## 一、 数据仓库

### 1． 数据仓库的基本概念

数据仓库，

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）。它出于分析性报告和决策支持目的而创建。

数据仓库本身并不“生产”任何数据，同时自身也不需要“消费”任何的数据，数据来源于外部，并且开放给外部应用，这也是为什么叫“仓库”，而不叫“工厂”的原因。

### 2． 数据仓库的主要特征

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。

#### 2.1． 面向主题

传统数据库中，最大的特点是面向应用进行数据的组织，各个业务系统可能是相互分离的。而数据仓库则是面向主题的。主题是一个抽象的概念，是较高层次上企业信息系统中的数据综合、归类并进行分析利用的抽象。在逻辑意义上，它是对应企业中某一宏观分析领域所涉及的分析对象。

操作型处理（传统数据）对数据的划分并不适用于决策分析。而基于主题组织的数据则不同，它们被划分为各自独立的领域，每个领域有各自的逻辑内涵但互不交叉，在抽象层次上对数据进行完整、一致和准确的描述。一些主题相关的数据通常分布在多个操作型系统中。

#### 2.2． 集成性

通过对分散、独立、异构的数据库数据进行抽取、清理、转换和汇总便得到了数据仓库的数据，这样保证了数据仓库内的数据关于整个企业的一致性。

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，必然要经过统一与综合，这一步是数据仓库建设中最关键、最复杂的一步，所要完成的工作有：

（1）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。

（2）进行数据综合和计算。数据仓库中的数据综合工作可以在从原有数据库抽取数据时生成，但许多是在数据仓库内部生成的，即进入数据仓库以后进行综合生成的。

下图说明一个保险公司综合数据的简单处理过程，其中数据仓库中与“保险” 主题有关的数据来自于多个不同的操作型系统。这些系统内部数据的命名可能不同，数据格式也可能不同。把不同来源的数据存储到数据仓库之前，需要去除这些不一致。



#### 2.3． 非易失性（不可更新性）

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

的数据库快照的集合，以及基于这些快照进行统计、综合和重组的导出数据。

数据非易失性主要是针对应用而言。数据仓库的用户对数据的操作大多是数据查询或比较复杂的挖掘，一旦数据进入数据仓库以后，一般情况下被较长时间保留。数据仓库中一般有大量的查询操作，但修改和删除操作很少。因此，数据经加工和集成进入数据仓库后是极少更新的，通常只需要定期的加载和更新。

#### 2.4． 时变性

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更新，以适应决策的需要。从这个角度讲，数据仓库建设是一个项目，更是一个过程 。数据仓库的数据随时间的变化表现在以下几个方面。

（1） 数据仓库的数据时限一般要远远长于操作型数据的数据时限。

（2） 操作型系统存储的是当前数据，而数据仓库中的数据是历史数据。

（3） 数据仓库中的数据是按照时间顺序追加的，它们都带有时间属性。

### 3． 数据仓库与数据库区别

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。

首先要明白，数据仓库的出现，并不是要取代数据库。

* 数据库是面向事务的设计，数据仓库是面向主题设计的。
* 数据库一般存储业务数据，数据仓库存储的一般是历史数据。
* 数据库设计是尽量避免冗余，一般针对某一业务应用进行设计，比如一张简单的User表，记录用户名、密码等简单数据即可，符合业务应用，但是不符合分析。数据仓库在设计是有意引入冗余，依照分析需求，分析维度、分析指标进行设计。
* 数据库是为捕获数据而设计，数据仓库是为分析数据而设计。

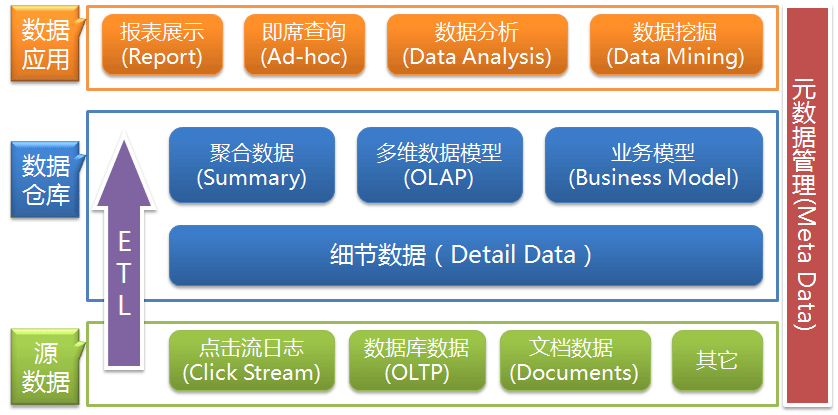
以银行业务为例。数据库是事务系统的数据平台，客户在银行做的每笔交易都会写入数据库，被记录下来，这里，可以简单地理解为用数据库记账。数据仓库是分析系统的数据平台，它从事务系统获取数据，并做汇总、加工，为决策者提供决策的依据。比如，某银行某分行一个月发生多少交易，该分行当前存款余额是多少。如果存款又多，消费交易又多，那么该地区就有必要设立ATM了。

显然，银行的交易量是巨大的，通常以百万甚至千万次来计算。事务系统是实时的，这就要求时效性，客户存一笔钱需要几十秒是无法忍受的，这就要求数据库只能存储很短一段时间的数据。而分析系统是事后的，它要提供关注时间段内所有的有效数据。这些数据是海量的，汇总计算起来也要慢一些，但是，只要能够提供有效的分析数据就达到目的了。

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，它决不是所谓的“大型数据库”。

### 4、数据仓库分层架构

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。

为什么要对数据仓库分层？

用空间换时间，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据；不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大。

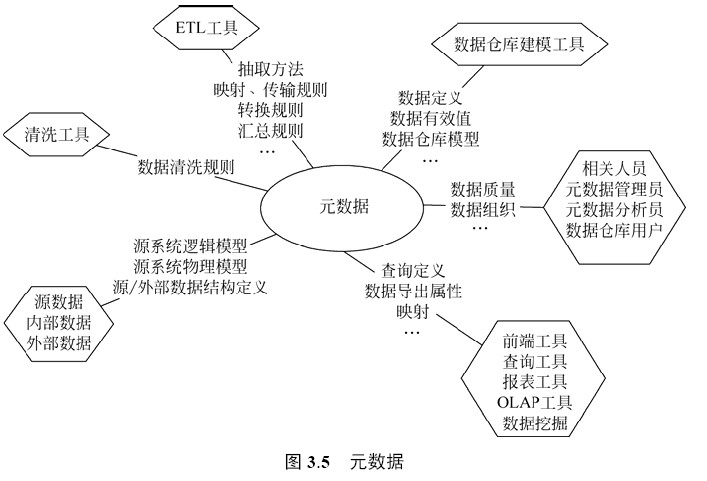
通过数据分层管理可以简化数据清洗的过程，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

### 5、数据仓库元数据管理

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和业务元数据。技术元数据为开发和管理数据仓库的IT 人员使用，它描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。而业务元数据为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

由上可见，元数据不仅定义了数据仓库中数据的模式、来源、抽取和转换规则等，而且是整个数据仓库系统运行的基础，元数据把数据仓库系统中各个松散的组件联系起来，组成了一个有机的整体。

## 二、Hive基本概念

### 2.1、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。

功能扩展很方便。

#### 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

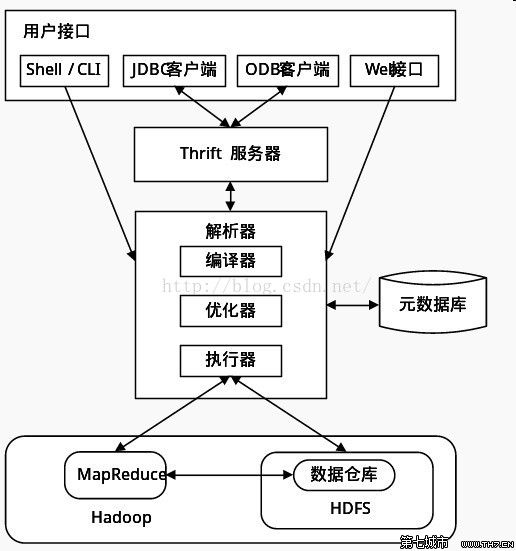
Hive支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

* 容错

良好的容错性，节点出现问题SQL仍可完成执行。

### 2.2、 Hive架构

#### 架构图



#### 基本组成

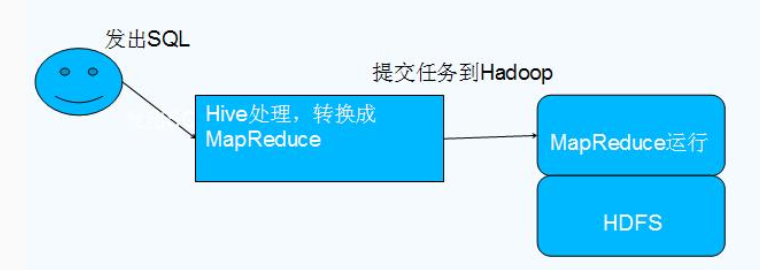
用户接口：包括CLI、JDBC/ODBC、WebGUI。其中，CLI(command line interface)为shell命令行；JDBC/ODBC是Hive的JAVA实现，与传统数据库JDBC类似；WebGUI是通过浏览器访问Hive。

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。

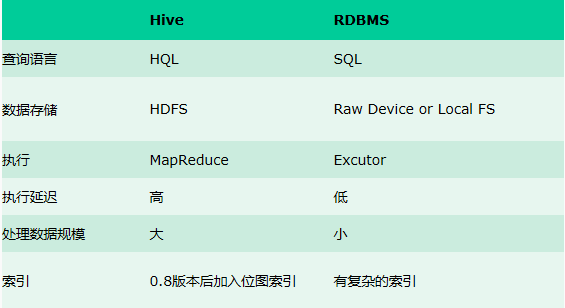
解释器、编译器、优化器、执行器:完成HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在HDFS 中，并在随后有MapReduce 调用执行。

### 2.3、Hive与Hadoop的关系

Hive利用HDFS存储数据，利用MapReduce查询分析数据



### 2.4、Hive与传统数据库对比 hive用于海量数据的离线数据分析



*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*

### 2.5、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）

SequenceFile是hadoop中的一种文件格式：

文件内容是以序列化的kv对象来组织的

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：与table类似，不过其数据存放位置可以在任意指定路径
* partition：在hdfs中表现为table目录下的子目录
* bucket：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

### 2.6、HIVE的安装部署

我们在此处选择第三台机器作为我们hive的安装机器

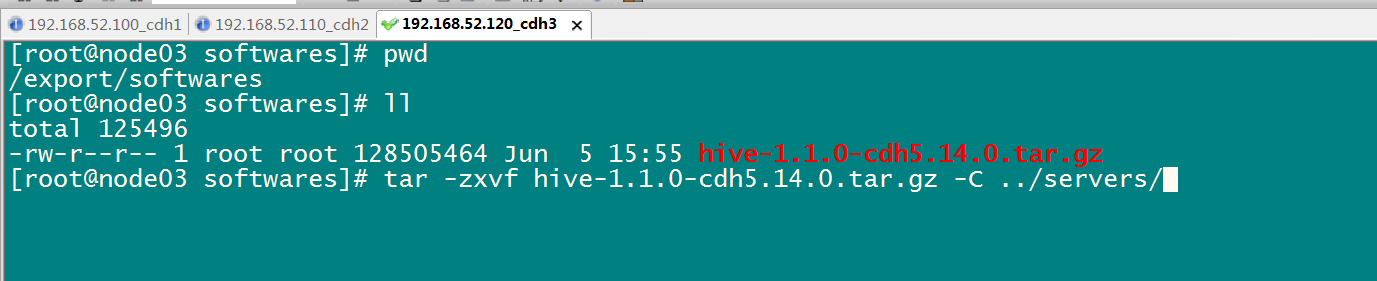
#### 2.6.1 安装

##### 2.6.1.1、derby版hive直接使用：

1、解压hive

cd /export/softwares

tar -zxvf hive-1.1.0-cdh5.14.0.tar.gz -C ../servers/



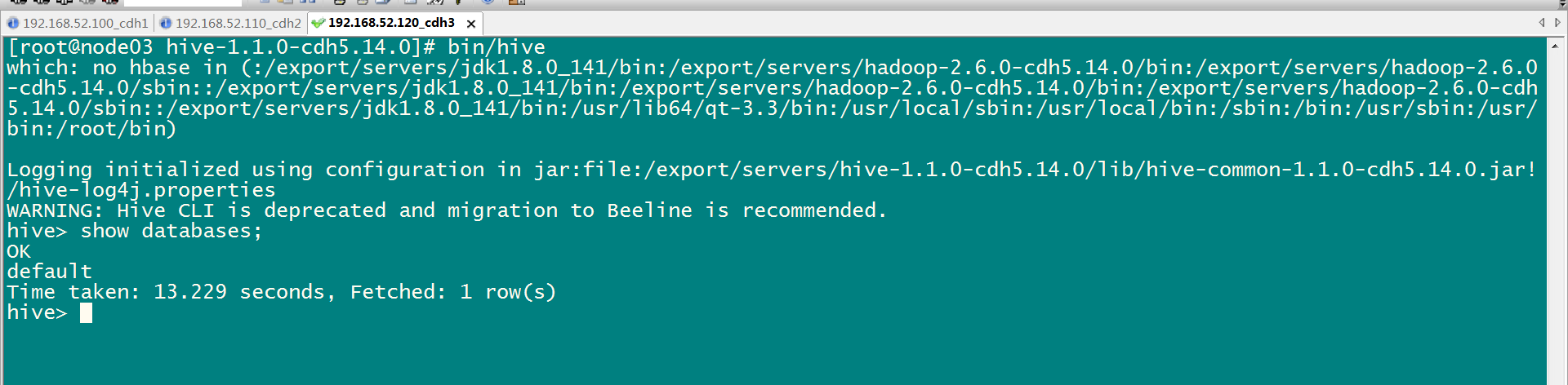
1. 直接启动 bin/hive

cd ../servers/

cd hive-1.1.0-cdh5.14.0/

bin/hive

hive> create database mytest;



缺点：多个地方安装hive后，每一个hive是拥有一套自己的元数据，大家的库、表就不统一；

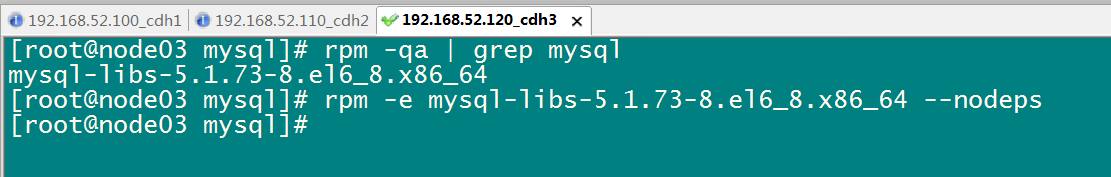
##### 2.6.1.2、使用mysql共享hive元数据

###### mysql数据库的安装（使用rpm包的方式进行安装，不推荐）

第一步：查看系统自带的mysql的rpm包

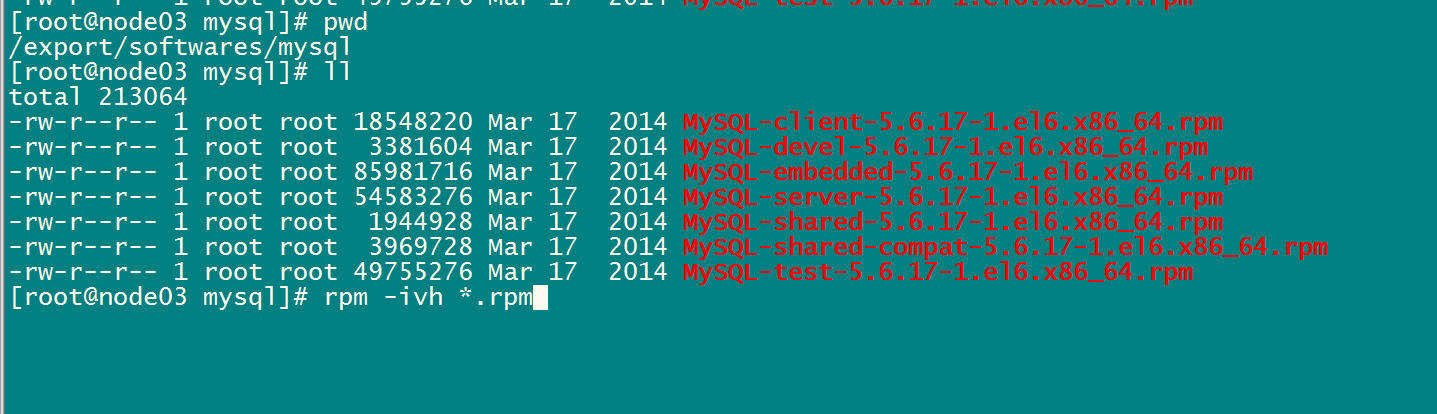
rpm -qa | grep mysql

rpm -e mysql-libs-5.1.73-8.el6\_8.x86\_64 --nodeps



第二步：安装mysql的rpm包

rpm -ivh \*.rpm

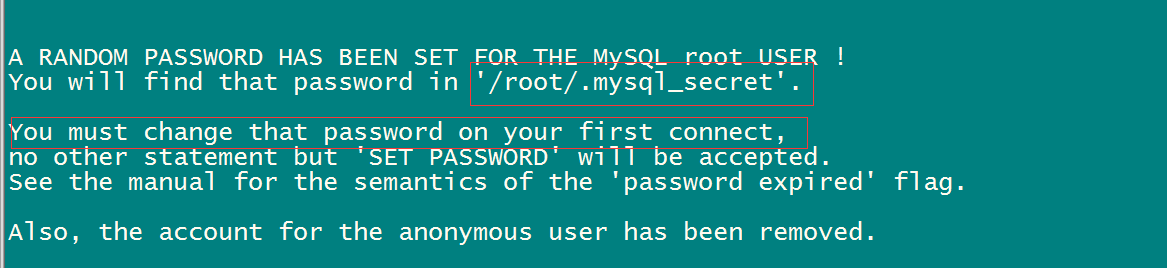


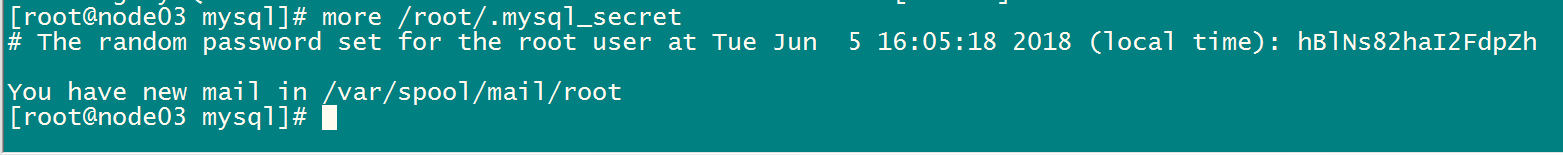
第三步：启动mysql的服务

service mysqld start

第四步：查看mysql初始化密码

more /root/.mysql\_secret





第五步：使用随记密码登录mysql并更新密码

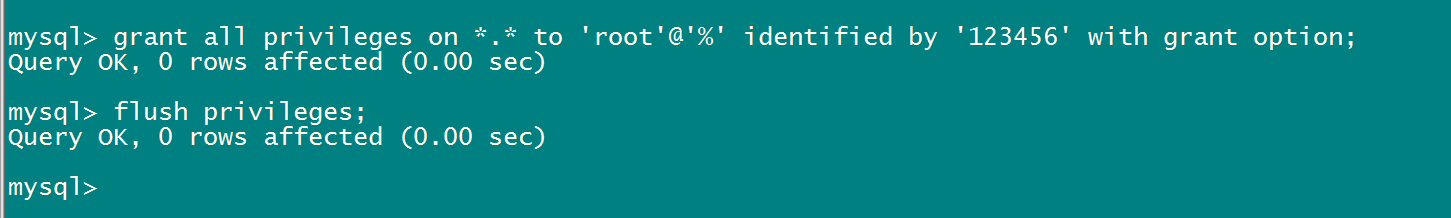
mysql -uroot –p

SET PASSWORD = PASSWORD('123456');

第六步：开启mysql远程连接

grant all privileges on \*.\* to 'root'@'%' identified by '123456' with grant option;

flush privileges;



第七步：设置mysql的开机启动

chkconfig --add mysqld

chkconfig mysqld on

###### mysql数据库的安装（使用yum源进行安装，强烈推荐）

第一步：在线安装mysql相关的软件包

yum install mysql mysql-server mysql-devel

第二步：启动mysql的服务

/etc/init.d/mysqld start

第三步：通过mysql安装自带脚本进行设置

/usr/bin/mysql\_secure\_installation

第四步：进入mysql的客户端然后进行授权

grant all privileges on \*.\* to 'root'@'%' identified by '123456' with grant option;

flush privileges;



###### 修改hive的配置文件

修改hive-env.sh

添加我们的hadoop的环境变量

cd /export/servers/hive-1.1.0-cdh5.14.0/conf

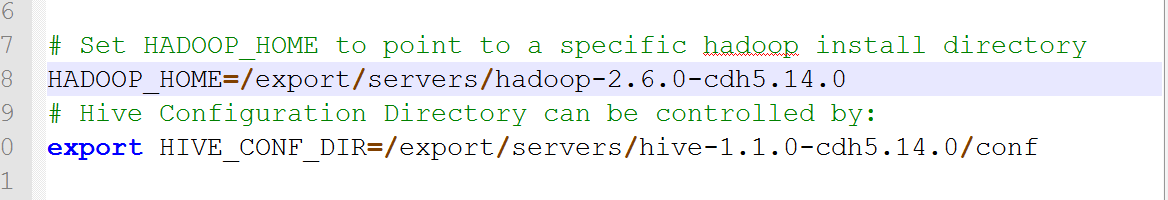
cp hive-env.sh.template hive-env.sh

vim hive-env.sh

HADOOP\_HOME=/export/servers/hadoop-2.6.0-cdh5.14.0

# Hive Configuration Directory can be controlled by:

export HIVE\_CONF\_DIR=/export/servers/hive-1.1.0-cdh5.14.0/conf



修改hive-site.xml

cd /export/servers/hive-1.1.0-cdh5.14.0/conf

vim hive-site.xml

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

<configuration>

<property>

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

<value>jdbc:mysql://node03.hadoop.com:3306/hive?createDatabaseIfNotExist=true</value>

</property>

<property>

<name>javax.jdo.option.ConnectionDriverName</name>

<value>com.mysql.jdbc.Driver</value>

</property>

<property>

<name>javax.jdo.option.ConnectionUserName</name>

<value>root</value>

</property>

<property>

<name>javax.jdo.option.ConnectionPassword</name>

<value>123456</value>

</property>

<property>

<name>hive.cli.print.current.db</name>

<value>true</value>

</property>

<property>

<name>hive.cli.print.header</name>

<value>true</value>

</property>

<property>

<name>hive.server2.thrift.bind.host</name>

<value>node03.hadoop.com</value>

</property>

<!--

<property>

<name>hive.metastore.uris</name>

<value>thrift://node03.hadoop.com:9083</value>

</property>

-->

</configuration>

上传mysql的lib驱动包

将mysql的lib驱动包上传到hive的lib目录下

cd /export/servers/hive-1.1.0-cdh5.14.0/lib

将mysql-connector-java-5.1.38.jar 上传到这个目录下

#### 2.6.2 使用方式

##### 第一种交互方式：Hive交互shell

bin/hive

查看所有的数据库

hive (default)> show databases;

创建一个数据库

hive (default)> create database myhive;

使用该数据库并创建数据库表

hive (default)> use myhive;

hive (myhive)> create table test(id int,name string);

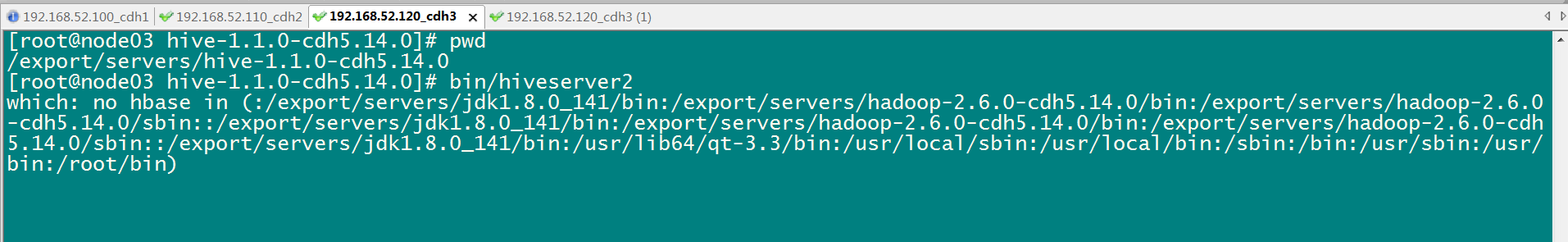
##### 第二种交互方式：Hive JDBC服务

###### 启动hiveserver2服务

前台启动

cd /export/servers/hive-1.1.0-cdh5.14.0

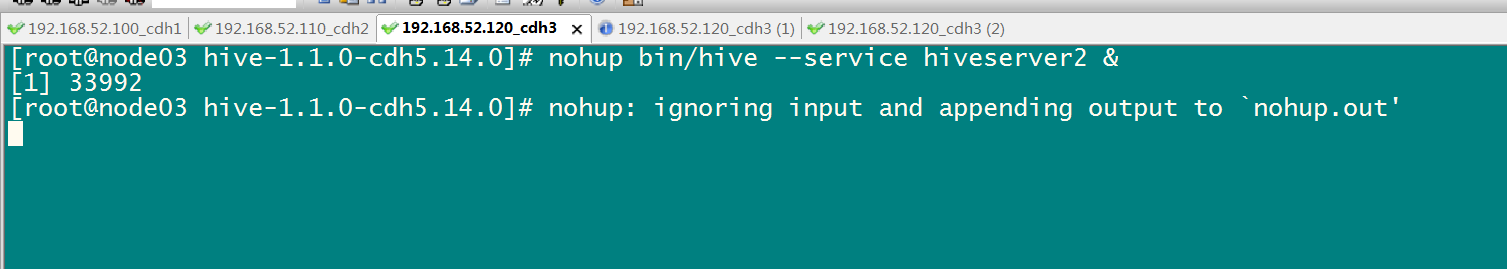
bin/hive --service hiveserver2



后台启动

cd /export/servers/hive-1.1.0-cdh5.14.0

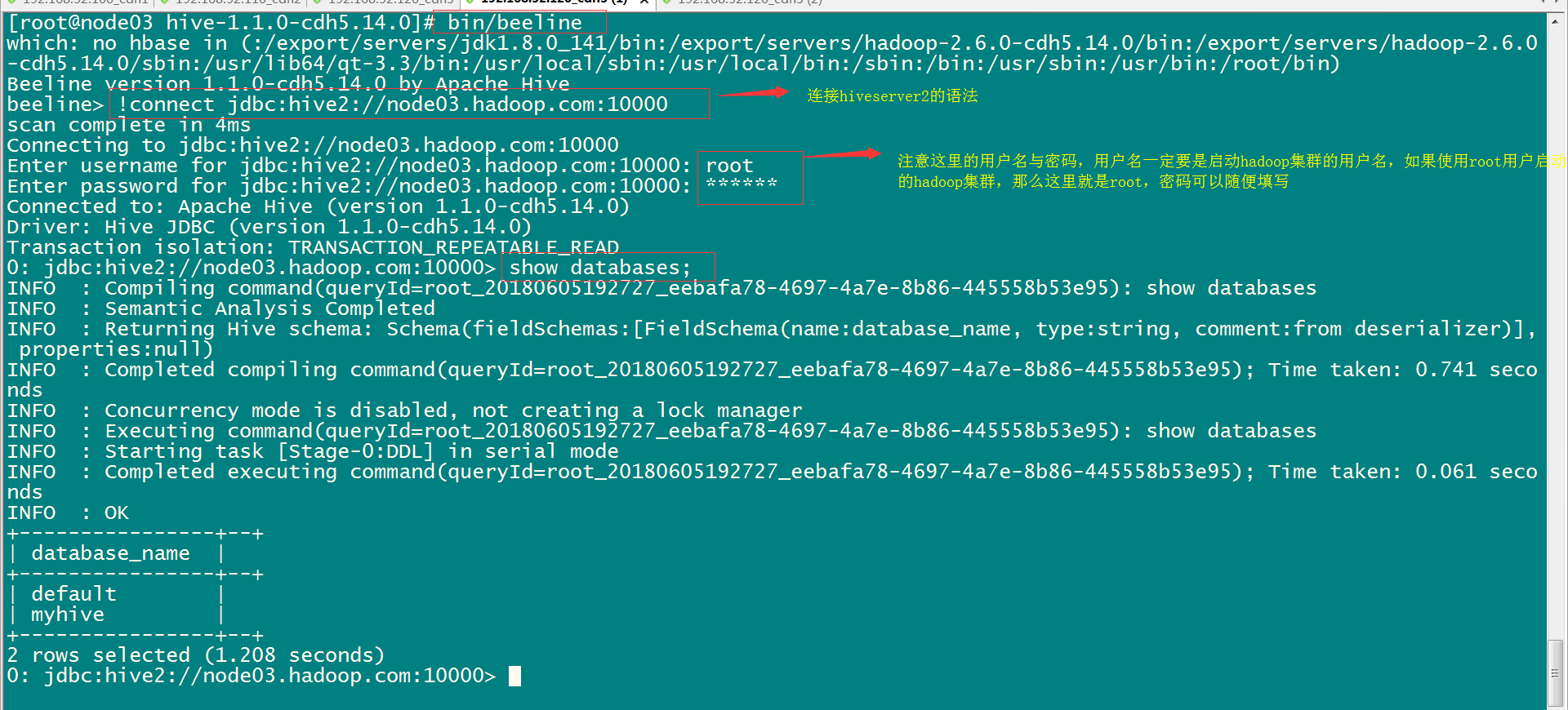
nohup bin/hive --service hiveserver2 &



###### beeline连接hiveserver2

bin/beeline

beeline> !connect jdbc:hive2://node01.hadoop.com:10000



注意：如果使用beeline方式连接hiveserver2，一定要保证hive在mysql当中的元数据库已经创建成功，不然就会拒绝连接

##### 第三种交互方式：Hive命令

使用 –e 参数来直接执行hql的语句

bin/hive -e "use myhive;select \* from test;"

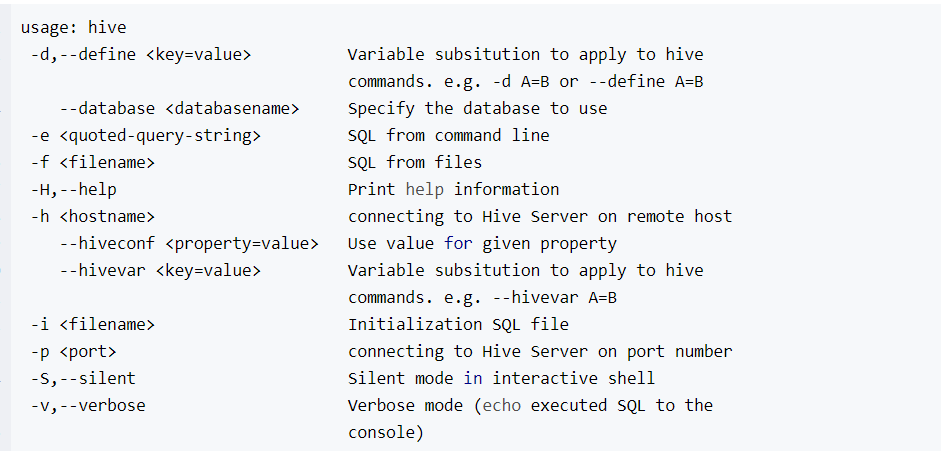
使用 –f 参数通过指定文本文件来执行hql的语句

vim hive.sql

use myhive;select \* from test;

bin/hive -f hive.sql

更多参数参考以下



## 三、Hive基本操作

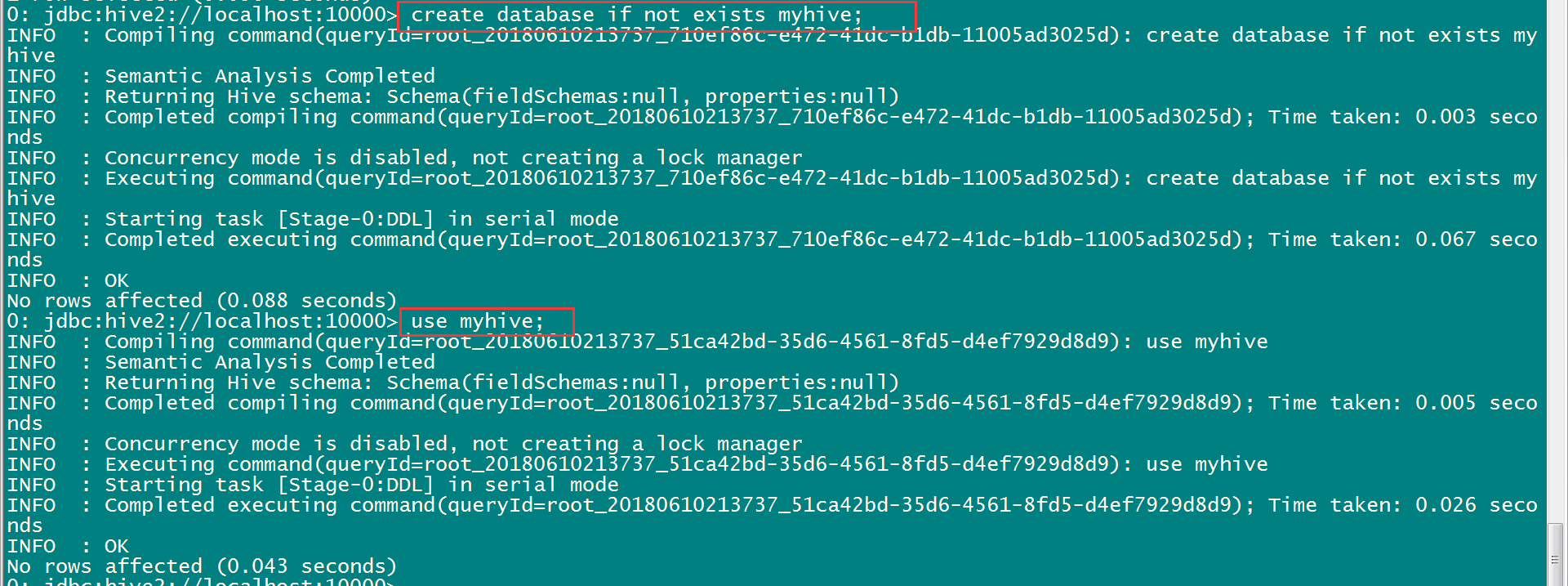
### 3.1、DDL操作

#### 创建数据库操作

##### 创建数据库

create database if not exists myhive;

use myhive;



说明：hive的表存放位置模式是由hive-site.xml当中的一个属性指定的

<name>hive.metastore.warehouse.dir</name>

<value>/user/hive/warehouse</value>

##### 创建数据库并指定hdfs存储位置

create database myhive2 location '/myhive2';

##### 修改数据库

可以使用alter database 命令来修改数据库的一些属性。但是数据库的元数据信息是不可更改的，包括数据库的名称以及数据库所在的位置

alter database myhive2 set dbproperties('createtime'='20180611');

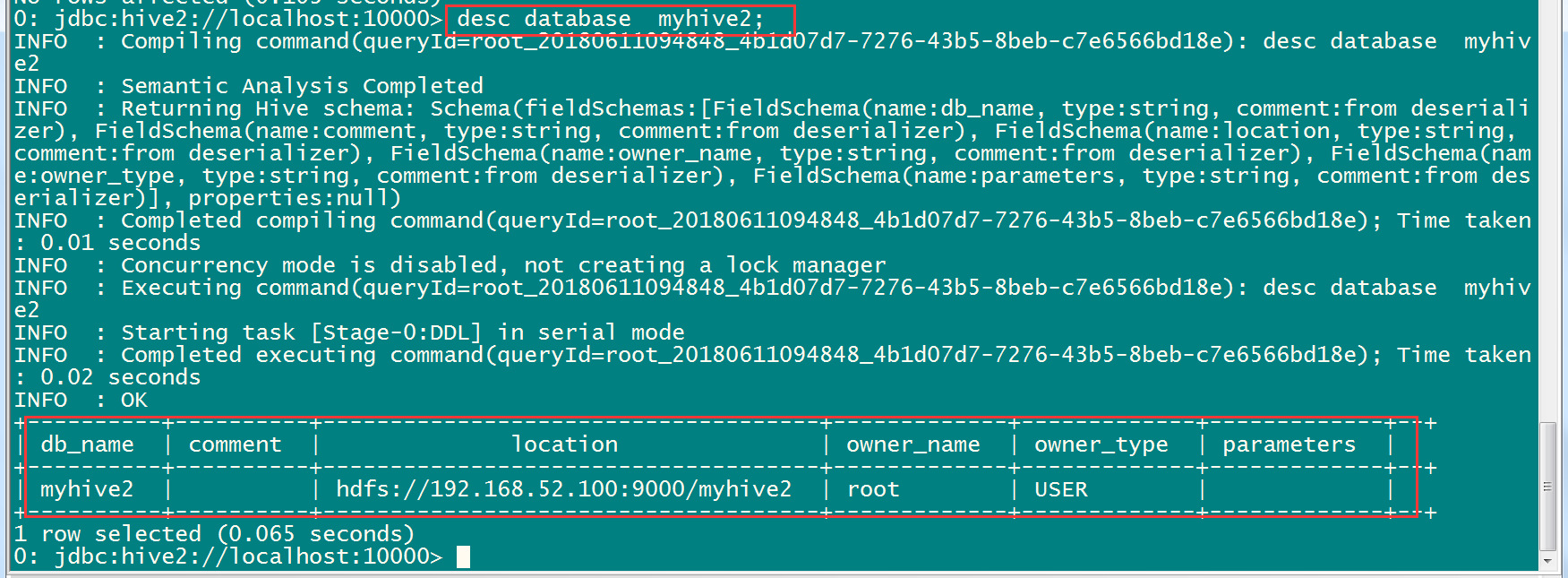
##### 查看数据库详细信息

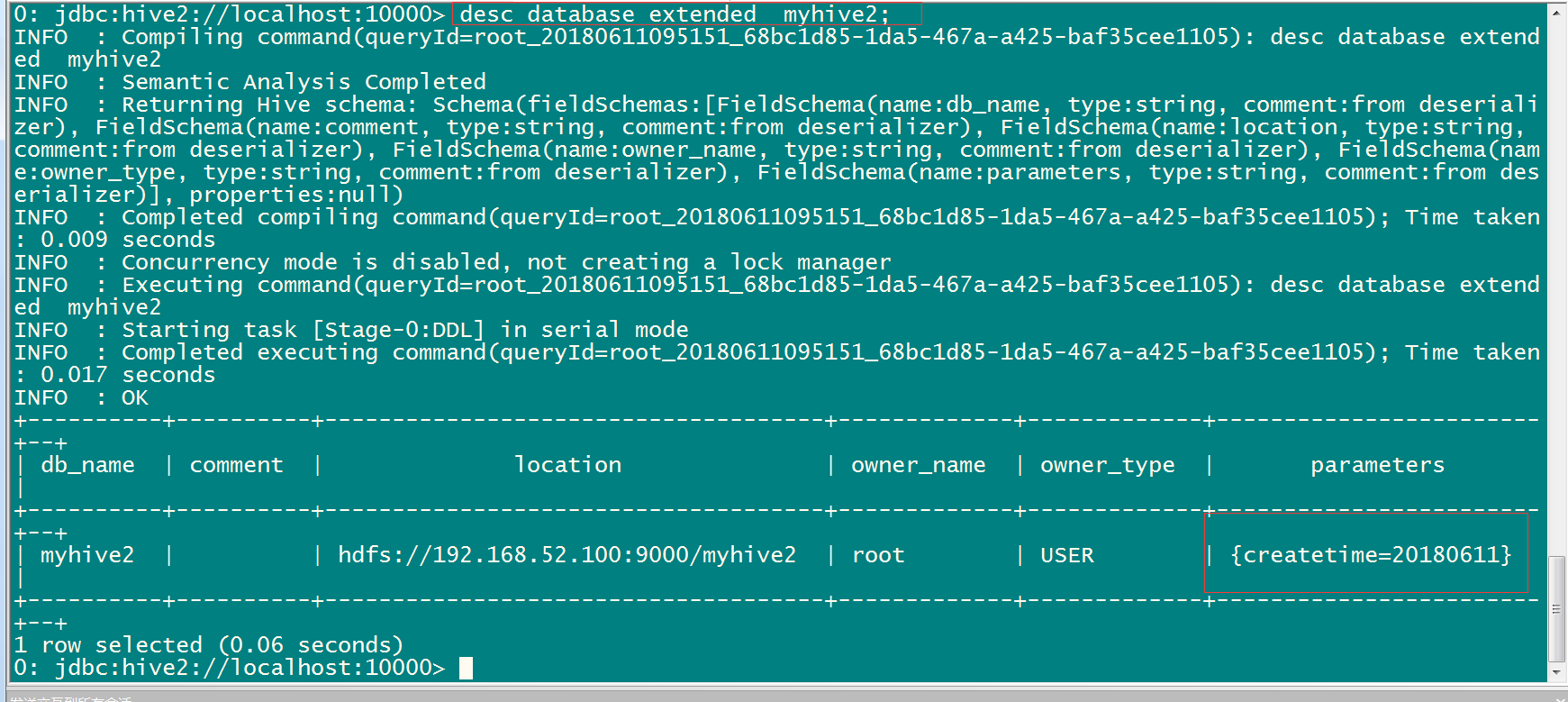
查看数据库基本信息

desc database myhive2;

查看数据库更多详细信息

desc database extended myhive2;





##### 删除数据库

删除一个空数据库，如果数据库下面有数据表，那么就会报错

drop database myhive2;

强制删除数据库，包含数据库下面的表一起删除

drop database myhive cascade; 不要执行了

#### 创建数据库表操作

##### 创建数据库表语法

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。

##### 管理表

###### hive建表

use myhive;

create table stu(id int,name string);

insert into stu values (1,"zhangsan");

select \* from stu;

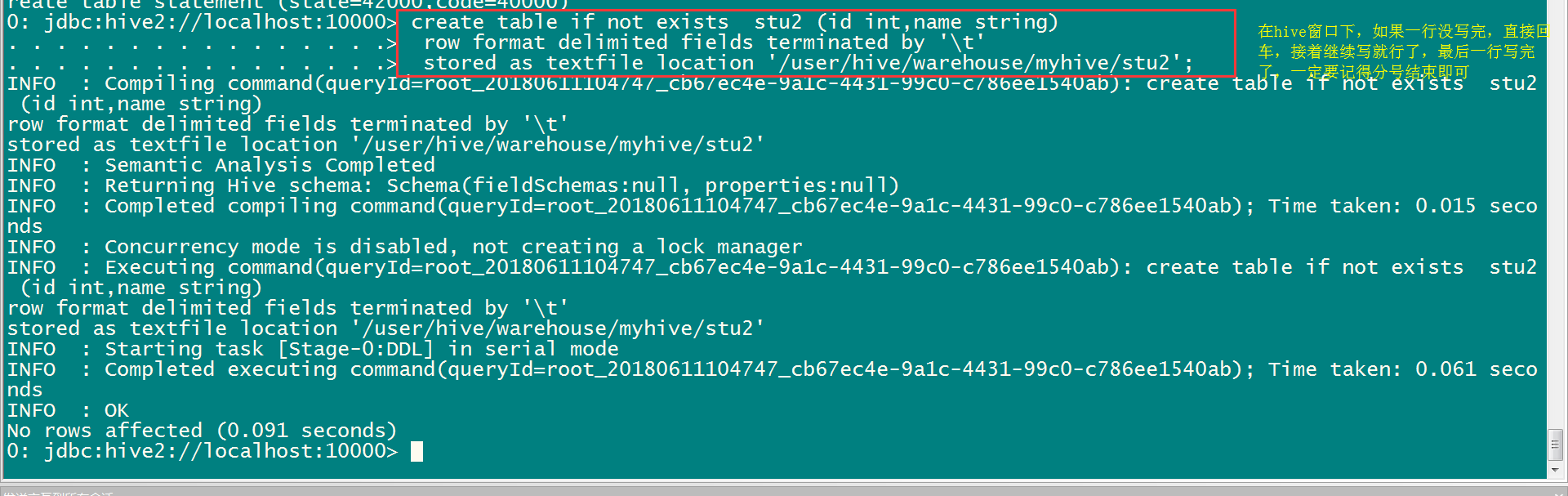
###### Hive建表时候的字段类型

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

| **分类** | **类型** | **描述** | **字面量示例** |
| --- | --- | --- | --- |
| 原始类型 | BOOLEAN | true/false | TRUE |
|  | TINYINT | 1字节的有符号整数 -128~127 | 1Y |
|  | SMALLINT | 2个字节的有符号整数，-32768~32767 | 1S |
|  | INT | 4个字节的带符号整数 | 1 |
|  | BIGINT | 8字节带符号整数 | 1L |
|  | FLOAT | 4字节单精度浮点数1.0 |  |
|  | DOUBLE | 8字节双精度浮点数 | 1.0 |
|  | DEICIMAL | 任意精度的带符号小数 | 1.0 |
|  | STRING | 字符串，变长 | “a”,’b’ |
|  | VARCHAR | 变长字符串 | “a”,’b’ |
|  | CHAR | 固定长度字符串 | “a”,’b’ |
|  | BINARY | 字节数组 | 无法表示 |
|  | TIMESTAMP | 时间戳，纳秒精度 | 122327493795 |
|  | DATE | 日期 | ‘2016-03-29’ |
|  | [INTERVAL](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types" \l "LanguageManualTypes-Intervals) | 时间频率间隔 |  |
| 复杂类型 | ARRAY | 有序的的同类型的集合 | array(1,2) |
|  | MAP | key-value,key必须为原始类型，value可以任意类型 | map(‘a’,1,’b’,2) |
|  | STRUCT | 字段集合,类型可以不同 | struct(‘1’,1,1.0), named\_stract(‘col1’,’1’,’col2’,1,’clo3’,1.0) |
|  | UNION | 在有限取值范围内的一个值 | create\_union(1,’a’,63) |

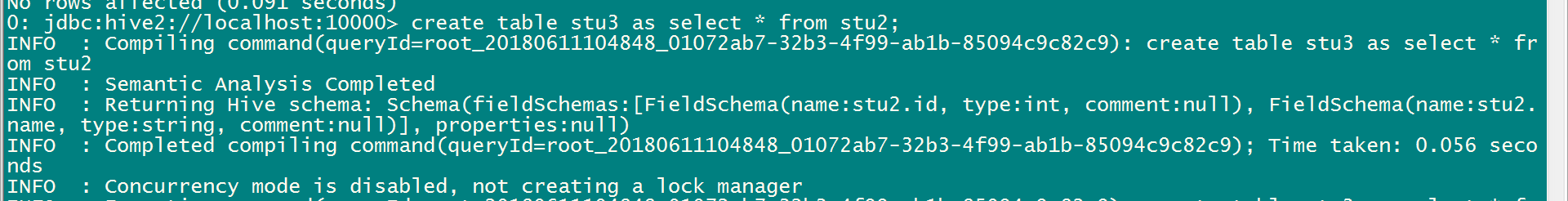
###### 创建表并指定字段之间的分隔符

create table if not exists stu2 (id int,name string) row format delimited fields terminated by '\t' stored as textfile location '/user/hive/warehouse/myhive/stu2';



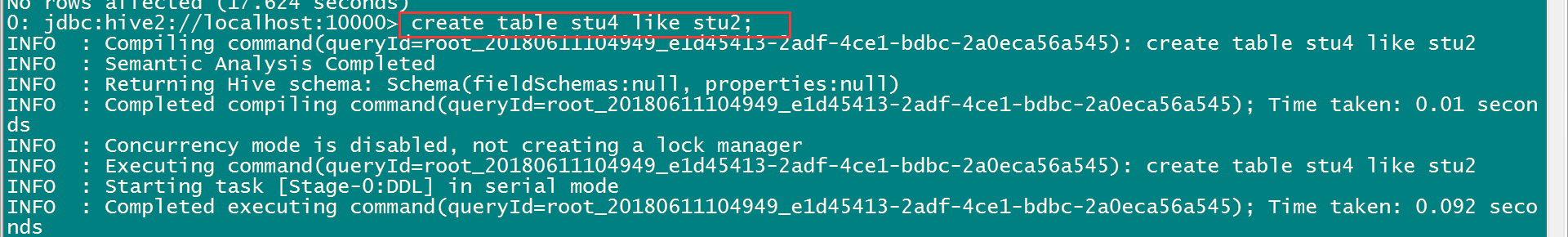
###### 根据查询结果创建表

create table stu3 as select \* from stu2;



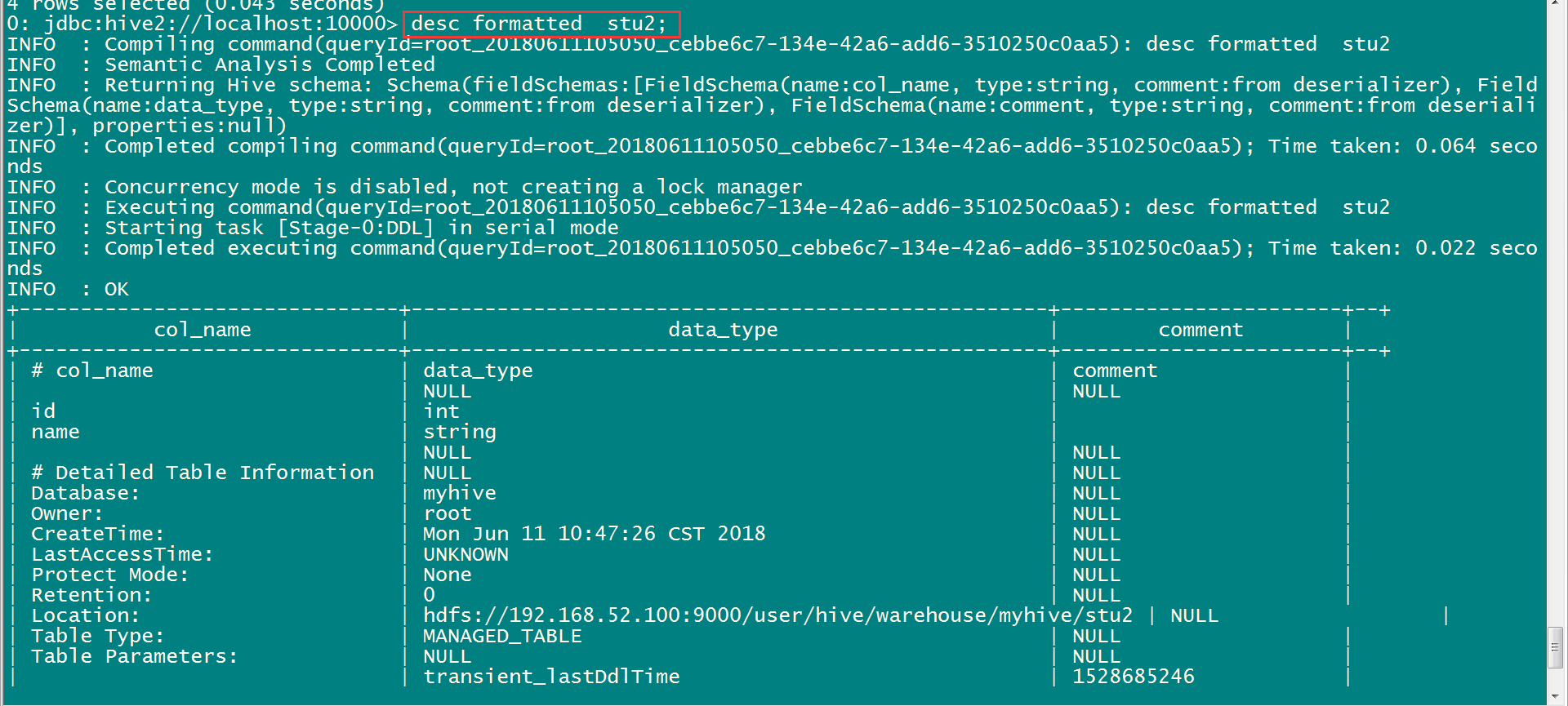
###### 根据已经存在的表结构创建表

create table stu4 like stu2;



###### 查询表的类型

desc formatted stu2;



##### 外部表：

###### 外部表说明：

外部表因为是指定其他的hdfs路径的数据加载到表当中来，所以hive表会认为自己不完全独占这份数据，所以删除hive表的时候，数据仍然存放在hdfs当中，不会删掉

###### 管理表和外部表的使用场景：

每天将收集到的网站日志定期流入HDFS文本文件。在外部表（原始日志表）的基础上做大量的统计分析，用到的中间表、结果表使用内部表存储，数据通过SELECT+INSERT进入内部表。

###### 操作案例

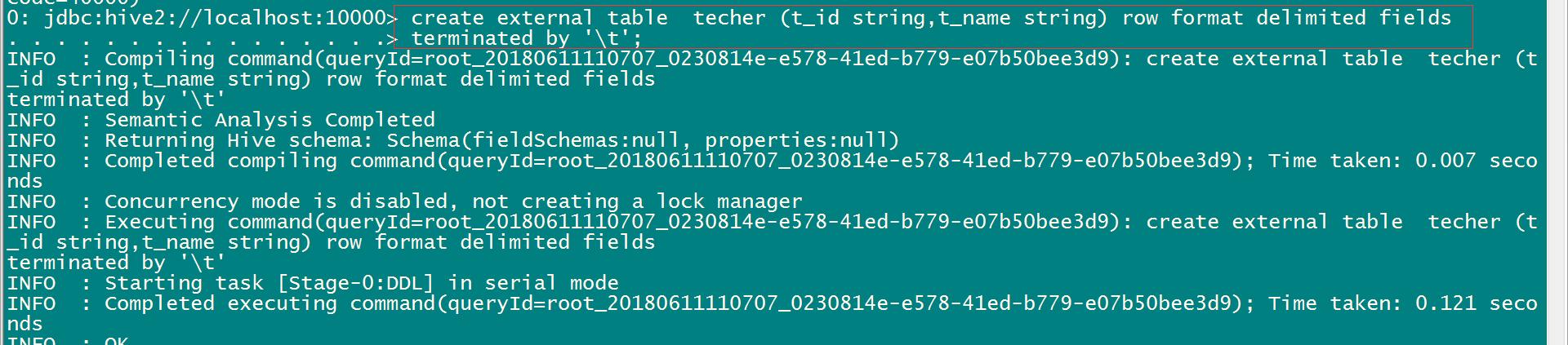
分别创建老师与学生表外部表，并向表中加载数据

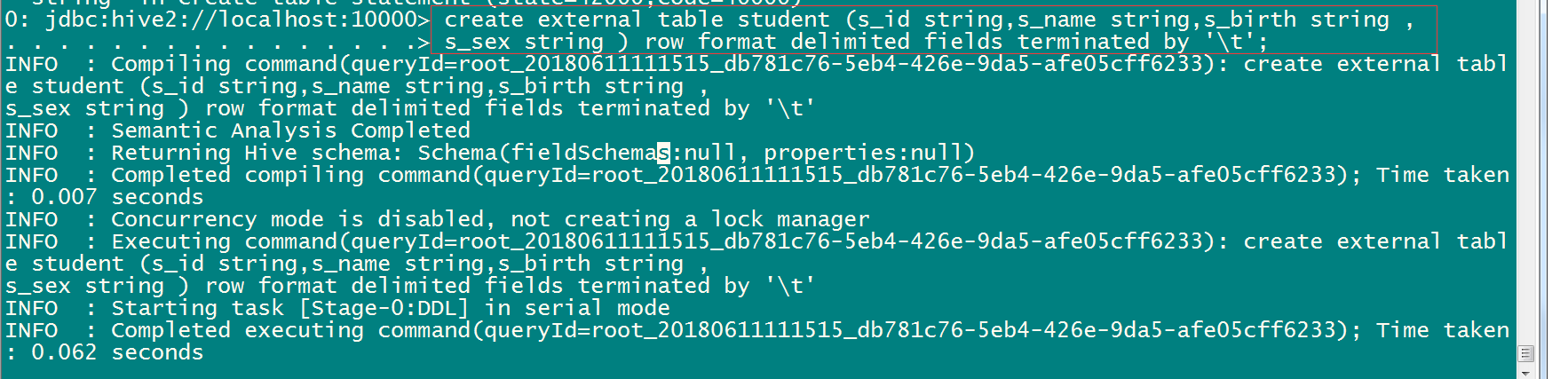
创建老师表：

create external table techer (t\_id string,t\_name string) row format delimited fields terminated by '\t';

创建学生表：

create external table student (s\_id string,s\_name string,s\_birth string , s\_sex string ) row format delimited fields terminated by '\t';





从本地文件系统向表中加载数据

load data local inpath '/export/servers/hivedatas/student.csv' into table student;

加载数据并覆盖已有数据

load data local inpath '/export/servers/hivedatas/student.csv' overwrite into table student;

从hdfs文件系统向表中加载数据（需要提前将数据上传到hdfs文件系统，其实就是一个移动文件的操作）

cd /export/servers/hivedatas

hdfs dfs -mkdir -p /hivedatas

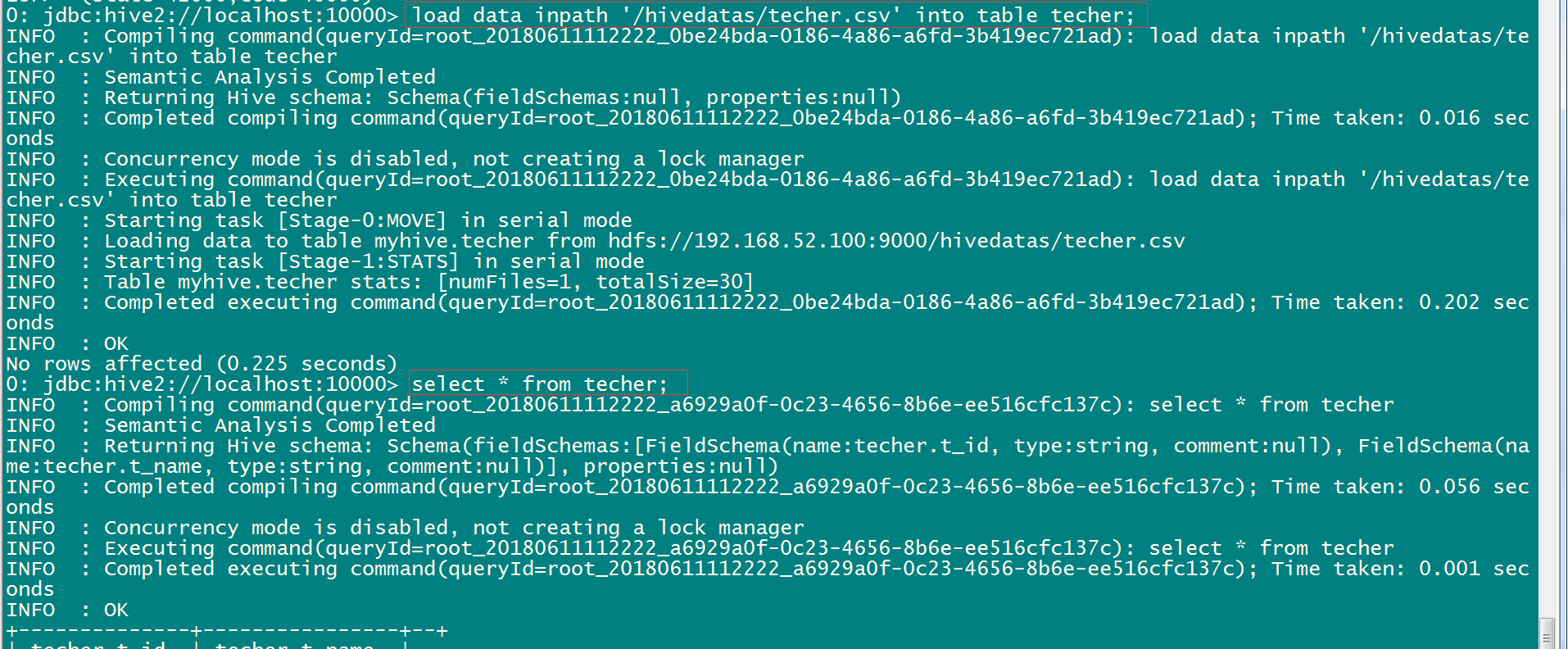
hdfs dfs -put techer.csv /hivedatas/

load data inpath '/hivedatas/techer.csv' into table techer;

加载hdfs数据并覆盖已有数据

load data inpath '/hivedatas/techer.csv' overwrite into table techer;

如果删掉student表，hdfs的数据仍然存在，并且重新创建表之后，表中就直接存在数据了,因为我们的student表使用的是外部表，drop table之后，表当中的数据依然保留在hdfs上面了



##### 分区表：

在大数据中，最常用的一种思想就是分治，我们可以把大的文件切割划分成一个个的小的文件，这样每次操作一个小的文件就会很容易了，同样的道理，在hive当中也是支持这种思想的，就是我们可以把大的数据，按照每天，或者每小时进行切分成一个个的小的文件，这样去操作小的文件就会容易得多了

创建分区表语法

create table score(s\_id string,c\_id string, s\_score int) partitioned by (month string) row format delimited fields terminated by '\t';

创建一个表带多个分区

create table score2 (s\_id string,c\_id string, s\_score int) partitioned by (year string,month string,day string) row format delimited fields terminated by '\t';

加载数据到分区表中

load data local inpath '/export/servers/hivedatas/score.csv' into table score partition (month='201806');

加载数据到一个多分区的表中去

load data local inpath '/export/servers/hivedatas/score.csv' into table score2 partition(year='2018',month='06',day='01');

多分区联合查询使用union all来实现

select \* from score where month = '201806' union all select \* from score where month = '201806';

查看分区

show partitions score;

添加一个分区

alter table score add partition(month='201805');

同时添加多个分区

alter table score add partition(month='201804') partition(month = '201803');

注意：添加分区之后就可以在hdfs文件系统当中看到表下面多了一个文件夹

删除分区

alter table score drop partition(month = '201806');

外部分区表综合练习：

需求描述：现在有一个文件score.csv文件，存放在集群的这个目录下/scoredatas/month=201806，这个文件每天都会生成，存放到对应的日期文件夹下面去，文件别人也需要公用，不能移动。需求，创建hive对应的表，并将数据加载到表中，进行数据统计分析，且删除表之后，数据不能删除

需求实现:

数据准备：

hdfs dfs -mkdir -p /scoredatas/month=201806

hdfs dfs -put score.csv /scoredatas/month=201806/

创建外部分区表，并指定文件数据存放目录

create external table score4(s\_id string, c\_id string,s\_score int) partitioned by (month string) row format delimited fields terminated by '\t' location '/scoredatas';

进行表的修复,说白了就是建立我们表与我们数据文件之间的一个关系映射

msck repair table score4;

修复成功之后即可看到数据已经全部加载到表当中去了

第二种实现方式，上传数据之后手动添加分区即可

数据准备：

hdfs dfs -mkdir -p /scoredatas/month=201805

hdfs dfs -put score.csv /scoredatas/month=201805

修改表，进行手动添加方式

alter table score4 add partition(month='201805');

##### 分桶表

将数据按照指定的字段进行分成多个桶中去，说白了就是将数据按照字段进行划分，可以将数据按照字段划分到多个文件当中去

开启hive的桶表功能

set hive.enforce.bucketing=true;

设置reduce的个数

set mapreduce.job.reduces=3;

创建通表

create table course (c\_id string,c\_name string,t\_id string) clustered by(c\_id) into 3 buckets row format delimited fields terminated by '\t';

桶表的数据加载，由于通标的数据加载通过hdfs dfs -put文件或者通过load data均不好使，只能通过insert overwrite

创建普通表，并通过insert overwrite的方式将普通表的数据通过查询的方式加载到桶表当中去

创建普通表：

create table course\_common (c\_id string,c\_name string,t\_id string) row format delimited fields terminated by '\t';

普通表中加载数据

load data local inpath '/export/servers/hivedatas/course.csv' into table course\_common;

通过insert overwrite给桶表中加载数据

insert overwrite table course select \* from course\_common cluster by(c\_id);

##### 修改表

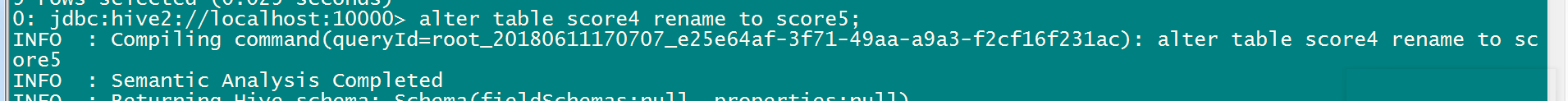
###### 表重命名

基本语法：

alter table old\_table\_name rename to new\_table\_name;

把表score4修改成score5

alter table score4 rename to score5;



###### 增加/修改列信息

（1）查询表结构

desc score5;

（2）添加列

alter table score5 add columns (mycol string, mysco string);

（3）查询表结构

desc score5;

（4）更新列(可以修改列的属性和名称)

alter table score5 change column mysco mysconew int;

##### 删除表

drop table score5;

##### hive表中加载数据

###### 直接向分区表中插入数据

create table score3 like score;

insert into table score3 partition(month ='201807') values ('001','002','100');

###### 通过查询插入数据

通过load方式加载数据

load data local inpath '/export/servers/hivedatas/score.csv' overwrite into table score partition(month='201806');

如果数据在HDFS上直接加上HDFS上的路径就可以不用加local

load data inpath'/score.csv' overwrite into table score partition(month='201806');

通过查询方式加载数据

create table score4 like score;

insert overwrite table score4 partition(month = '201806') select s\_id,c\_id,s\_score from score;

###### 多插入模式

常用于实际生产环境当中，将一张表拆开成两部分或者多部分

给score表加载数据

load data local inpath '/export/servers/hivedatas/score.csv' overwrite into table score partition(month='201806');

创建第一部分表：

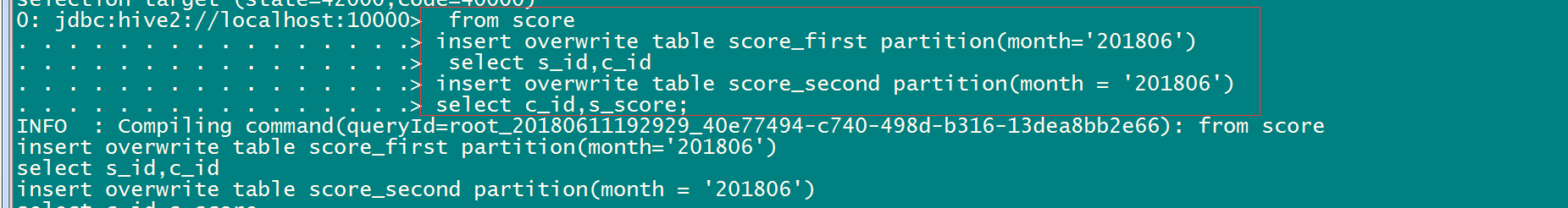
create table score\_first( s\_id string,c\_id string) partitioned by (month string) row format delimited fields terminated by '\t' ;

创建第二部分表：

create table score\_second(c\_id string,s\_score int) partitioned by (month string) row format delimited fields terminated by '\t';

分别给第一部分与第二部分表加载数据

from score insert overwrite table score\_first partition(month='201806') select s\_id,c\_id insert overwrite table score\_second partition(month = '201806') select c\_id,s\_score;



###### 查询语句中创建表并加载数据（as select）

将查询的结果保存到一张表当中去

create table score5 as select \* from score;

###### 创建表时通过location指定加载数据路径

1. 创建表，并指定在hdfs上的位置

create external table score6 (s\_id string,c\_id string,s\_score int) row format delimited fields terminated by '\t' location '/myscore6';

2）上传数据到hdfs上

hdfs dfs -mkdir -p /myscore6

hdfs dfs -put score.csv /myscore6;

3）查询数据

select \* from score6;

###### export导出与import 导入 hive表数据（内部表操作）

create table techer2 like techer;

export table techer to '/export/techer';

import table techer2 from '/export/techer';

##### hive表中的数据导出

将hive表中的数据导出到其他任意目录，例如linux本地磁盘，例如hdfs，例如mysql等等

###### insert导出

1. 将查询的结果导出到本地

insert overwrite local directory '/export/servers/exporthive' select \* from score;

1. 将查询的结果格式化导出到本地

insert overwrite local directory '/export/servers/exporthive' row format delimited fields terminated by '\t' collection items terminated by '#' select \* from student;

1. 将查询的结果导出到HDFS上(没有local)

insert overwrite directory '/export/servers/exporthive' row format delimited fields terminated by '\t' collection items terminated by '#' select \* from score;

###### Hadoop命令导出到本地

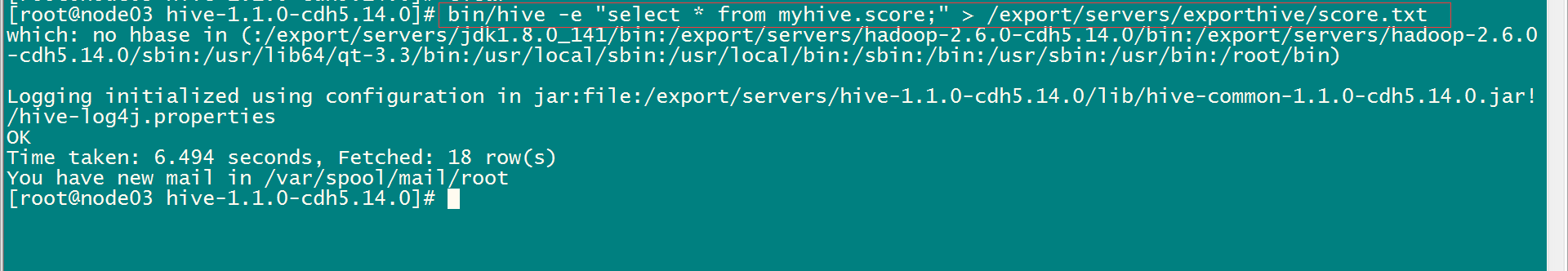
dfs -get /export/servers/exporthive/000000\_0 /export/servers/exporthive/local.txt;



###### hive shell 命令导出

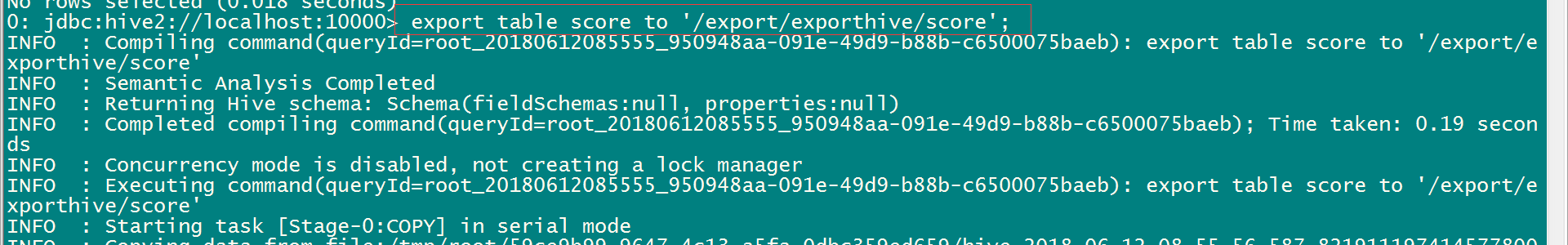
基本语法：（hive -f/-e 执行语句或者脚本 > file）

bin/hive -e "select \* from myhive.score;" > /export/servers/exporthive/score.txt



###### export导出到HDFS上

export table score to '/export/exporthive/score';



###### sqoop导出

后续单独讲。

##### 清空表数据

只能清空管理表，也就是内部表

truncate table score6;

### 3.2、 DML操作

#### 3.2.1、SELECT

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+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。*

*3、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。*

*4、Cluster by(字段) 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，此时，cluster by = distribute by + sort by*

分桶表的作用：最大的作用是用来提高join操作的效率；

（思考这个问题：

select a.id,a.name,b.addr from a join b on a.id = b.id;

如果a表和b表已经是分桶表，而且分桶的字段是id字段

做这个join操作时，还需要全表做笛卡尔积吗？）

##### 全表查询

select \* from score;

##### 选择特定列查询

select s\_id ,c\_id from score;

##### 列别名

1）重命名一个列。

2）便于计算。

3）紧跟列名，也可以在列名和别名之间加入关键字‘AS’

select s\_id as myid ,c\_id from score;

#### 3.2.2、常用函数

1）求总行数（count）

select count(1) from score;

2）求分数的最大值（max）

select max(s\_score) from score;

3）求分数的最小值（min）

select min(s\_score) from score;

4）求分数的总和（sum）

select sum(s\_score) from score;

5）求分数的平均值（avg）

select avg(s\_score) from score;

#### 3.2.3、LIMIT语句

典型的查询会返回多行数据。LIMIT子句用于限制返回的行数。

select \* from score limit 3;

#### 3.2.4、WHERE语句

1）使用WHERE 子句，将不满足条件的行过滤掉。

2）WHERE 子句紧随 FROM 子句。

3）案例实操

查询出分数大于60的数据

select \* from score where s\_score > 60;

#### 3.2.5、比较运算符（BETWEEN/IN/ IS NULL）

1）下面表中描述了谓词操作符，这些操作符同样可以用于JOIN…ON和HAVING语句中。

|  |  |  |
| --- | --- | --- |
| 操作符 | 支持的数据类型 | 描述 |
| A=B | 基本数据类型 | 如果A等于B则返回TRUE，反之返回FALSE |
| A<=>B | 基本数据类型 | 如果A和B都为NULL，则返回TRUE，其他的和等号（=）操作符的结果一致，如果任一为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任一为NULL，则结果为NULL。如果A的值大于等于B而且小于或等于C，则结果为TRUE，反之为FALSE。如果使用NOT关键字则可达到相反的效果。 |
| A IS NULL | 所有数据类型 | 如果A等于NULL，则返回TRUE，反之返回FALSE |
| A IS NOT NULL | 所有数据类型 | 如果A不等于NULL，则返回TRUE，反之返回FALSE |
| IN(数值1, 数值2) | 所有数据类型 | 使用 IN运算显示列表中的值 |
| A [NOT] LIKE B | STRING 类型 | B是一个SQL下的简单正则表达式，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。 |

2）案例实操

（1）查询分数等于80的所有的数据

select \* from score where s\_score = 80;

（2）查询分数在80到100的所有数据

select \* from score where s\_score between 80 and 100;

（3）查询成绩为空的所有数据

select \* from score where s\_score is null;

（4）查询成绩是80和90的数据

select \* from score where s\_score in(80,90);

#### 3.2.6、LIKE和RLIKE

1）使用LIKE运算选择类似的值

2）选择条件可以包含字符或数字:

% 代表零个或多个字符(任意个字符)。

\_ 代表一个字符。

3）RLIKE子句是Hive中这个功能的一个扩展，其可以通过Java的正则表达式这个更强大的语言来指定匹配条件。

4）案例实操

（1）查找以8开头的所有成绩

select \* from score where s\_score like '8%';

（2）查找第二个数值为9的所有成绩数据

select \* from score where s\_score like '\_9%';

（3）查找成绩中含9的所有成绩数据

select \* from score where s\_score rlike '[9]';

###### 逻辑运算符（AND/OR/NOT）

|  |  |
| --- | --- |
| 操作符 | 含义 |
| AND | 逻辑并 |
| OR | 逻辑或 |
| NOT | 逻辑否 |

案例实操

（1）查询成绩大于80，并且s\_id是01的数据

select \* from score where s\_score >80 and s\_id = '01';

（2）查询成绩大于80，或者s\_id 是01的数

select \* from score where s\_score > 80 or s\_id = '01';

（3）查询s\_id 不是 01和02的学生

select \* from score where s\_id not in ('01','02');

#### 3.2.7、分组

##### GROUP BY语句

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

案例实操：

（1）计算每个学生的平均分数

select s\_id ,avg(s\_score) from score group by s\_id;

（2）计算每个学生最高成绩

select s\_id ,max(s\_score) from score group by s\_id;

##### HAVING语句

1）having与where不同点

（1）where针对表中的列发挥作用，查询数据；having针对查询结果中的列发挥作用，筛选数据。

（2）where后面不能写分组函数，而having后面可以使用分组函数。

（3）having只用于group by分组统计语句。

2）案例实操：

求每个学生的平均分数

select s\_id ,avg(s\_score) from score group by s\_id;

求每个学生平均分数大于85的人

select s\_id ,avg(s\_score) avgscore from score group by s\_id having avgscore > 85;

#### 3.2.8、JOIN语句

##### 3.2.8.1、等值JOIN

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

案例操作

1. 查询分数对应的姓名

SELECT s.s\_id,s.s\_score,stu.s\_name,stu.s\_birth FROM score s LEFT JOIN student stu ON s.s\_id = stu.s\_id

##### 3.2.8.2、表的别名

1）好处

（1）使用别名可以简化查询。

（2）使用表名前缀可以提高执行效率。

2）案例实操

合并老师与课程表

select \* from techer t join course c on t.t\_id = c.t\_id;

##### 3.2.8.3、内连接（INNER JOIN）

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。

select \* from techer t inner join course c on t.t\_id = c.t\_id;

##### 3.2.8.4、左外连接（LEFT OUTER JOIN）

左外连接：JOIN操作符左边表中符合WHERE子句的所有记录将会被返回。

查询老师对应的课程

select \* from techer t left join course c on t.t\_id = c.t\_id;

##### 3.2.8.5、右外连接（RIGHT OUTER JOIN）

右外连接：JOIN操作符右边表中符合WHERE子句的所有记录将会被返回。

select \* from techer t right join course c on t.t\_id = c.t\_id;

##### 3.2.8.6、满外连接（FULL OUTER JOIN）

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

SELECT \* FROM techer t FULL JOIN course c ON t.t\_id = c.t\_id ;

##### 3.2.8.7、多表连接

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

多表连接查询，查询老师对应的课程，以及对应的分数，对应的学生

select \* from techer t

left join course c

on t.t\_id = c.t\_id

left join score s

on s.c\_id = c.c\_id

left join student stu

on s.s\_id = stu.s\_id;

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

#### 3.2.9、 排序

##### 3.2.9.1 全局排序（Order By）

Order By：全局排序，一个MapReduce

1）使用 ORDER BY 子句排序

ASC（ascend）: 升序（默认）

DESC（descend）: 降序

2）ORDER BY 子句在SELECT语句的结尾。

3）案例实操

（1）查询学生的成绩，并按照分数降序排列

SELECT \* FROM student s LEFT JOIN score sco ON s.s\_id = sco.s\_id ORDER BY sco.s\_score DESC;

（2）查询学生的成绩，并按照分数升序排列

SELECT \* FROM student s LEFT JOIN score sco ON s.s\_id = sco.s\_id ORDER BY sco.s\_score asc;

##### 3.2.9.2 按照别名排序

按照分数的平均值排序

select s\_id ,avg(s\_score) avg from score group by s\_id order by avg;

##### 3.2.9.3 多个列排序

按照学生id和成绩进行排序

select s\_id ,avg(s\_score) avg from score group by s\_id order by s\_id,avg;

##### 3.2.9.4 每个MapReduce内部排序（Sort By）

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;

3）查询成绩按照成绩降序排列

select \* from score sort by s\_score;

1. 将查询结果导入到文件中（按照成绩降序排列）

insert overwrite local directory '/export/servers/hivedatas/sort' select \* from score sort by s\_score;

##### 3.2.9.5 分区排序（DISTRIBUTE BY）

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

注意，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。

案例实操：

1. 先按照学生id进行分区，再按照学生成绩进行排序。

设置reduce的个数，将我们对应的s\_id划分到对应的reduce当中去

set mapreduce.job.reduces=7;

通过distribute by 进行数据的分区

insert overwrite local directory '/export/servers/hivedatas/sort' select \* from score distribute by s\_id sort by s\_score;

##### 3.2.9.6 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。但是排序只能是倒序排序，不能指定排序规则为ASC或者DESC。

1. 以下两种写法等价

select \* from score cluster by s\_id;

select \* from score distribute by s\_id sort by s\_id;

## 四、Hive Shell参数

### 4.1、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. 设置hive运行时候的参数配置

### 4.2、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。

参数声明 > 命令行参数 > 配置文件参数（hive）

## 五、Hive函数

### 5.1、内置函数

*内容较多，见《Hive官方文档》*

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

1）查看系统自带的函数

hive> show functions;

2）显示自带的函数的用法

hive> desc function upper;

3）详细显示自带的函数的用法

hive> desc function extended upper;

### 5.2 Hive自定义函数

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

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

3）根据用户自定义函数类别分为以下三种：

（1）UDF（User-Defined-Function）

一进一出

（2）UDAF（User-Defined Aggregation Function）

聚集函数，多进一出

类似于：count/max/min

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

一进多出

如lateral view explore()

4）官方文档地址

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

5）编程步骤：

（1）继承org.apache.hadoop.hive.ql.UDF

（2）需要实现evaluate函数；evaluate函数支持重载；

6）注意事项

（1）UDF必须要有返回类型，可以返回null，但是返回类型不能为void；

（2）UDF中常用Text/LongWritable等类型，不推荐使用java类型；

### 5.3、UDF开发实例

#### 简单UDF示例

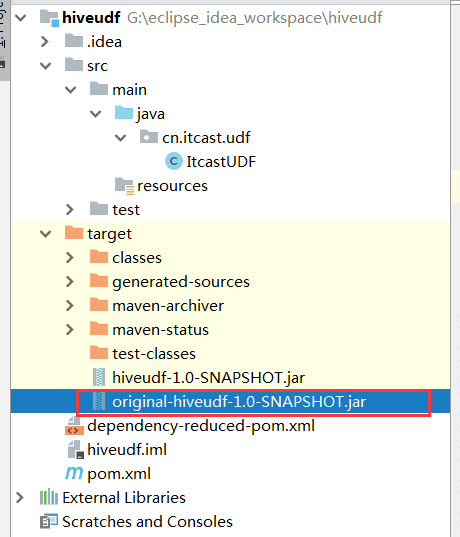
##### 第一步：创建maven java 工程，导入jar包

<repositories>  
 <repository>  
 <id>cloudera</id>  
 <url>https://repository.cloudera.com/artifactory/cloudera-repos/</url>  
 </repository>  
</repositories>  
<dependencies>  
 <dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-common</artifactId>  
 <version>2.6.0-cdh5.14.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.hive</groupId>  
 <artifactId>hive-exec</artifactId>  
 <version>1.1.0-cdh5.14.0</version>  
 </dependency>  
</dependencies>  
<build>  
<plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.0</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 <encoding>UTF-8</encoding>  
 </configuration>  
 </plugin>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-shade-plugin</artifactId>  
 <version>2.2</version>  
 <executions>  
 <execution>  
 <phase>package</phase>  
 <goals>  
 <goal>shade</goal>  
 </goals>  
 <configuration>  
 <filters>  
 <filter>  
 <artifact>\*:\*</artifact>  
 <excludes>  
 <exclude>META-INF/\*.SF</exclude>  
 <exclude>META-INF/\*.DSA</exclude>  
 <exclude>META-INF/\*/RSA</exclude>  
 </excludes>  
 </filter>  
 </filters>  
 </configuration>  
 </execution>  
 </executions>  
 </plugin>  
</plugins>  
</build>

##### 第二步：开发java类继承UDF，并重载evaluate 方法

**public class** ItcastUDF **extends** UDF {  
 **public** Text evaluate(**final** Text s) {  
 **if** (**null** == s) {  
 **return null**;  
 }  
 *//返回大写字母* **return new** Text(s.toString().toUpperCase());  
  
 }  
}

##### 第三步：将我们的项目打包，并上传到hive的lib目录下



##### 第四步：添加我们的jar包

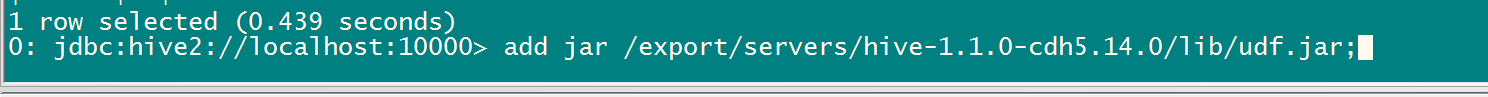
重命名我们的jar包名称

cd /export/servers/hive-1.1.0-cdh5.14.0/lib

mv original-day\_06\_hive\_udf-1.0-SNAPSHOT.jar udf.jar

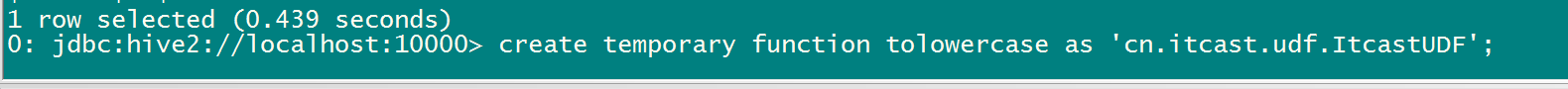
hive的客户端添加我们的jar包

add jar /export/servers/hive-1.1.0-cdh5.14.0/lib/udf.jar;



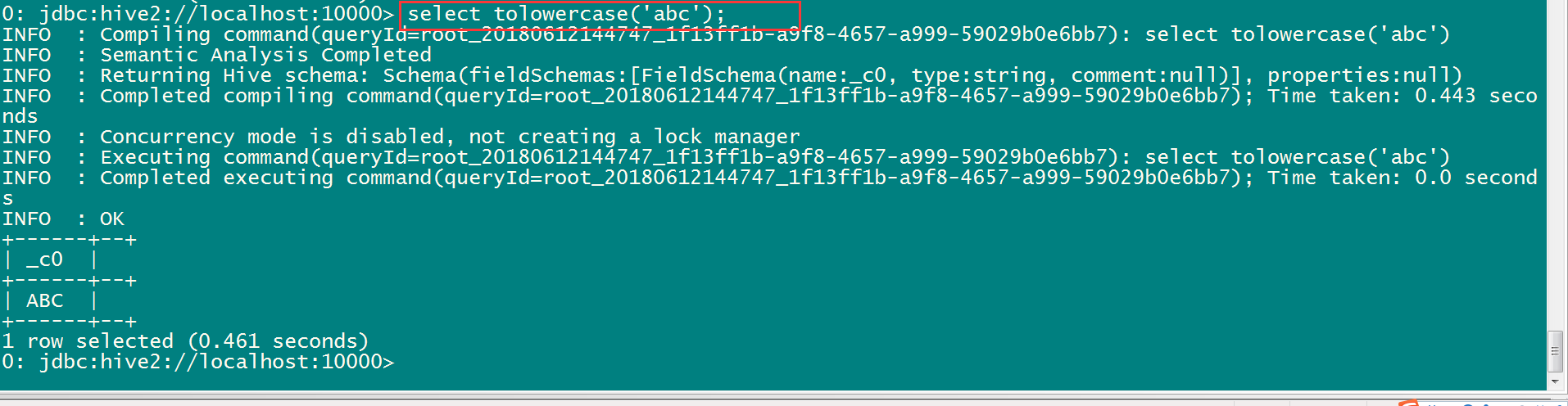
##### 第五步：设置函数与我们的自定义函数关联

create temporary function tolowercase as 'cn.itcast.udf.ItcastUDF';



##### 第六步：使用自定义函数

select tolowercase('abc');



#### Json数据解析UDF开发（作业）

作业：

有原始json数据如下：

|  |
| --- |
| {"movie":"1193","rate":"5","timeStamp":"978300760","uid":"1"}  {"movie":"661","rate":"3","timeStamp":"978302109","uid":"1"}  {"movie":"914","rate":"3","timeStamp":"978301968","uid":"1"}  {"movie":"3408","rate":"4","timeStamp":"978300275","uid":"1"}  {"movie":"2355","rate":"5","timeStamp":"978824291","uid":"1"}  {"movie":"1197","rate":"3","timeStamp":"978302268","uid":"1"}  {"movie":"1287","rate":"5","timeStamp":"978302039","uid":"1"} |

需要将数据导入到hive数据仓库中

我不管你中间用几个表，最终我要得到一个结果表：

|  |  |  |  |
| --- | --- | --- | --- |
| movie | rate | timestamp | uid |
| 1197 | 3 | 978302268 | 1 |

注：全在hive中完成，可以用自定义函数

第一步：自定义udf函数，将我们json数据给解析出来，解析成四个字段，整成一个\t分割的一行

第二步：注册我们的自定义函数

第三步：创建一个临时表，加载json格式的数据，加载到临时表里面的一个字段里面去

第四步：insert overwrite select 将临时表当中的数据通过我们的自定义函数，给查询出来，加载到新的表 里面去

## 六、hive的数据压缩

在实际工作当中，hive当中处理的数据，一般都需要经过压缩，前期我们在学习hadoop的时候，已经配置过hadoop的压缩，我们这里的hive也是一样的可以使用压缩来节省我们的MR处理的网络带宽

### 6.1、MR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，如下表所示

|  |  |
| --- | --- |
| 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 |

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

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

### 6.2、压缩配置参数

要在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 |

注意：将我们mapred-site.xml当中配置的压缩参数给去掉（之前在hadoop的课程当中配置过mapred-site.xml配置文件）

### 6.3、开启Map输出阶段压缩

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。具体配置如下：

**案例实操：**

1）开启hive中间传输数据压缩功能

hive (default)>set hive.exec.compress.intermediate=true;

2）开启mapreduce中map输出压缩功能

hive (default)>set mapreduce.map.output.compress=true;

3）设置mapreduce中map输出数据的压缩方式

hive (default)>set mapreduce.map.output.compress.codec= org.apache.hadoop.io.compress.SnappyCodec;

4）执行查询语句

select count(1) from score;

### 6.4 开启Reduce输出阶段压缩

当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。

**案例实操：**

1）开启hive最终输出数据压缩功能

hive (default)>set hive.exec.compress.output=true;

2）开启mapreduce最终输出数据压缩

hive (default)>set mapreduce.output.fileoutputformat.compress=true;

3）设置mapreduce最终数据输出压缩方式

hive (default)> set mapreduce.output.fileoutputformat.compress.codec = org.apache.hadoop.io.compress.SnappyCodec;

4）设置mapreduce最终数据输出压缩为块压缩

hive (default)>set mapreduce.output.fileoutputformat.compress.type=BLOCK;

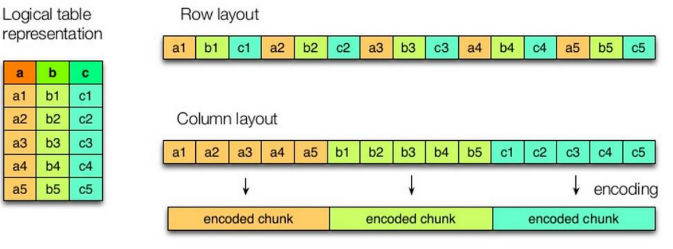
5）测试一下输出结果是否是压缩文件

insert overwrite local directory '/export/servers/snappy' select \* from score distribute by s\_id sort by s\_id desc;

## 七、hive的数据存储格式

Hive支持的存储数的格式主要有：TEXTFILE（行式存储） 、SEQUENCEFILE(行式存储)、ORC（列式存储）、PARQUET（列式存储）。

### 7.1 列式存储和行式存储



上图左边为逻辑表，右边第一个为行式存储，第二个为列式存储。

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

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

TEXTFILE和SEQUENCEFILE的存储格式都是基于行存储的；

ORC和PARQUET是基于列式存储的。

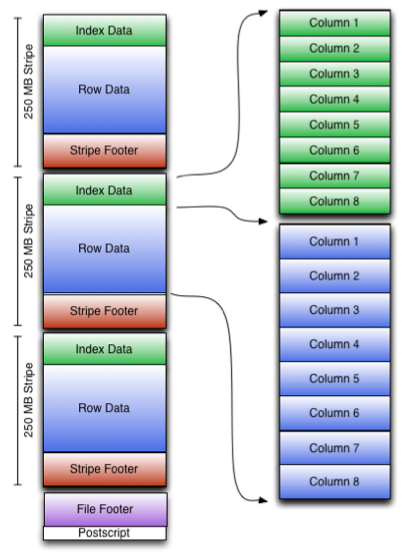
### 7.2 TEXTFILE格式

默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合Gzip、Bzip2使用(系统自动检查，执行查询时自动解压)，但使用这种方式，hive不会对数据进行切分，从而无法对数据进行并行操作。

### 7.3 ORC格式

Orc (Optimized Row Columnar)是hive 0.11版里引入的新的存储格式。

可以看到每个Orc文件由1个或多个stripe组成，每个stripe250MB大小，这个Stripe实际相当于RowGroup概念，不过大小由4MB->250MB，这样能提升顺序读的吞吐率。每个Stripe里有三部分组成，分别是Index Data,Row Data,Stripe Footer：



   1）Index Data：一个轻量级的index，默认是每隔1W行做一个索引。这里做的索引只是记录某行的各字段在Row Data中的offset。

    2）Row Data：存的是具体的数据，先取部分行，然后对这些行按列进行存储。对每个列进行了编码，分成多个Stream来存储。

    3）Stripe Footer：存的是各个Stream的类型，长度等信息。

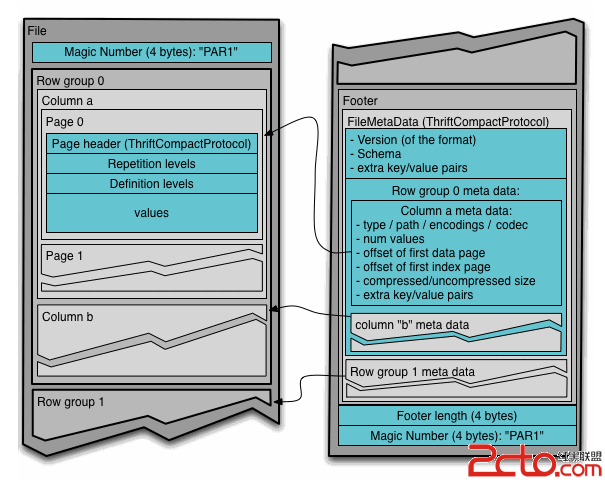
每个文件有一个File Footer，这里面存的是每个Stripe的行数，每个Column的数据类型信息等；每个文件的尾部是一个PostScript，这里面记录了整个文件的压缩类型以及FileFooter的长度信息等。在读取文件时，会seek到文件尾部读PostScript，从里面解析到File Footer长度，再读FileFooter，从里面解析到各个Stripe信息，再读各个Stripe，即从后往前读。

### 7.4 PARQUET格式

Parquet是面向分析型业务的列式存储格式，由Twitter和Cloudera合作开发，2015年5月从Apache的孵化器里毕业成为Apache顶级项目。

Parquet文件是以二进制方式存储的，所以是不可以直接读取的，文件中包括该文件的数据和元数据，因此Parquet格式文件是自解析的。

通常情况下，在存储Parquet数据的时候会按照Block大小设置行组的大小，由于一般情况下每一个Mapper任务处理数据的最小单位是一个Block，这样可以把每一个行组由一个Mapper任务处理，增大任务执行并行度。Parquet文件的格式如下图所示。



上图展示了一个Parquet文件的内容，一个文件中可以存储多个行组，文件的首位都是该文件的Magic Code，用于校验它是否是一个Parquet文件，Footer length记录了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量，文件的元数据中包括每一个行组的元数据信息和该文件存储数据的Schema信息。除了文件中每一个行组的元数据，每一页的开始都会存储该页的元数据，在Parquet中，有三种类型的页：数据页、字典页和索引页。数据页用于存储当前行组中该列的值，字典页存储该列值的编码字典，每一个列块中最多包含一个字典页，索引页用来存储当前行组下该列的索引，目前Parquet中还不支持索引页。

### 7.5 主流文件存储格式对比实验

从存储文件的压缩比和查询速度两个角度对比。

**存储文件的压缩比测试：**

0）测试数据 参见log.data

1）TextFile

（1）创建表，存储数据格式为TEXTFILE

|  |
| --- |
| create table log\_text (  track\_time string,  url string,  session\_id string,  referer string,  ip string,  end\_user\_id string,  city\_id string  )  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  STORED AS TEXTFILE ; |

（2）向表中加载数据

|  |
| --- |
| load data local inpath '/export/servers/hivedatas/log.data' into table log\_text ; |

（3）查看表中数据大小

|  |
| --- |
| dfs -du -h /user/hive/warehouse/myhive.db/log\_text; |

18.1 M /user/hive/warehouse/log\_text/log.data

2）ORC

（1）创建表，存储数据格式为ORC

|  |
| --- |
| create table log\_orc(  track\_time string,  url string,  session\_id string,  referer string,  ip string,  end\_user\_id string,  city\_id string  )  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  STORED AS orc ; |

（2）向表中加载数据

|  |
| --- |
| insert into table log\_orc select \* from log\_text ; |

（3）查看表中数据大小

|  |
| --- |
| dfs -du -h /user/hive/warehouse/myhive.db/log\_orc; |

2.8 M /user/hive/warehouse/log\_orc/123456\_0

3）Parquet

（1）创建表，存储数据格式为parquet

|  |
| --- |
| create table log\_parquet(  track\_time string,  url string,  session\_id string,  referer string,  ip string,  end\_user\_id string,  city\_id string  )  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  STORED AS PARQUET ; |

（2）向表中加载数据

|  |
| --- |
| insert into table log\_parquet select \* from log\_text ; |

（3）查看表中数据大小

|  |
| --- |
| dfs -du -h /user/hive/warehouse/myhive.db/log\_parquet; |

13.1 M /user/hive/warehouse/log\_parquet/123456\_0

存储文件的压缩比总结：

ORC > Parquet > textFile

**存储文件的查询速度测试：**

1）TextFile

hive (default)> select count(\*) from log\_text;

\_c0

100000

Time taken: 21.54 seconds, Fetched: 1 row(s)

2）ORC

hive (default)> select count(\*) from log\_orc;

\_c0

100000

Time taken: 20.867 seconds, Fetched: 1 row(s)

3）Parquet

hive (default)> select count(\*) from log\_parquet;

\_c0

100000

Time taken: 22.922 seconds, Fetched: 1 row(s)

存储文件的查询速度总结：

ORC > TextFile > Parquet

## 八、存储和压缩结合

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

ORC存储方式的压缩：

|  |  |  |
| --- | --- | --- |
| Key | Default | Notes |
| orc.compress | ZLIB | high level compression (one of NONE, ZLIB, SNAPPY) |
| orc.compress.size | 262,144 | number of bytes in each compression chunk |
| orc.stripe.size | 67,108,864 | number of bytes in each stripe |
| orc.row.index.stride | 10,000 | number of rows between index entries (must be >= 1000) |
| orc.create.index | true | whether to create row indexes |
| orc.bloom.filter.columns | "" | comma separated list of column names for which bloom filter should be created |
| orc.bloom.filter.fpp | 0.05 | false positive probability for bloom filter (must >0.0 and <1.0) |

1）创建一个非压缩的的ORC存储方式

（1）建表语句

|  |
| --- |
| create table log\_orc\_none(  track\_time string,  url string,  session\_id string,  referer string,  ip string,  end\_user\_id string,  city\_id string  )  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  STORED AS orc tblproperties ("orc.compress"="NONE"); |

（2）插入数据

|  |
| --- |
| insert into table log\_orc\_none select \* from log\_text ; |

（3）查看插入后数据

|  |
| --- |
| dfs -du -h /user/hive/warehouse/myhive.db/log\_orc\_none; |

7.7 M /user/hive/warehouse/log\_orc\_none/123456\_0

2）创建一个SNAPPY压缩的ORC存储方式

（1）建表语句

|  |
| --- |
| create table log\_orc\_snappy(  track\_time string,  url string,  session\_id string,  referer string,  ip string,  end\_user\_id string,  city\_id string  )  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  STORED AS orc tblproperties ("orc.compress"="SNAPPY"); |

（2）插入数据

|  |
| --- |
| insert into table log\_orc\_snappy select \* from log\_text ; |

（3）查看插入后数据

|  |
| --- |
| dfs -du -h /user/hive/warehouse/myhive.db/log\_orc\_snappy ; |

3.8 M /user/hive/warehouse/log\_orc\_snappy/123456\_0

3）上一节中默认创建的ORC存储方式，导入数据后的大小为

2.8 M /user/hive/warehouse/log\_orc/123456\_0

比Snappy压缩的还小。原因是orc存储文件默认采用ZLIB压缩。比snappy压缩的小。

**4）存储方式和压缩总结：**

在实际的项目开发当中，hive表的数据存储格式一般选择：orc或parquet。压缩方式一般选择snappy。

## 九、调优

### 9.1 Fetch抓取（Hive可以避免进行MapReduce）

Hive中对某些情况的查询可以不必使用MapReduce计算。例如：SELECT \* FROM employees;在这种情况下，Hive可以简单地读取employee对应的存储目录下的文件，然后输出查询结果到控制台。

在hive-default.xml.template文件中hive.fetch.task.conversion默认是more，老版本hive默认是minimal，该属性修改为more以后，在全局查找、字段查找、limit查找等都不走mapreduce。

|  |
| --- |
| <property>  <name>hive.fetch.task.conversion</name>  <value>more</value>  <description>  Expects one of [none, minimal, more].  Some select queries can be converted to single FETCH task minimizing latency.  Currently the query should be single sourced not having any subquery and should not have  any aggregations or distincts (which incurs RS), lateral views and joins.  0. none : disable hive.fetch.task.conversion  1. minimal : SELECT STAR, FILTER on partition columns, LIMIT only 所有的都要走MR  2. more : SELECT, FILTER, LIMIT only (support TABLESAMPLE and virtual columns)  </description>  </property> |

案例实操：

1）把hive.fetch.task.conversion设置成none，然后执行查询语句，都会执行mapreduce程序。

hive (default)> set hive.fetch.task.conversion=none;

hive (default)> select \* from score;

hive (default)> select s\_score from score;

hive (default)> select s\_score from score limit 3;

2）把hive.fetch.task.conversion设置成more，然后执行查询语句，如下查询方式都不会执行mapreduce程序。

hive (default)> set hive.fetch.task.conversion=more;

hive (default)> select \* from score;

hive (default)> select s\_score from score;

hive (default)> select s\_score from score limit 3;

9.2 本地模式

大多数的Hadoop Job是需要Hadoop提供的完整的可扩展性来处理大数据集的。不过，有时Hive的输入数据量是非常小的。在这种情况下，为查询触发执行任务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。

|  |
| --- |
| set hive.exec.mode.local.auto=true;  //开启本地mr  //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; |

案例实操：

1）开启本地模式，并执行查询语句

hive (default)> set hive.exec.mode.local.auto=true;

hive (default)> select \* from score cluster by s\_id;

18 rows selected (1.568 seconds)

2）关闭本地模式，并执行查询语句

hive (default)> set hive.exec.mode.local.auto=false;

hive (default)> select \* from score cluster by s\_id;

18 rows selected (11.865 seconds)

### 9.2 表的优化

#### 9.2.1 Join

**Join原则：**

1）小表Join大表，

将key相对分散，并且数据量小的表放在join的左边，这样可以有效减少内存溢出错误发生的几率；再进一步，可以使用Group让小的维度表（1000条以下的记录条数）先进内存。在map端完成reduce。

select count(distinct s\_id) from score;

select count(s\_id) from score group by s\_id; 在map端进行聚合，效率更高

2）多个表关联时，最好分拆成小段，避免大sql（无法控制中间Job）

3）大表Join大表

（1）空KEY过滤

有时join超时是因为某些key对应的数据太多，而相同key对应的数据都会发送到相同的reducer上，从而导致内存不够。此时我们应该仔细分析这些异常的key，很多情况下，这些key对应的数据是异常数据，我们需要在SQL语句中进行过滤。例如key对应的字段为空，操作如下：

环境准备：

|  |
| --- |
| create table ori(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  create table nullidtable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  create table jointable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  load data local inpath '/export/servers/hivedatas/hive\_big\_table/\*' into table ori;  load data local inpath '/export/servers/hivedatas/hive\_have\_null\_id/\*' into table nullidtable; |

不过滤：

|  |
| --- |
| INSERT OVERWRITE TABLE jointable  SELECT a.\* FROM nullidtable a JOIN ori b ON a.id = b.id;  结果：  No rows affected (152.135 seconds) |

过滤：

|  |
| --- |
| INSERT OVERWRITE TABLE jointable  SELECT a.\* FROM (SELECT \* FROM nullidtable WHERE id IS NOT NULL ) a JOIN ori b ON a.id = b.id;  结果：  No rows affected (141.585 seconds) |

 （2）空key转换

有时虽然某个key为空对应的数据很多，但是相应的数据不是异常数据，必须要包含在join的结果中，此时我们可以表a中key为空的字段赋一个随机的值，使得数据随机均匀地分不到不同的reducer上。例如：

不随机分布：

|  |
| --- |
| set hive.exec.reducers.bytes.per.reducer=32123456;  set mapreduce.job.reduces=7;  INSERT OVERWRITE TABLE jointable  SELECT a.\*  FROM nullidtable a  LEFT JOIN ori b ON CASE WHEN a.id IS NULL THEN 'hive' ELSE a.id END = b.id;  No rows affected (41.668 seconds) 52.477 |

**结果：这样的后果就是所有为null值的id全部都变成了相同的字符串，及其容易造成数据的倾斜（所有的key相同，相同key的数据会到同一个reduce当中去）**

**为了解决这种情况，我们可以通过hive的rand函数，随记的给每一个为空的id赋上一个随机值，这样就不会造成数据倾斜**

随机分布：

|  |
| --- |
| set hive.exec.reducers.bytes.per.reducer=32123456;  set mapreduce.job.reduces=7;  INSERT OVERWRITE TABLE jointable  SELECT a.\*  FROM nullidtable a  LEFT JOIN ori b ON CASE WHEN a.id IS NULL THEN concat('hive', rand()) ELSE a.id END = b.id;  No rows affected (42.594 seconds) |

**4）案例实操**

（0）需求：测试大表JOIN小表和小表JOIN大表的效率 （新的版本当中已经没有区别了，旧的版本当中需要使用小表）

（1）建大表、小表和JOIN后表的语句

|  |
| --- |
| create table bigtable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  create table smalltable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  create table jointable2(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t'; |

（2）分别向大表和小表中导入数据

hive (default)> load data local inpath '/export/servers/hivedatas/big\_data' into table bigtable;

hive (default)>load data local inpath '/export/servers/hivedatas/small\_data' into table smalltable;

（3）关闭mapjoin功能（默认是打开的）

set hive.auto.convert.join = false;

（4）执行小表JOIN大表语句

|  |
| --- |
| INSERT OVERWRITE TABLE jointable2  SELECT b.id, b.time, b.uid, b.keyword, b.url\_rank, b.click\_num, b.click\_url  FROM smalltable s  left JOIN bigtable b  ON b.id = s.id; |

Time taken: 67.411 seconds

（5）执行大表JOIN小表语句

|  |
| --- |
| INSERT OVERWRITE TABLE jointable2  SELECT b.id, b.time, b.uid, b.keyword, b.url\_rank, b.click\_num, b.click\_url  FROM bigtable b  left JOIN smalltable s  ON s.id = b.id; |

Time taken: 69.376seconds

可以看出大表join小表或者小表join大表，就算是关闭map端join的情况下，在新的版本当中基本上没有区别了（hive为了解决数据倾斜的问题，会自动进行过滤）

#### 9.2.2 MapJoin

如果不指定MapJoin或者不符合MapJoin的条件，那么Hive解析器会将Join操作转换成Common Join，即：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。

1）开启MapJoin参数设置：

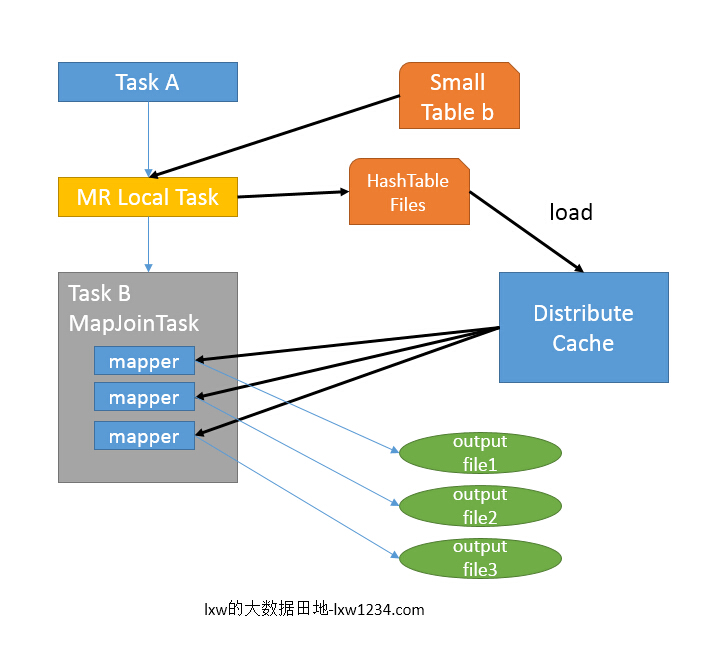
（1）设置自动选择Mapjoin

set hive.auto.convert.join = true; 默认为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;

2）MapJoin工作机制



首先是Task A，它是一个Local Task（在客户端本地执行的Task），负责扫描小表b的数据，将其转换成一个HashTable的数据结构，并写入本地的文件中，之后将该文件加载到DistributeCache中。

接下来是Task B，该任务是一个没有Reduce的MR，启动MapTasks扫描大表a,在Map阶段，根据a的每一条记录去和DistributeCache中b表对应的HashTable关联，并直接输出结果。

由于MapJoin没有Reduce，所以由Map直接输出结果文件，有多少个Map Task，就有多少个结果文件。

**案例实操：**

（1）开启Mapjoin功能

set hive.auto.convert.join = true; 默认为true

（2）执行小表JOIN大表语句

|  |
| --- |
| INSERT OVERWRITE TABLE jointable2  SELECT b.id, b.time, b.uid, b.keyword, b.url\_rank, b.click\_num, b.click\_url  FROM smalltable s  JOIN bigtable b  ON s.id = b.id; |

Time taken: 31.814 seconds

（3）执行大表JOIN小表语句

|  |
| --- |
| INSERT OVERWRITE TABLE jointable2  SELECT b.id, b.time, b.uid, b.keyword, b.url\_rank, b.click\_num, b.click\_url  FROM bigtable b  JOIN smalltable s  ON s.id = b.id; |

Time taken: 28.46 seconds

#### 9.2.3 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。

1）开启Map端聚合参数设置

（1）是否在Map端进行聚合，默认为True

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）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。

#### 9.2.4 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：

环境准备：

|  |
| --- |
| create table bigtable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  load data local inpath '/home/admin/softwares/data/100万条大表数据（id除以10取整）/bigtable' into table bigtable; |

|  |
| --- |
| set hive.exec.reducers.bytes.per.reducer=32123456;  SELECT count(DISTINCT id) FROM bigtable;  结果：  c0  10000  Time taken: 35.49 seconds, Fetched: 1 row(s) |

**可以转换成：**

|  |
| --- |
| set hive.exec.reducers.bytes.per.reducer=32123456;  SELECT count(id) FROM (SELECT id FROM bigtable GROUP BY id) a;  结果：  Stage-Stage-1: Map: 1 Reduce: 4 Cumulative CPU: 13.07 sec HDFS Read: 120749896 HDFS Write: 464 SUCCESS  Stage-Stage-2: Map: 3 Reduce: 1 Cumulative CPU: 5.14 sec HDFS Read: 8987 HDFS Write: 7 SUCCESS  \_c0  10000  Time taken: 51.202 seconds, Fetched: 1 row(s) |

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。

#### 9.2.5 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。

#### 9.2.6 使用分区剪裁、列剪裁

在SELECT中，只拿需要的列，如果有，尽量使用分区过滤，少用SELECT \*。

在分区剪裁中，当使用外关联时，如果将副表的过滤条件写在Where后面，那么就会先全表关联，之后再过滤，比如：

环境准备：

|  |
| --- |
| create table ori(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  create table bigtable(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) row format delimited fields terminated by '\t';  load data local inpath '/home/admin/softwares/data/加递增id的原始数据/ori' into table ori;  load data local inpath '/home/admin/softwares/data/100万条大表数据（id除以10取整）/bigtable' into table bigtable; |

先关联再Where：

|  |
| --- |
| SELECT a.id  FROM bigtable a  LEFT JOIN ori b ON a.id = b.id  WHERE b.id <= 10; |

正确的写法是写在ON后面：先Where再关联

|  |
| --- |
| SELECT a.id  FROM ori a  LEFT JOIN bigtable b ON (a.id <= 10 AND a.id = b.id); |

或者直接写成子查询：

|  |
| --- |
| SELECT a.id  FROM bigtable a  RIGHT JOIN (SELECT id  FROM ori  WHERE id <= 10  ) b ON a.id = b.id; |

#### 9.2.7 动态分区调整

关系型数据库中，对分区表Insert数据时候，数据库自动会根据分区字段的值，将数据插入到相应的分区中，Hive中也提供了类似的机制，即动态分区(Dynamic Partition)，只不过，使用Hive的动态分区，需要进行相应的配置。

说白了就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

1）开启动态分区参数设置

（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。比如：源数据中包含了一年的数据，即day字段有365个值，那么该参数就需要设置成大于365，如果使用默认值100，则会报错。

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;

2）案例实操

需求：将ori中的数据按照时间(如：20111231234568)，插入到目标表ori\_partitioned的相应分区中。

（1）准备数据原表

|  |
| --- |
| create table ori\_partitioned(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string)  PARTITIONED BY (p\_time bigint)  row format delimited fields terminated by '\t';  load data local inpath '/export/servers/hivedatas/small\_data' into table ori\_partitioned partition (p\_time='20111230000010');  load data local inpath '/export/servers/hivedatas/small\_data' into table ori\_partitioned partition (p\_time='20111230000011'); |

（2）创建分区表

|  |
| --- |
| create table ori\_partitioned\_target(id bigint, time bigint, uid string, keyword string, url\_rank int, click\_num int, click\_url string) PARTITIONED BY (p\_time STRING) row format delimited fields terminated by '\t'; |

（3）分析

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。

|  |
| --- |
| set hive.exec.dynamic.partition = true;  set hive.exec.dynamic.partition.mode = nonstrict;  set hive.exec.max.dynamic.partitions = 1000;  set hive.exec.max.dynamic.partitions.pernode = 100;  set hive.exec.max.created.files = 100000;  set hive.error.on.empty.partition = false;  INSERT overwrite TABLE ori\_partitioned\_target PARTITION (p\_time)  SELECT id, time, uid, keyword, url\_rank, click\_num, click\_url, p\_time  FROM ori\_partitioned; |

注意：在**PARTITION (month,day)**中指定分区字段名即可；

在SELECT子句的最后几个字段，必须对应前面**PARTITION (month,day)**中指定的分区字段，包括顺序。

查看分区

hive> show partitions ori\_partitioned\_target;

OK

p\_time=20111230000010

p\_time=20111230000011

#### 9.2.8 分桶

参见分桶表

### 9.3 数据倾斜

#### 9.3.1 Map数

**1）通常情况下，作业会通过input的目录产生一个或者多个map任务。**

主要的决定因素有：input的文件总个数，input的文件大小，集群设置的文件块大小(目前为128M，可在hive中通过set dfs.block.size;命令查看到，该参数不能自定义修改)；

**2）举例：**

a)  假设input目录下有1个文件a，大小为780M，那么hadoop会将该文件a分隔成7个块（6个128m的块和1个12m的块），从而产生7个map数。

b) 假设input目录下有3个文件a，b，c大小分别为10m，20m，150m，那么hadoop会分隔成4个块（10m，20m，128m，22m），从而产生4个map数。即，如果文件大于块大小(128m)，那么会拆分，如果小于块大小，则把该文件当成一个块。

**3）是不是map数越多越好？**

答案是否定的。如果一个任务有很多小文件（远远小于块大小128m），则每个小文件也会被当做一个块，用一个map任务来完成，而一个map任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的map数是受限的。

**4）是不是保证每个map处理接近128m的文件块，就高枕无忧了？**

答案也是不一定。比如有一个127m的文件，正常会用一个map去完成，但这个文件只有一个或者两个小字段，却有几千万的记录，如果map处理的逻辑比较复杂，用一个map任务去做，肯定也比较耗时。

针对上面的问题3和4，我们需要采取两种方式来解决：即减少map数和增加map数；

#### 9.3.2 小文件进行合并

在map执行前合并小文件，减少map数：

1. 参数设置（下面的API属于hadoop低版本的API）

set mapred.max.split.size=112345600;

set mapred.min.split.size.per.node=112345600;

set mapred.min.split.size.per.rack=112345600;

set hive.input.format= org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;

这个参数表示执行前进行小文件合并，前面三个参数确定合并文件块的大小，大于文件块大小128m的，按照128m来分隔，小于128m，大于100m的，按照100m来分隔，把那些小于100m的（包括小文件和分隔大文件剩下的），进行合并。

#### 9.3.3 如何适当的增加map数

当input的文件都很大，任务逻辑复杂，map执行非常慢的时候，可以考虑增加Map数，来使得每个map处理的数据量减少，从而提高任务的执行效率。

假设有这样一个任务：

|  |
| --- |
| Select data\_desc,  count(1),  count(distinct id),  sum(case when …),  sum(case when …),  sum(…)  from a group by data\_desc |

如果表a只有一个文件，大小为120M，但包含几千万的记录，如果用1个map去完成这个任务，肯定是比较耗时的，这种情况下，我们要考虑将这一个文件合理的拆分成多个，这样就可以用多个map任务去完成。

|  |
| --- |
| set mapreduce.job.reduces =10;  create table a\_1 as  select \* from a  distribute by rand(123); |

这样会将a表的记录，随机的分散到包含10个文件的a\_1表中，再用a\_1代替上面sql中的a表，则会用10个map任务去完成。

每个map任务处理大于12M（几百万记录）的数据，效率肯定会好很多。

看上去，貌似这两种有些矛盾，一个是要合并小文件，一个是要把大文件拆成小文件，这点正是重点需要关注的地方，根据实际情况，控制map数量需要遵循两个原则：使大数据量利用合适的map数；使单个map任务处理合适的数据量；

#### 9.3.4 reduce数

**1）调整reduce个数方法一**

（1）每个Reduce处理的数据量默认是256MB

hive.exec.reducers.bytes.per.reducer=256123456

（2）每个任务最大的reduce数，默认为1009

hive.exec.reducers.max=1009

（3）计算reducer数的公式

N=min(参数2，总输入数据量/参数1)

**2）调整reduce个数方法二**

在hadoop的mapred-default.xml文件中修改

设置每个job的Reduce个数

set mapreduce.job.reduces = 15;

**3）reduce个数并不是越多越好**

1）过多的启动和初始化reduce也会消耗时间和资源；

2）另外，有多少个reduce，就会有多少个输出文件，如果生成了很多个小文件，那么如果这些小文件作为下一个任务的输入，则也会出现小文件过多的问题；

在设置reduce个数的时候也需要考虑这两个原则：处理大数据量利用合适的reduce数；使单个reduce任务处理数据量大小要合适；

### 9.4 使用EXPLAIN（执行计划）

1）基本语法

EXPLAIN [EXTENDED | DEPENDENCY | AUTHORIZATION] query

2）案例实操

（1）查看下面这条语句的执行计划

hive (default)> explain select \* from course;

hive (default)> explain select s\_id ,avg(s\_score) avgscore from score group by s\_id;

（2）查看详细执行计划

hive (default)> explain extended select \* from course;

hive (default)> explain extended select s\_id ,avg(s\_score) avgscore from score group by s\_id;

### 9.5 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。

|  |
| --- |
| set hive.exec.parallel=true;   //打开任务并行执行  set hive.exec.parallel.thread.number=16; //同一个sql允许最大并行度，默认为8。 |

当然，得是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。

### 9.6设置本地模式

这个命令必须要在hive里面的执行才行

Set hive.exex.mode.local.auto=true;

### 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向不到的不好的影响的查询。

通过设置属性hive.mapred.mode值为默认是非严格模式nonstrict 。开启严格模式需要修改hive.mapred.mode值为strict，开启严格模式可以禁止3种类型的查询。

|  |
| --- |
| <property>  <name>hive.mapred.mode</name>  <value>strict</value>  <description>  The mode in which the Hive operations are being performed.  In strict mode, some risky queries are not allowed to run. They include:  Cartesian Product.  No partition being picked up for a query.  Comparing bigints and strings.  Comparing bigints and doubles.  Orderby without limit.  </description>  </property> |

1）对于分区表，除非where语句中含有分区字段过滤条件来限制范围，否则不允许执行。换句话说，就是用户不允许扫描所有分区。进行这个限制的原因是，通常分区表都拥有非常大的数据集，而且数据增加迅速。没有进行分区限制的查询可能会消耗令人不可接受的巨大资源来处理这个表。

2）对于使用了order by语句的查询，要求必须使用limit语句。因为order by为了执行排序过程会将所有的结果数据分发到同一个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。

3）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。

### 9.7 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。

|  |
| --- |
| <property>  <name>mapreduce.job.jvm.numtasks</name>  <value>10</value>  <description>How many tasks to run per jvm. If set to -1, there is  no limit.  </description>  </property> |

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。

### 9.8 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，它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：Hadoop的mapred-site.xml文件中进行配置

|  |
| --- |
| <property>  <name>mapreduce.map.speculative</name>  <value>true</value>  <description>If true, then multiple instances of some map tasks  may be executed in parallel.</description>  </property>  <property>  <name>mapreduce.reduce.speculative</name>  <value>true</value>  <description>If true, then multiple instances of some reduce tasks  may be executed in parallel.</description>  </property> |

不过hive本身也提供了配置项来控制reduce-side的推测执行：

|  |
| --- |
| <property>  <name>hive.mapred.reduce.tasks.speculative.execution</name>  <value>true</value>  <description>Whether speculative execution for reducers should be turned on. </description>  </property> |

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大。

### 9.9 压缩

参见前面

## 十、hive语句综合练习

<https://blog.csdn.net/mrbcy/article/details/68965271>

1、查询"01"课程比"02"课程成绩高的学生的信息及课程分数

SELECT a.\* ,b.s\_score AS 01\_score,c.s\_score AS 02\_score FROM

student a

LEFT JOIN score b ON a.s\_id=b.s\_id AND b.c\_id='01'

LEFT JOIN score c ON a.s\_id=c.s\_id AND c.c\_id = '02' WHERE b.s\_score>c.s\_score;

2、查询"01"课程比"02"课程成绩低的学生的信息及课程分数

SELECT a.\* ,b.s\_score AS 01\_score,c.s\_score AS 02\_score FROM

student a LEFT JOIN score b ON a.s\_id=b.s\_id AND b.c\_id='01'

JOIN score c ON a.s\_id=c.s\_id AND c.c\_id='02' WHERE b.s\_score<c.s\_score;

3、查询平均成绩大于等于60分的同学的学生编号和学生姓名和平均成绩

SELECT b.s\_id,b.s\_name,ROUND(AVG(a.s\_score),2) AS avg\_score FROM

student b

JOIN score a ON b.s\_id = a.s\_id

GROUP BY b.s\_id,b.s\_name HAVING ROUND(AVG(a.s\_score),2)>=60;

4、查询平均成绩小于60分的同学的学生编号和学生姓名和平均成绩

-- (包括有成绩的和无成绩的)

SELECT b.s\_id,b.s\_name,ROUND(AVG(a.s\_score),2) AS avg\_score FROM

student b

LEFT JOIN score a ON b.s\_id = a.s\_id

GROUP BY b.s\_id,b.s\_name HAVING ROUND(AVG(a.s\_score),2)<60

UNION ALL

SELECT a.s\_id,a.s\_name,0 AS avg\_score FROM

student a

WHERE a.s\_id NOT IN (

SELECT DISTINCT s\_id FROM score);

-- 5、查询所有同学的学生编号、学生姓名、选课总数、所有课程的总成绩

SELECT a.s\_id,a.s\_name,COUNT(b.c\_id) AS sum\_course,SUM(b.s\_score) AS sum\_score FROM

student a

LEFT JOIN score b ON a.s\_id=b.s\_id

GROUP BY a.s\_id,a.s\_name;

-- 6、查询"李"姓老师的数量

select count(t\_id) from techer where t\_name like '李%';

-- 7、查询学过"张三"老师授课的同学的信息

SELECT a.\*

FROM student a LEFT JOIN score b ON a.s\_id = b.s\_id WHERE b.c\_id IN (

SELECT c.c\_id

FROM course c LEFT JOIN techer t ON c.t\_id = t.t\_id WHERE t.t\_name = '张三'

) ;

-- 8、查询没学过"张三"老师授课的同学的信息

SELECT s.\*

FROM student s LEFT JOIN (

SELECT a.s\_id

FROM student a LEFT JOIN score b ON a.s\_id = b.s\_id WHERE b.c\_id IN (

SELECT c.c\_id

FROM course c LEFT JOIN techer t ON c.t\_id = t.t\_id WHERE t.t\_name = '张三'

)

) ss ON s.s\_id = ss.s\_id WHERE ss.s\_id IS NULL;

-- 9、查询学过编号为"01"并且也学过编号为"02"的课程的同学的信息

select a.\* from

student a,score b,score c

where a.s\_id = b.s\_id and a.s\_id = c.s\_id and b.c\_id='01' and c.c\_id='02';

-- 10、查询学过编号为"01"但是没有学过编号为"02"的课程的同学的信息

SELECT qq.\*

FROM (

SELECT s.\*

FROM student s LEFT JOIN score sco ON s.s\_id = sco.s\_id LEFT JOIN course c ON sco.c\_id = c.c\_id WHERE c.c\_id='01'

) qq

LEFT JOIN (

SELECT stu.\*

FROM student stu LEFT JOIN score mysco ON stu.s\_id = mysco.s\_id LEFT JOIN course cou ON mysco.c\_id = cou.c\_id WHERE cou.c\_id='02'

) pp

ON qq.s\_id = pp.s\_id WHERE pp.s\_id IS NULL;

-- 11、查询没有学全所有课程的同学的信息

SELECT ss.s\_id

FROM (

SELECT stu.s\_id,COUNT(stu.s\_id) AS num

FROM student stu LEFT JOIN score sco ON stu.s\_id = sco.s\_id LEFT JOIN course cou ON sco.c\_id = cou.c\_id

GROUP BY stu.s\_id

) ss WHERE ss.num < 3

-- 12、查询至少有一门课与学号为"01"的同学所学相同的同学的信息

SELECT stu.\*

FROM student stu LEFT JOIN

(

SELECT s.s\_id

FROM score s WHERE s.c\_id IN(

SELECT c\_id FROM score WHERE s\_id = '01'

)GROUP BY s\_id

) pp ON stu.s\_id = pp.s\_id WHERE pp.s\_id IS NOT NULL;