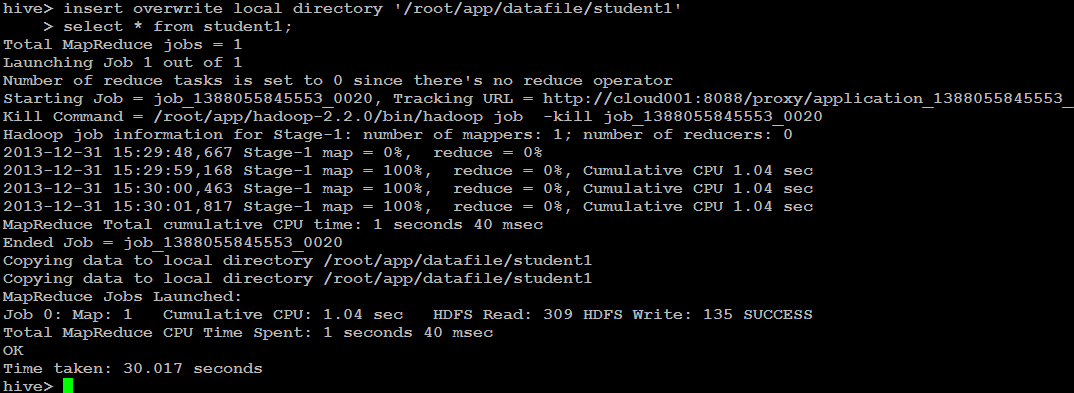
FROM from\_statement

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select\_statement1

[INSERT OVERWRITE [LOCAL] DIRECTORY directory2 select\_statement2] ...

* 具体实例

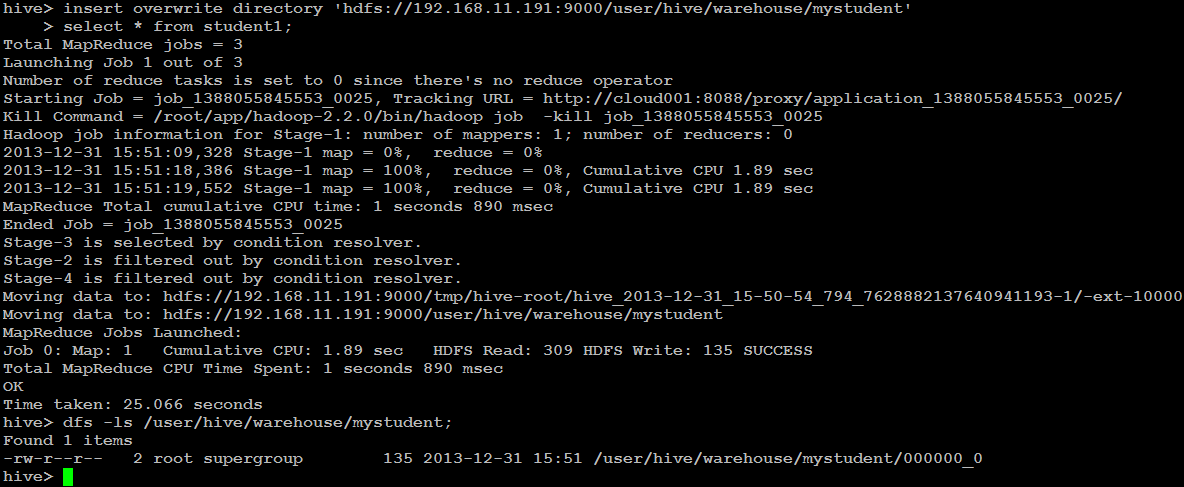
1、导出文件到本地。



说明：

*数据写入到文件系统时进行****文本序列化****，且每列用^A来区分，\n为换行符。用more命令查看时不容易看出分割符，可以使用: sed -e 's/\x01/|/g' filename来查看。*

2、导出数据到HDFS。



### SELECT

* 基本的Select操作
* 语法结构

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]

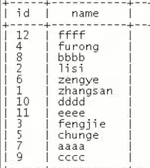
]

[LIMIT number]

***注****：1、****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。*

*2、****sort by（排序）****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（****hive中设置set mapred.reduce.tasks=4****），则sort by****只保证每个reducer内有序，不保证全局有序。默认升序***





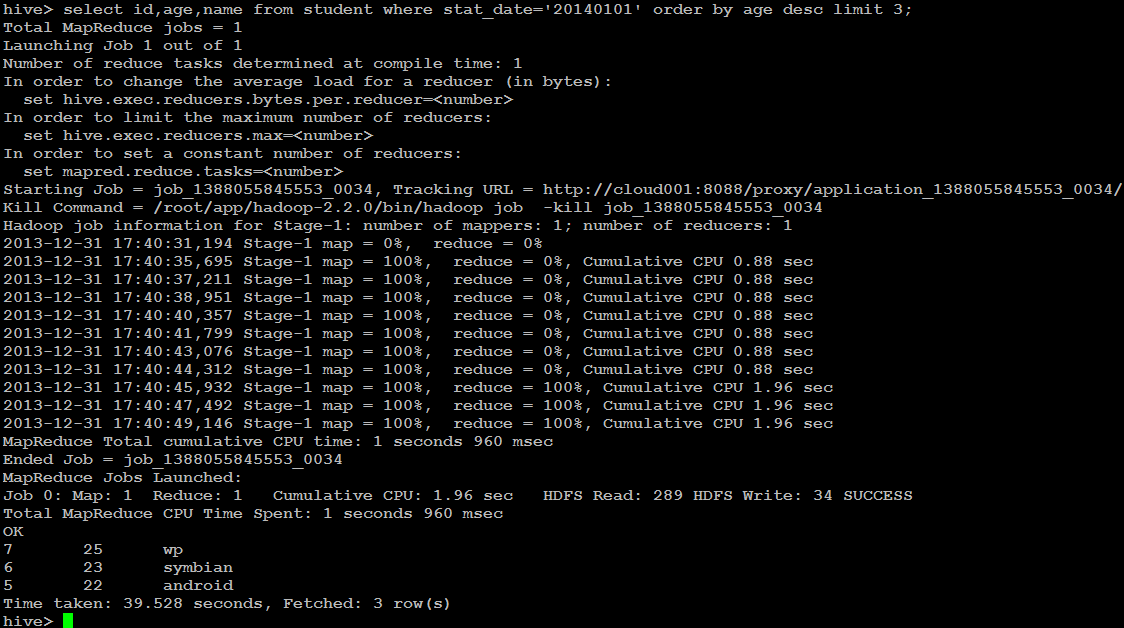
*这个reduce为4 区内是有序的： 12 4 8 是一个区 2 6是一个区 1 10 11是一个 3 5 7 9是一个*

*3、****distribute by（分区hash分区）****根据distribute by****指定的内容将数据分到同一个reducer。***

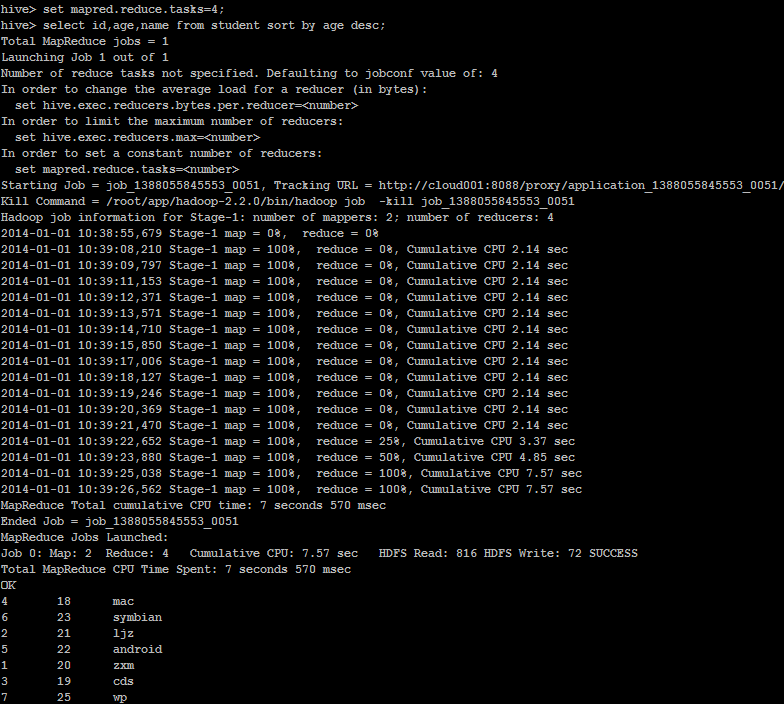
*4、****Cluster by*** *除了具有Distribute by的功能外，还会对该字段进行排序。因此，如果分桶和sort字段是同一个时，常常认为cluster by（id） = distribute by（id） + sort by（id****默认升序****） 这个没有用distribute by（id） + sort by（id）灵活，如：distribute by（id） + sort by（id DESC）*

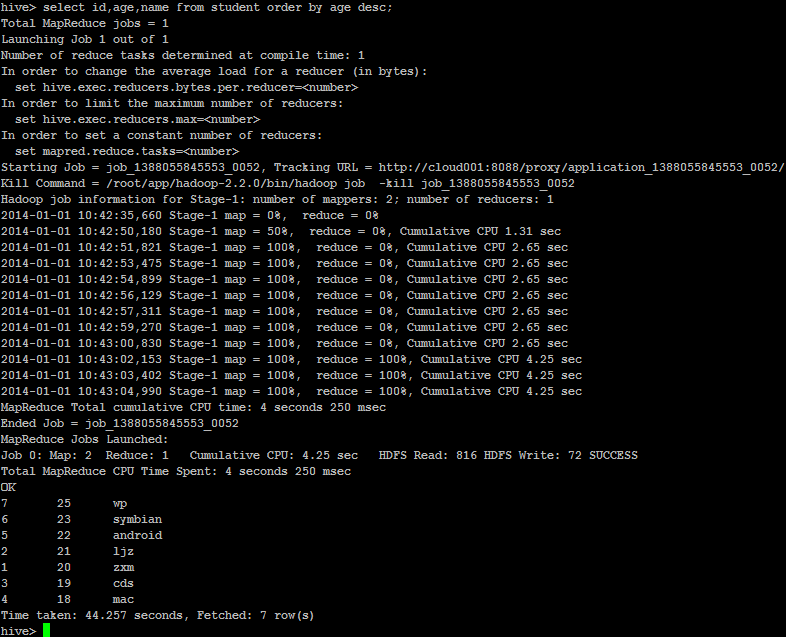
* 具体实例

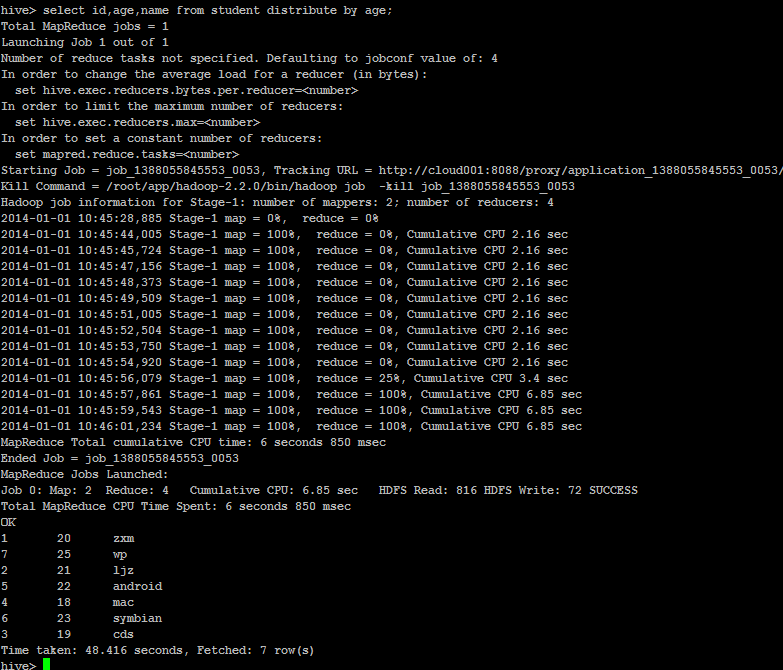
1、获取年龄大的3个学生。



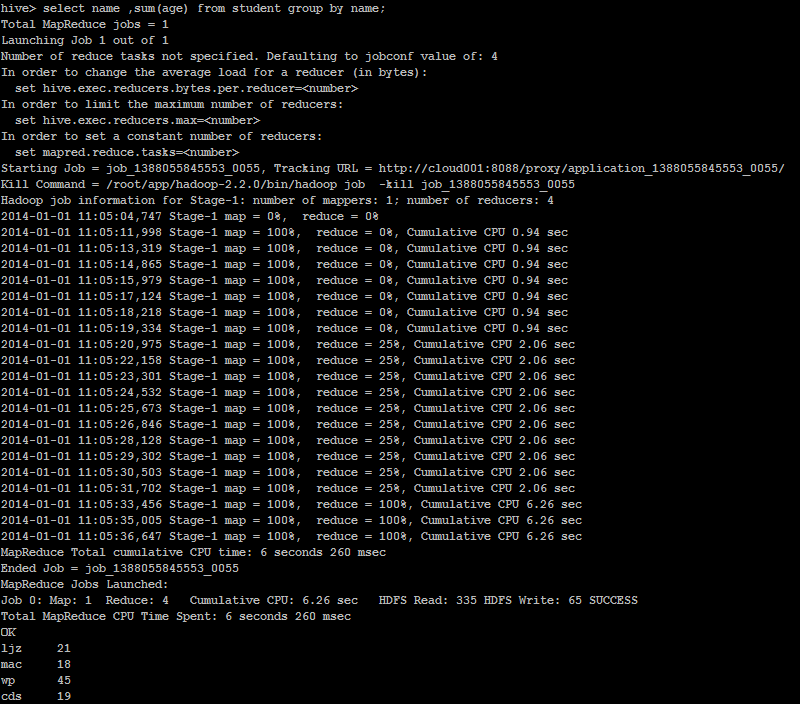
2、查询学生信息按年龄，降序排序。







3、按学生名称汇总学生年龄。



## 2.3 Hive Join

* 语法结构

join\_table:

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference join\_condition

| table\_reference LEFT SEMI JOIN table\_reference join\_condition

Hive 支持等值连接（equality joins）、外连接（outer joins）和（left/right joins）。Hive **不支持非等值的连接**，因为非等值连接非常难转化到 map/reduce 任务。

另外，Hive 支持多于 2 个表的连接。

写 join 查询时，需要注意几个关键点：

**1. 只支持等值join**

例如：

SELECT a.\* FROM a JOIN b ON (a.id = b.id)

SELECT a.\* FROM a JOIN b

ON (a.id = b.id AND a.department = b.department)

是正确的，然而:

**SELECT a.\* FROM a JOIN b ON (a.id>b.id)**

**是错误的。**

**2. 可以 join 多于 2 个表。**

例如

SELECT a.val, b.val, c.val FROM a JOIN b

ON (a.key = b.key1) JOIN c ON (c.key = b.key2)

**如果join中多个表的 join key 是同一个，则 join 会被转化为单个 map/reduce 任务**，例如：

SELECT a.val, b.val, c.val FROM a JOIN b

ON (a.key = b.key1) JOIN c

ON (c.key = b.key1)

被转化为单个 map/reduce 任务，因为 join 中只使用了 b.key1 作为 join key。

SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1)

JOIN c ON (c.key = b.key2)

而这一 join 被转化为 2 个 map/reduce 任务。因为 b.key1 用于第一次 join 条件，而 b.key2 用于第二次 join。

**3．join 时，每次 map/reduce 任务的逻辑：**

reducer 会缓存 join 序列中除了最后一个表的所有表的记录，再通过最后一个表将结果序列化到文件系统。这一实现有助于在 reduce 端减少内存的使用量。实践中，**应该把最大的那个表写在最后（否则会因为缓存浪费大量内存）**。例如：

SELECT a.val, b.val, c.val FROM a

JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)

所有表都使用同一个 join key（使用 1 次 map/reduce 任务计算）。Reduce 端会缓存 a 表和 b 表的记录，然后每次取得一个 c 表的记录就计算一次 join 结果，类似的还有：

SELECT a.val, b.val, c.val FROM a

JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)

这里用了 2 次 map/reduce 任务。第一次缓存 a 表，用 b 表序列化；第二次缓存第一次 map/reduce 任务的结果，然后用 c 表序列化。

**4．LEFT，RIGHT 和 FULL OUTER 关键字用于处理 join 中空记录的情况**

例如：

SELECT a.val, b.val FROM

a LEFT OUTER JOIN b ON (a.key=b.key)

对应所有 a 表中的记录都有一条记录输出。输出的结果应该是 a.val, b.val，当 a.key=b.key 时，而当 b.key 中找不到等值的 a.key 记录时也会输出:

a.val, NULL

所以 a 表中的所有记录都被保留了；

“a RIGHT OUTER JOIN b”会保留所有 b 表的记录。

**Join 发生在 WHERE 子句之前**。如果你想限制 join 的输出，应该在 WHERE 子句中写过滤条件——或是在 join 子句中写。这里面一个容易混淆的问题是表分区的情况：

SELECT a.val, b.val FROM a

LEFT OUTER JOIN b ON (a.key=b.key)

WHERE a.ds='2009-07-07' AND b.ds='2009-07-07'

会 join a 表到 b 表（OUTER JOIN），列出 a.val 和 b.val 的记录。WHERE 从句中可以使用其他列作为过滤条件。但是，如前所述，如果 b 表中找不到对应 a 表的记录，b 表的所有列都会列出 NULL，**包括 ds 列**。也就是说，join 会过滤 b 表中不能找到匹配 a 表 join key 的所有记录。这样的话，LEFT OUTER 就使得查询结果与 WHERE 子句无关了。解决的办法是在 OUTER JOIN 时使用以下语法：

SELECT a.val, b.val FROM a LEFT OUTER JOIN b

ON (a.key=b.key AND

b.ds='2009-07-07' AND

a.ds='2009-07-07')

这一查询的结果是预先在 join 阶段过滤过的，所以不会存在上述问题。这一逻辑也可以应用于 RIGHT 和 FULL 类型的 join 中。

**Join 是不能交换位置的。**无论是 LEFT 还是 RIGHT join，都是左连接的。

SELECT a.val1, a.val2, b.val, c.val

FROM a

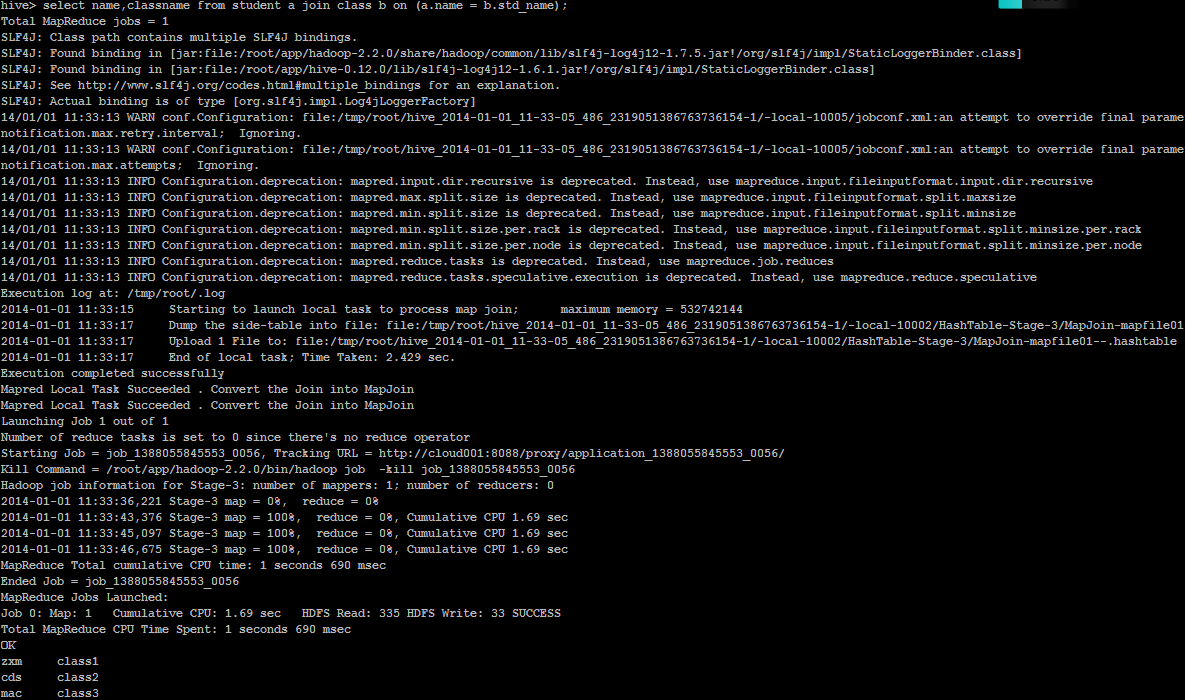
JOIN b ON (a.key = b.key)

LEFT OUTER JOIN c ON (a.key = c.key)

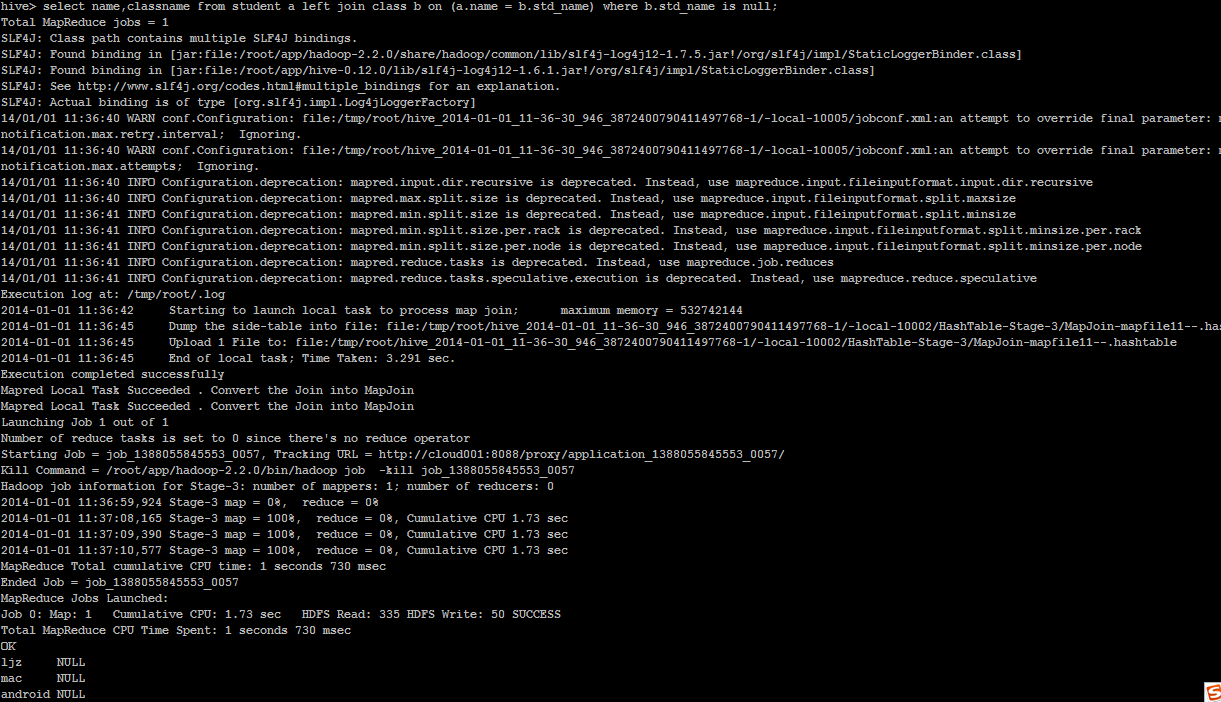
先 join a 表到 b 表，丢弃掉所有 join key 中不匹配的记录，然后用这一中间结果和 c 表做 join。这一表述有一个不太明显的问题，就是当一个 key 在 a 表和 c 表都存在，但是 b 表中不存在的时候：整个记录在第一次 join，即 a JOIN b 的时候都被丢掉了（包括a.val1，a.val2和a.key），然后我们再和 c 表 join 的时候，如果 c.key 与 a.key 或 b.key 相等，就会得到这样的结果：NULL, NULL, NULL, c.val

* 具体实例

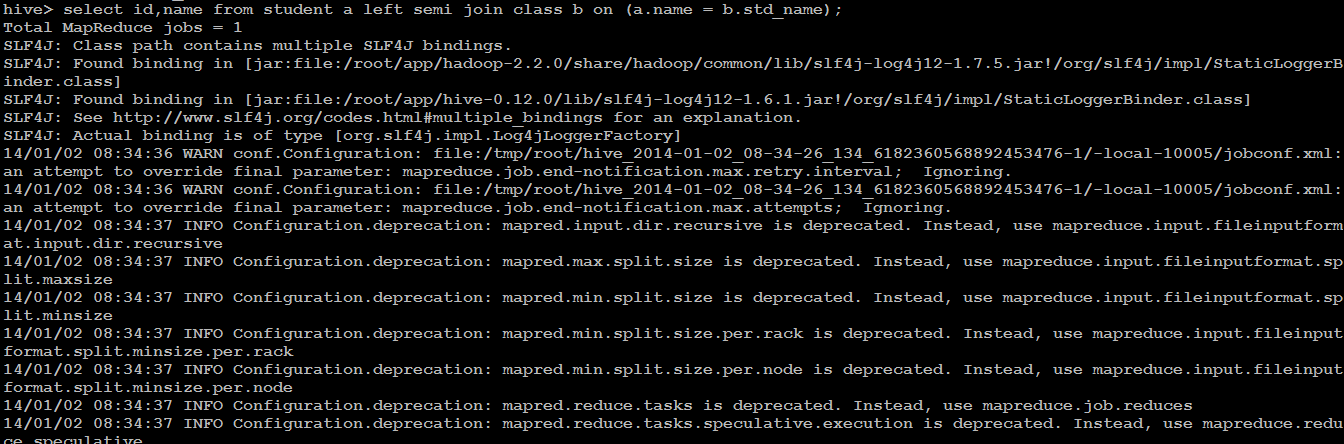
1. 获取已经分配班级的学生姓名。



1. 获取尚未分配班级的学生姓名。



1. LEFT SEMI JOIN是IN/EXISTS的高效实现。



# 3 Hive Shell参数

### 3.1 Hive命令行

* 语法结构

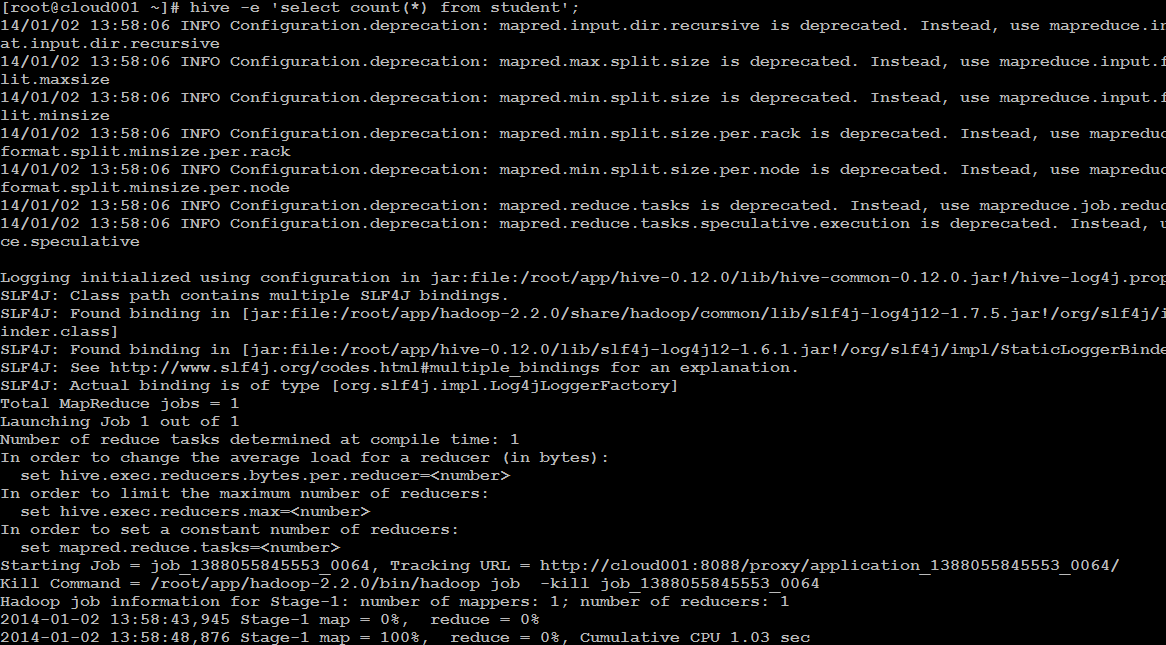
hive [-hiveconf x=y]\* [<-i filename>]\* [<-f filename>|<-e query-string>] [-S]

说明：

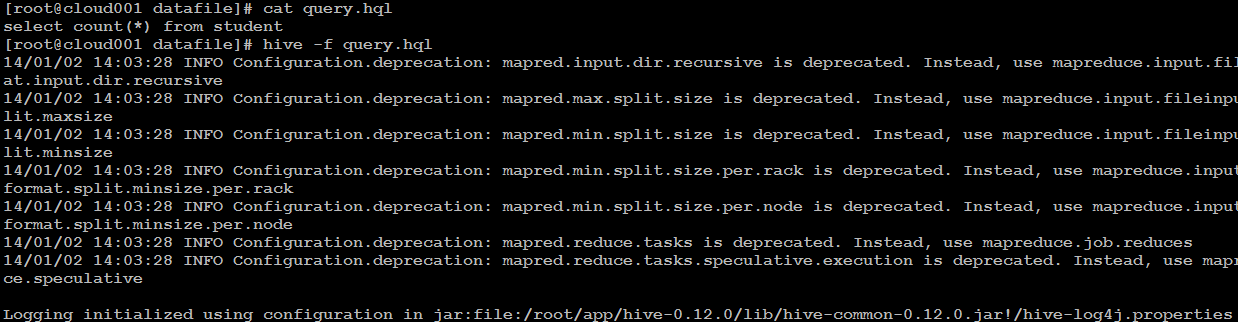
1. -i 从文件初始化HQL。
2. -e从命令行执行指定的HQL
3. -f 执行HQL脚本
4. -v 输出执行的HQL语句到控制台
5. -p <port> connect to Hive Server on port number
6. -hiveconf x=y Use this to set hive/hadoop configuration variables.

* 具体实例

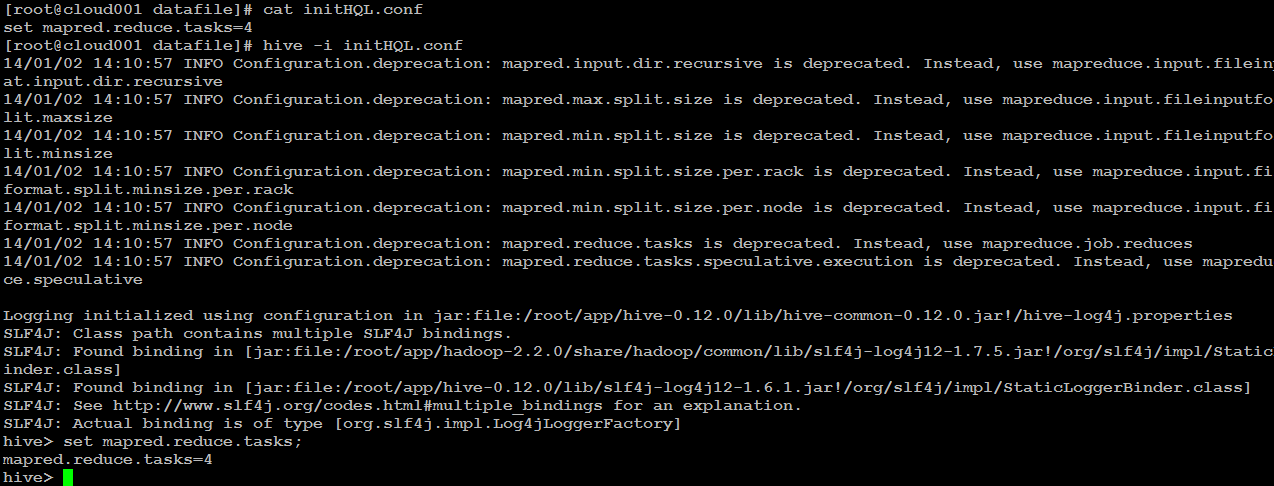
1、运行一个查询。



2、运行一个文件。



3、运行参数文件。



### 3.2 Hive参数配置方式

*Hive参数大全：*

*https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties*

开发Hive应用时，不可避免地需要设定Hive的参数。设定Hive的参数可以调优HQL代码的执行效率，或帮助定位问题。然而实践中经常遇到的一个问题是，为什么设定的参数没有起作用？这通常是错误的设定方式导致的。

**对于一般参数，有以下三种设定方式：**

* 配置文件
* 命令行参数
* 参数声明

**配置文件**：Hive的配置文件包括

* 用户自定义配置文件：$HIVE\_CONF\_DIR/hive-site.xml
* 默认配置文件：$HIVE\_CONF\_DIR/hive-default.xml

用户自定义配置会覆盖默认配置。

另外，Hive也会读入Hadoop的配置，因为**Hive是作为Hadoop的客户端启动的，Hive的配置会覆盖Hadoop的配置。**

配置文件的设定对本机启动的所有Hive进程都有效。

**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，例如：

**bin/hive -hiveconf hive.root.logger=INFO,console**

这一设定对本次启动的Session（对于Server方式启动，则是所有请求的Sessions）有效。

**参数声明**：可以在HQL中使用SET关键字设定参数，例如：

**set mapred.reduce.tasks=100;**

这一设定的作用域也是**session级**的。

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

# 4. Hive函数

## 4.1 内置运算符

*内容较多，见《Hive官方文档》*

## 4.2 内置函数

*内容较多，见《Hive官方文档》*

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

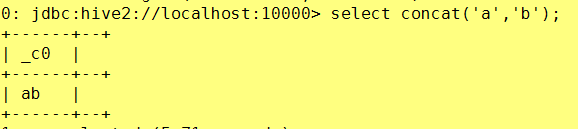
测试各种内置函数的快捷方法：

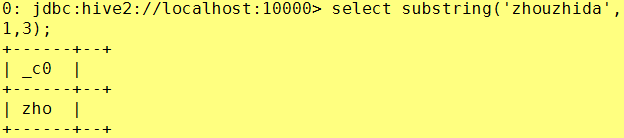
1. 创建一个dual表

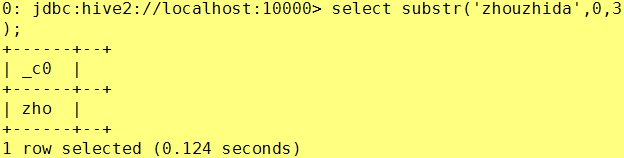
create table dual(id string);

1. load一个文件（一行，一个空格）到dual表
2. select substr('angelababy',2,3) from dual;

**其实可以这样**：直接在cli里面测试函数如select concat(‘a’,’b’);







## 4.3 Hive自定义函数和Transform

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

### 4.3.1 自定义函数类别

UDF 作用于单个数据行，产生一个数据行作为输出。（数学函数，字符串函数）

UDAF（用户定义聚集函数）：接收多个输入数据行，并产生一个输出数据行。（count，max）

### 4.3.2 UDF开发实例

1、先开发一个java类，继承UDF，并重载evaluate方法

|  |
| --- |
| package cn.itcast.bigdata.udf  import org.apache.hadoop.hive.ql.exec.UDF;  import org.apache.hadoop.io.Text;  public final class Lower extends UDF{  public Text evaluate(final Text s){  if(s==null){return null;}  return new Text(s.toString().toLowerCase());  }  } |

2、打成jar包上传到服务器

3、将jar包添加到hive的classpath

hive>add JAR /home/hadoop/udf.jar;

1. 创建临时函数与开发好的java class关联

|  |
| --- |
| Hive>create temporary function toLow as 'cn.itcast.bigdata.udf. toLowerCase; |

1. 即可在hql中使用自定义的函数strip

Select toLow (name),age from t\_test;

**案例：将rating.json文件中的数据解析到hive’表中：**

原始数据：

{"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}

{"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}

现在hive中建一个初始表:create table t\_json(line string);

然后将rating.json数据load到t\_json中；

然后自定义UDF：

|  |
| --- |
| package cn.jxust.bigdata.hiveudf;  import java.io.IOException;  import org.apache.hadoop.hive.ql.exec.UDF;  import org.codehaus.jackson.JsonParseException;  import org.codehaus.jackson.map.JsonMappingException;  import org.codehaus.jackson.map.ObjectMapper;  /\*  \* 解析json  \* {"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}  \*  \* 将解析成的一行赋给新表的四个字段：  \* insert overwrite table t\_rating  select split(parsejson(line),'\t')[0]as movieid,split(parsejson(line),'\t')[1] as rate,split(parsejson(line),'\t')[2] as timestring,split(parsejson(line),'\t')[3] as uid from rat\_json limit 10;    \*hive中自带解析：select get\_json\_object(line,'$.movie') as moive,get\_json\_object(line,'$.rate') as rate from rat\_json limit 10;  \*/  public class JsonParser extends UDF{  public String evaluate(String jsonLine){  ObjectMapper objectMapper = new ObjectMapper();//使用jkson工具类解析  MovieRateBean bean;  try {  bean = objectMapper.readValue(jsonLine, MovieRateBean.class);//将一行数据解析为MovieRateBean类型  return bean.toString();  } catch (Exception e) {  return "";  }      }  }  package cn.jxust.bigdata.hiveudf;  /\*  \* movie bean javabean中字段要与json中的字段名相同才能解析！！  \* {"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}  \*/  public class MovieRateBean {  private String movie;  private String rate;  private String timeStamp;  private String uid;  public String getMovie() {  return movie;  }  public void setMovie(String movie) {  this.movie = movie;  }  public String getRate() {  return rate;  }  public void setRate(String rate) {  this.rate = rate;  }  public String getTimeStamp() {  return timeStamp;  }  public void setTimeStamp(String timeStamp) {  this.timeStamp = timeStamp;  }  public String getUid() {  return uid;  }  public void setUid(String uid) {  this.uid = uid;  }  @Override  public String toString() {  return movie+"\t"+rate+"\t"+timeStamp+"\t"+uid;  }    } |

然后打成jar放到hive命令行中，并自定义函数名为create temporary jsonparse….

因为解析后还是一行数据，想要分成四个字段放到新表中需要分割一下，所以：

Create table t\_moive

As

select split(jsonparse(line),'\t')[0],split(jsonparse(line),'\t')[1],split(jsonparse(line),'\t')[2],split(jsonparse(line),'\t')[3] from t\_json limit 10;

json的内置函数：

select get\_json\_object(line,'$.movie') as moive,get\_json\_object(line,'$.rate') as rate from rat\_json limit 10;

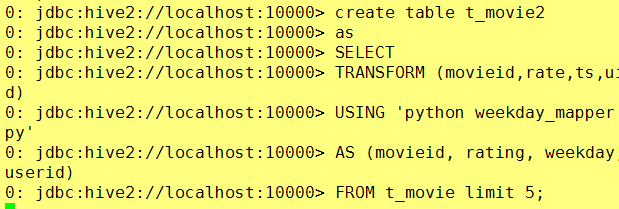
### 4.3.3 Transform实现

Hive的 TRANSFORM 关键字***提供了在SQL中调用自写脚本的功能***

适合实现Hive中没有的功能又不想写UDF的情况

使用示例1：下面这句sql就是借用了weekday\_mapper.py对数据进行了处理.

|  |
| --- |
| CREATE TABLE u\_data\_new (  movieid INT,  rating INT,  weekday INT,  userid INT)  ROW FORMAT DELIMITED  FIELDS TERMINATED BY '\t';  add FILE weekday\_mapper.py;  INSERT OVERWRITE TABLE u\_data\_new  SELECT  TRANSFORM (movieid, rating, unixtime,userid)旧表中的字段  USING 'python weekday\_mapper.py'  AS (movieid, rating, weekday,userid) 新表中的字段  FROM u\_data; |



其中weekday\_mapper.py内容如下

|  |
| --- |
| #!/bin/python  import sys  import datetime  for line in sys.stdin:  line = line.strip()  movieid, rating, unixtime,userid = line.split('\t')  weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()  print '\t'.join([movieid, rating, str(weekday),userid]) |

使用示例2：下面的例子则是使用了**shell的cat命令**来处理数据

|  |
| --- |
| FROM invites a INSERT OVERWRITE TABLE events SELECT TRANSFORM(a.foo, a.bar) AS (oof, rab) USING '/bin/cat' WHERE a.ds > '2008-08-09'; |

# 5. Hive实战

## Hive 实战案例1——数据ETL

### 需求：

* 对web点击流日志基础数据表进行etl（按照仓库模型设计）
* 按各时间维度统计来源域名top10

已有数据表 “t\_orgin\_weblog” ：

|  |
| --- |
| +------------------+------------+----------+--+  | col\_name | data\_type | comment |  +------------------+------------+----------+--+  | valid | string | |  | remote\_addr | string | |  | remote\_user | string | |  | time\_local | string | |  | request | string | |  | status | string | |  | body\_bytes\_sent | string | |  | http\_referer | string | |  | http\_user\_agent | string | |  +------------------+------------+----------+--+ |

### 数据示例：

|  |
| --- |
| | true|1.162.203.134| - | 18/Sep/2013:13:47:35| /images/my.jpg | 200| 19939 | "http://www.angularjs.cn/A0d9" | "Mozilla/5.0 (Windows |  | true|1.202.186.37 | - | 18/Sep/2013:15:39:11| /wp-content/uploads/2013/08/windjs.png| 200| 34613 | "http://cnodejs.org/topic/521a30d4bee8d3cb1272ac0f" | "Mozilla/5.0 (Macintosh;| |

### 实现步骤：

1、对原始数据进行抽取转换

--将来访url分离出host path query query id

|  |
| --- |
| drop table if exists t\_etl\_referurl;  create table t\_etl\_referurl as  SELECT a.\*,b.\*  FROM t\_orgin\_weblog a LATERAL VIEW parse\_url\_tuple(regexp\_replace(http\_referer, "\"", ""), 'HOST', 'PATH','QUERY', 'QUERY:id') b as host, path, query, query\_id |

3、从前述步骤进一步分离出日期时间形成ETL明细表“t\_etl\_detail” day tm

|  |
| --- |
| drop table if exists t\_etl\_detail;  create table t\_etl\_detail as  select b.\*,substring(time\_local,0,11) as daystr,  substring(time\_local,13) as tmstr,  substring(time\_local,4,3) as month,  substring(time\_local,0,2) as day,  substring(time\_local,13,2) as hour  from t\_etl\_referurl b; |

3、对etl数据进行分区(包含所有数据的结构化信息)

|  |
| --- |
| drop table t\_etl\_detail\_prt;  create table t\_etl\_detail\_prt(  valid string,  remote\_addr string,  remote\_user string,  time\_local string,  request string,  status string,  body\_bytes\_sent string,  http\_referer string,  http\_user\_agent string,  host string,  path string,  query string,  query\_id string,  daystr string,  tmstr string,  month string,  day string,  hour string)  partitioned by (mm string,dd string); |

导入数据

|  |
| --- |
| insert into table t\_etl\_detail\_prt partition(mm='Sep',dd='18')  select \* from t\_etl\_detail where daystr='18/Sep/2013';  insert into table t\_etl\_detail\_prt partition(mm='Sep',dd='19')  select \* from t\_etl\_detail where daystr='19/Sep/2013'; |

分个时间维度统计各referer\_host的访问次数并排序

|  |
| --- |
| create table t\_refer\_host\_visit\_top\_tmp as  select referer\_host,count(\*) as counts,mm,dd,hh from t\_display\_referer\_counts group by hh,dd,mm,referer\_host order by hh asc,dd asc,mm asc,counts desc; |

4、来源访问次数topn各时间维度URL

取各时间维度的referer\_host访问次数topn

|  |
| --- |
| select \* from (select referer\_host,counts,concat(hh,dd),row\_number() over (partition by concat(hh,dd) order by concat(hh,dd) asc) as od from t\_refer\_host\_visit\_top\_tmp) t where od<=3; |

## Hive 实战案例2——访问时长统计

### 需求：

从web日志中统计每日访客平均停留时间

### 实现步骤：

1. 由于要从大量请求中分辨出用户的各次访问，逻辑相对复杂，通过hive直接实现有困难，因此编写一个mr程序来求出访客访问信息（详见代码）

启动mr程序获取结果：

|  |
| --- |
| [hadoop@hdp-node-01 ~]$ hadoop jar weblog.jar cn.itcast.bigdata.hive.mr.UserStayTime /weblog/input /weblog/stayout |

1. 将mr的处理结果导入hive表

|  |
| --- |
| drop table t\_display\_access\_info\_tmp;  create table t\_display\_access\_info\_tmp(remote\_addr string,firt\_req\_time string,last\_req\_time string,stay\_long bigint)  row format delimited fields terminated by '\t';  load data inpath '/weblog/stayout4' into table t\_display\_access\_info\_tmp; |

3、得出访客访问信息表 "t\_display\_access\_info"

由于有一些访问记录是单条记录，mr程序处理处的结果给的时长是0，所以考虑给单次请求的停留时间一个默认市场30秒

|  |
| --- |
| drop table t\_display\_access\_info;  create table t\_display\_access\_info as  select remote\_addr,firt\_req\_time,last\_req\_time,  case stay\_long  when 0 then 30000  else stay\_long  end as stay\_long  from t\_display\_access\_info\_tmp; |

4、统计所有用户停留时间平均值

select avg(stay\_long) from t\_display\_access\_info;

## Hive实战案例3——级联求和

### 需求：

有如下访客访问次数统计表 t\_access\_times

|  |  |  |
| --- | --- | --- |
| 访客 | 月份 | 访问次数 |
| A | 2015-01-02 | 5 |
| A | 2015-01-03 | 15 |
| B | 2015-01-01 | 5 |
| A | 2015-01-04 | 8 |
| B | 2015-01-05 | 25 |
| A | 2015-01-06 | 5 |
| A | 2015-02-02 | 4 |
| A | 2015-02-06 | 6 |
| B | 2015-02-06 | 10 |
| B | 2015-02-07 | 5 |
| …… | …… | …… |

需要输出报表：t\_access\_times\_accumulate

|  |  |  |  |
| --- | --- | --- | --- |
| 访客 | 月份 | 月访问总计 | 累计访问总计 |
| A | 2015-01 | 33 | 33 |
| A | 2015-02 | 10 | 43 |
| ……. | ……. | ……. | ……. |
| B | 2015-01 | 30 | 30 |
| B | 2015-02 | 15 | 45 |
| ……. | ……. | ……. | ……. |

### 实现步骤

可以用一个hql语句即可实现：

|  |
| --- |
| select A.username,A.month,max(A.salary) as salary,sum(B.salary) as accumulate  from  (select username,month,sum(salary) as salary from t\_access\_times group by username,month) A  inner join  (select username,month,sum(salary) as salary from t\_access\_times group by username,month) B  on  A.username=B.username  where B.month <= A.month  group by A.username,A.month  order by A.username,A.month; |