



# 数据仓库-Hive

## 1. 数据仓库

### 1.1. 基本概念

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

数据仓库是存数据的，企业的各种数据往里面存，主要目的是为了分析有效数据，后续会基于它产出供分析挖掘的数据，或者数据应用需要的数据，如企业的分析性报告和各类报表等。

可以理解为：**面向分析的存储系统**。

### 1.2. 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。

#### 1.2.1. 面向主题

数据仓库是面向主题的，数据仓库通过一个个主题域将多个业务系统的数据加载到一起，为了各个主题（如：用户、订单、商品等）进行分析而建，操作型数据库是为了支撑各种业务而建立。

#### 1.2.2. 集成性

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

#### 1.2.3. 非易失性



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的集合，以及基于这些快照进行统计、综合和重组的导出数据。数据仓库中的数据一般仅执行查询操作，很少会有删除和更新。但是需定期加载和刷新数据。

#### 1.2.4. 时变性

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定时更新，以适应决策的需要。

### 1.3. 数据库与数据仓库的区别

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。

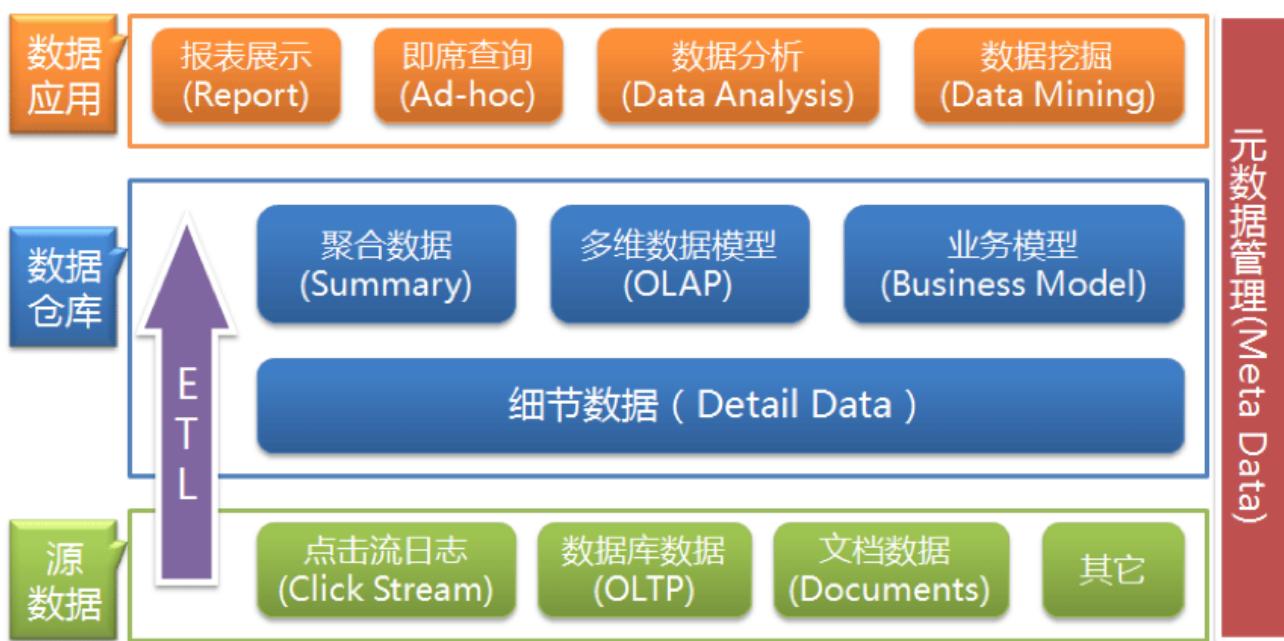
- 数据库是面向事务的设计，数据仓库是面向主题设计的。
- 数据库一般存储业务数据，数据仓库存储的一般是历史数据。
- 数据库设计是尽量避免冗余，一般针对某一业务应用进行设计，比如一张简单的User表，记录用户名、密码等简单数据即可，符合业务应用，但是不符合分析。数据仓库在设计是有意引入冗余，依照分析需求，分析维度、分析指标进行设计。
- 数据库是为捕获数据而设计，数据仓库是为分析数据而设计。



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”。

## 1.4. 数仓的分层架构

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。

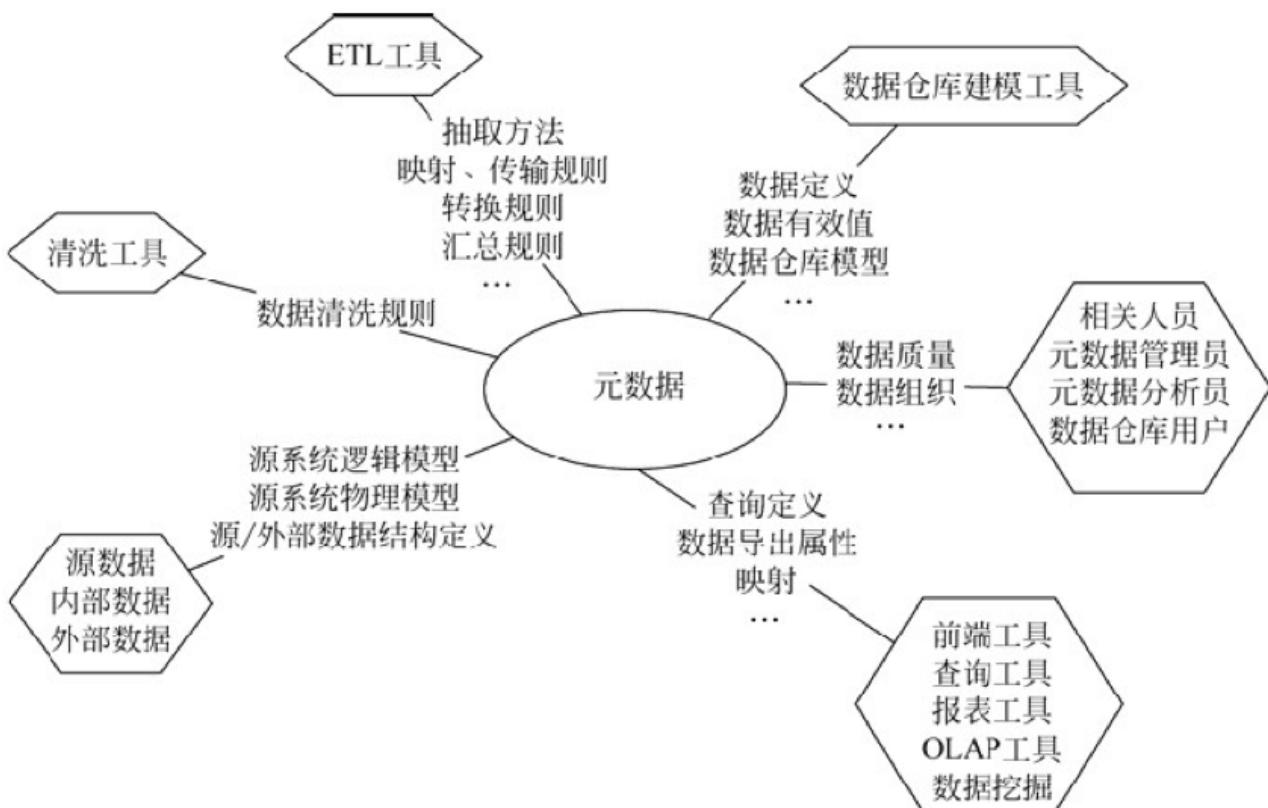
通过数据分层管理可以简化数据清洗的过程，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

## 1.5. 数仓的元数据管理

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。

## 2. Hive 的基本概念

### 2.1. Hive 简介

#### 什么是 Hive

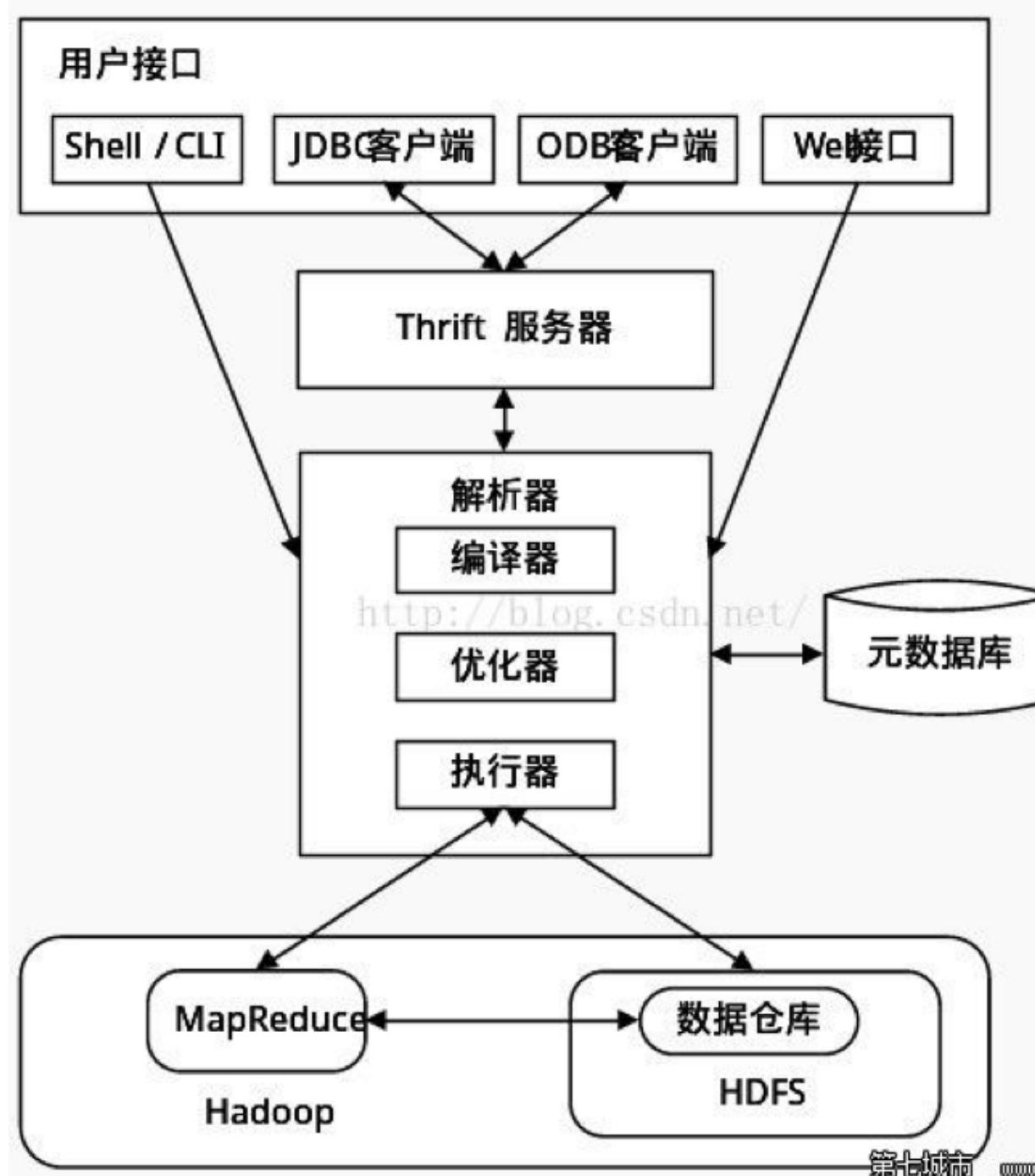
Hive是基于Hadoop的一个数据仓库工具，可以将**结构化的数据**文件映射为一张数据库表，并提供类SQL查询功能。

其本质是将SQL转换为MapReduce的任务进行运算，底层由HDFS来提供数据的存储，说白了hive可以理解为一个将SQL转换为MapReduce的任务的工具，甚至更进一步可以说hive就是一个MapReduce的客户端

## 为什么使用 Hive

- 采用类SQL语法去操作数据，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。

## 2.2. Hive 架构

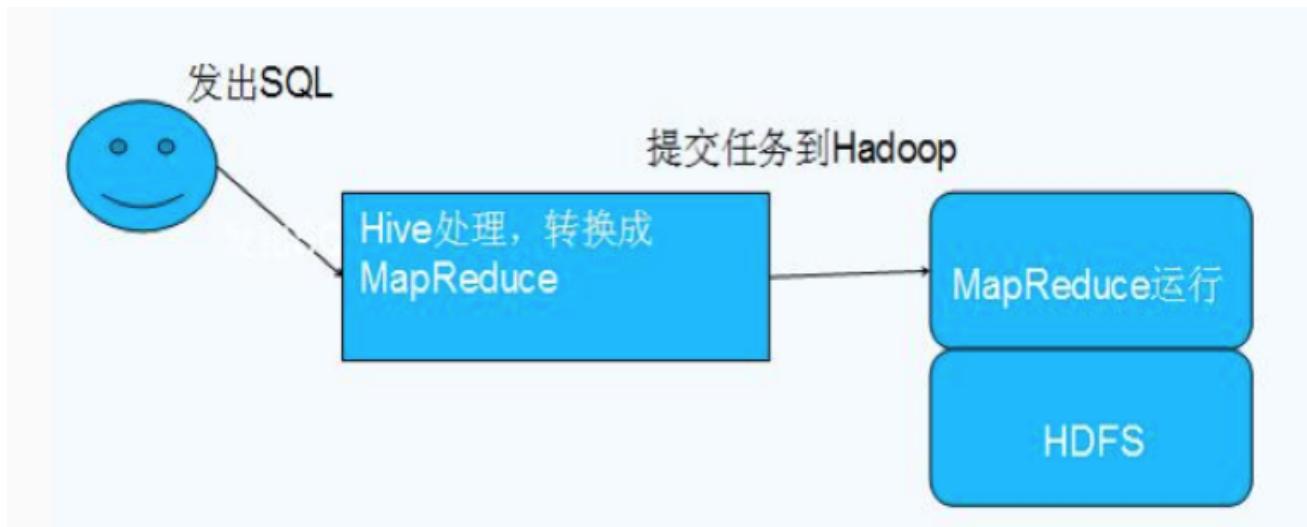




- 用户接口：**包括CLI、JDBC/ODBC、WebGUI。其中，CLI(command line interface)为shell命令行；JDBC/ODBC是Hive的JAVA实现，与传统数据库JDBC类似；WebGUI是通过浏览器访问Hive。
- 元数据存储：**通常是存储在关系数据库如mysql/derby中。Hive将元数据存储在数据库中。Hive中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
- 解释器、编译器、优化器、执行器：**完成HQL查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在HDFS中，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



|        | Hive         | RDBMS                  |
|--------|--------------|------------------------|
| 查询语言   | HQL          | SQL                    |
| 数据存储   | HDFS         | Raw Device or Local FS |
| 执行     | MapReduce    | Executor               |
| 执行延迟   | 高            | 低                      |
| 处理数据规模 | 大            | 小                      |
| 索引     | 0.8版本后加入位图索引 | 有复杂的索引                 |

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 的安装

这里我们选用hive的版本是2.1.1 下载地址为：<http://archive.apache.org/dist/hive/hive-2.1.1/apache-hive-2.1.1-bin.tar.gz>

下载之后，将我们的安装包上传到第三台机器的/export/softwares目录下面去

### 第一步：上传并解压安装包

将我们的hive的安装包上传到第三台服务器的/export/softwares路径下，然后进行解压

```
1 cd /export/softwares/
2 tar -zxvf apache-hive-2.1.1-bin.tar.gz -C ../servers/
```

### 第二步：安装mysql

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;
```

### 第三步：修改hive的配置文件

#### 修改hive-env.sh

```
1 cd /export/servers/apache-hive-2.1.1-bin/conf
2 cp hive-env.sh.template hive-env.sh
```

```
1 HADOOP_HOME=/export/servers/hadoop-2.7.5
2 export HIVE_CONF_DIR=/export/servers/apache-hive-2.1.1-bin/conf
```

#### 修改hive-site.xml

```
1 cd /export/servers/apache-hive-2.1.1-bin/conf
2 vim hive-site.xml
```

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <?xmlstylesheet type="text/xsl" href="configuration.xsl"?>
3 <configuration>
4   <property>
5     <name>javax.jdo.option.ConnectionUserName</name>
6     <value>root</value>
7   </property>
8   <property>
9     <name>javax.jdo.option.ConnectionPassword</name>
10    <value>123456</value>
11  </property>
12  <property>
13    <name>javax.jdo.option.ConnectionURL</name>
14    <value>jdbc:mysql://node03:3306/hive?
createDatabaseIfNotExist=true&useSSL=false</value>
15  </property>
```



```
16 <property>
17   <name>javax.jdo.option.ConnectionDriverName</name>
18   <value>com.mysql.jdbc.Driver</value>
19 </property>
20 <property>
21   <name>hive.metastore.schema.verification</name>
22   <value>false</value>
23 </property>
24 <property>
25   <name>datanucleus.schema.autoCreateAll</name>
26   <value>true</value>
27 </property>
28 <property>
29   <name>hive.server2.thrift.bind.host</name>
30   <value>node03</value>
31 </property>
32 </configuration>
```

#### 第四步：添加mysql的连接驱动包到hive的lib目录下

hive使用mysql作为元数据存储，必然需要连接mysql数据库，所以我们添加一个mysql的连接驱动包到hive的安装目录下，然后就可以准备启动hive了

将我们准备好的mysql-connector-java-5.1.38.jar这个jar包直接上传到  
`/export/servers/apache-hive-2.1.1-bin/lib` 这个目录下即可

至此，hive的安装部署已经完成，接下来我们来看下hive的三种交互方式

#### 第五步：配置hive的环境变量

node03服务器执行以下命令配置hive的环境变量

```
1 sudo vim /etc/profile
```

```
1 export HIVE_HOME=/export/servers/apache-hive-2.1.1-bin
2 export PATH=$HIVE_HOME/bin:$PATH
```

## 2.6. Hive 的交互方式

### 第一种交互方式 `bin/hive`



```
1 cd /export/servers/apache-hive-2.1.1-bin/  
2 bin/hive
```

创建一个数据库

```
1 create database if not exists mytest;
```

## 第二种交互方式： 使用sql语句或者sql脚本进行交互

不进入hive的客户端直接执行hive的hql语句

```
1 cd /export/servers/apache-hive-2.1.1-bin  
2 bin/hive -e "create database if not exists mytest;"
```

或者我们可以将我们的hql语句写成一个sql脚本然后执行

```
1 cd /export/servers  
2 vim hive.sql
```

```
1 create database if not exists mytest;  
2 use mytest;  
3 create table stu(id int, name string);
```

通过hive -f 来执行我们的sql脚本

```
1 bin/hive -f /export/servers/hive.sql
```

## 3. Hive 的基本操作

### 3.1 数据库操作

#### 3.1.1 创建数据库

```
1 create database if not exists myhive;  
2 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



```
1 <name>hive.metastore.warehouse.dir</name>
2 <value>/user/hive/warehouse</value>
```

### 3.1.2 创建数据库并指定位置

```
1 create database myhive2 location '/myhive2' ;
```

### 3.1.3 设置数据库键值对信息

数据库可以有一些描述性的键值对信息，在创建时添加：

```
1 create database foo with dbproperties ('owner'='itcast',
'date'='20190120');
```

查看数据库的键值对信息：

```
1 describe database extended foo;
```

修改数据库的键值对信息：

```
1 alter database foo set dbproperties ('owner'='itheima');
```

### 3.1.4 查看数据库更多详细信息

```
1 desc database extended myhive2;
```

### 3.1.5 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

```
1 drop database myhive2;
```

强制删除数据库，包含数据库下面的表一起删除

```
1 drop database myhive cascade;
```

## 3.2 数据库表操作

### 3.2.1 创建表的语法：



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

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**

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. **comment**

表示注释，默认不能使用中文

4. **partitioned by**

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

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

6. **sorted by**

指定排序字段和排序规则

7. **row format**

指定表文件字段分隔符

8. **storted as** 指定表文件的存储格式，常用格式：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 storted as SEQUENCEFILE。

9. **location**



## 指定表文件的存储路径



### 3.2.2 内部表的操作

创建表时,如果没有使用external关键字,则该表是内部表 (managed table)

**Hive建表字段类型**



| 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,<br>-128~127       | 1Y           |
|      | SMALLINT  | 2个字节的有符号整<br>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          |
|      | DECIMAL   | 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, 毫秒值精度                   | 122327493795 |
|      | DATE      | 日期                           | ‘2016-03-29’ |
|      | INTERVAL  | 时间频率间隔                       |              |
| 复杂类型 | ARRAY     | 有序的的同类型的集合                   | array(1,2)   |



| 分类 | 类型     | 描述                                 | 字面量示例  |
|----|--------|------------------------------------|--|
|    | 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),<br>named_struct('col1','1','col2',1,'col3',1.0) |
|    | UNION  | 在有限取值范围内的一个值                       | create_union(1,'a',63)   |

## 建表入门:

```
1 use myhive;
2 create table stu(id int, name string);
3 insert into stu values (1, "zhangsan"); #插入数据
4 select * from stu;
```

## 创建表并指定字段之间的分隔符

```
1 create table if not exists stu2(id int ,name string) row format delimited
  fields terminated by '\t';
```

## 创建表并指定表文件的存放路径

```
1 create table if not exists stu2(id int ,name string) row format delimited
  fields terminated by '\t' location '/user/stu2';
```

## 根据查询结果创建表

```
1 create table stu3 as select * from stu2; # 通过复制表结构和表内容创建新表
```

## 根据已经存在的表结构创建表

```
1 create table stu4 like stu;
```

## 查询表的详细信息



```
1 desc formatted stu2;
```

## . 删除表

```
1 drop table stu4;
```

### 3.2.3 外部表的操作

#### 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。

#### 内部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

```
1 create external table teacher (t_id string,t_name string) row format  
delimited fields terminated by '\t';
```

#### 创建学生表

```
1 create external table student (s_id string,s_name string,s_birth string ,  
s_sex string ) row format delimited fields terminated by '\t';
```

#### 加载数据

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

#### 加载数据并覆盖已有数据



```
1 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）

```
1 cd /export/servers/hivedatas
2 hdfs dfs -mkdir -p /hivedatas
3 hdfs dfs -put techer.csv /hivedatas/
4 load data inpath '/hivedatas/techer.csv' into table teacher;
```

### 3.2.4 分区表的操作

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，按照每月，或者天进行切分成一个个的小的文件，存放在不同的文件夹中。

#### 创建分区表语法

```
1 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

```
1 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

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

#### 加载数据到多分区表中

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

#### 多分区表联合查询(使用 `union all`)



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

## 查看分区

```
1 show partitions score;
```

## 添加一个分区

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

## 删除分区

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

## 3.2.5 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

### 数据准备：

```
1 hdfs dfs -mkdir -p /scoredatas/month=201806
2 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

```
1 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(建立表与数据文件之间的一个关系映射)

```
1 msck repair table score4;
```



### 3.2.6 分桶表操作

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

开启 Hive 的分桶功能

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

设置 Reduce 个数

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

创建分桶表

```
1 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 overwriter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

```
1 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

```
1 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

```
1 insert overwrite table course select * from course_common cluster  
by(c_id);
```

### 3.3 修改表结构

重命名：



```
1 alter table old_table_name rename to new_table_name;
```

把表score4修改成score5

```
1 alter table score4 rename to score5;
```

### 增加/修改列信息:

- 查询表结构

```
1 desc score5;
```

- 添加列

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

- 更新列

```
1 alter table score5 change column mysco mysconew int;
```

- 删除表

```
1 drop table score5;
```

## 1.8. hive表中加载数据

直接向分区表中插入数据

```
1 create table score3 like score;
2
3 insert into table score3 partition(month = '201807') values
('001', '002', '100');
4
```

通过查询插入数据

通过load方式加载数据



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

通过查询方式加载数据

```
1 create table score4 like score;
2 insert overwrite table score4 partition(month = '201806') select
  s_id,c_id,s_score from score;
```

## 4. Hive 查询语法

### 4.1. SELECT

```
1 SELECT [ALL | DISTINCT] select_expr, select_expr, ...
2 FROM table_reference
3 [WHERE where_condition]
4 [GROUP BY col_list [HAVING condition]]
5 [CLUSTER BY col_list
6 | [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]
7 ]
8 [LIMIT number]
```

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distribute 和sort字段是同一个时，此时，`cluster by = distribute by + sort by`

### 4.2. 查询语法

#### 全表查询



```
1 select * from score;
```

## 选择特定列

```
1 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’

```
1 select s_id as myid ,c_id from score;
```

## 4.3. 常用函数

- 求总行数 (count)

```
1 select count(1) from score;
```

- 求分数的最大值 (max)

```
1 select max(s_score) from score;
```

- 求分数的最小值 (min)

```
1 select min(s_score) from score;
```

- 求分数的总和 (sum)

```
1 select sum(s_score) from score;
```

- 求分数的平均值 (avg)

```
1 select avg(s_score) from score;
```

## 4.4. LIMIT语句

典型的查询会返回多行数据。LIMIT子句用于限制返回的行数。



```
1 select * from score limit 3;
```

## 4.5. 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

```
1 select * from score where s_score > 60;
```

### 比较运算符



| 操作符                     | 支持的数据类型      | 描述   |
|-------------------------|--------------|--|
| A=B                     | 基本数据类型       | 如果A等于B则返回TRUE, 反之返回FALSE   |
| A<=>B                   | 基本数据类型       | 如果A和B都为NULL, 则返回TRUE, 其他的和等号(=)操作符的结果一致, 如果任一为NULL则结果为NULL   |
| A<>B,<br>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,<br>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<br>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。 |



| 操作符                         | 支持的数据类型 | 描述   |
|-----------------------------|---------|--|
| A RLIKE<br>B, A<br>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。 |

- 查询分数等于80的所有数据

```
1 select * from score where s_score = 80;
```

- 查询分数在80到100的所有数据

```
1 select * from score where s_score between 80 and 100;
```

- 查询成绩为空的所有数据

```
1 select * from score where s_score is null;
```

- 查询成绩是80和90的数据

```
1 select * from score where s_score in(80,90);
```

## 4.6. LIKE 和 RLIKE

- 使用LIKE运算选择类似的值
- 选择条件可以包含字符或数字：

```
1 % 代表零个或多个字符(任意个字符)。  
2 _ 代表一个字符。
```

- 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。
- 案例实操

- 查找以8开头的所有成绩

```
1 select * from score where s_score like '8%';
```



- 查找第二个数值为9的所有成绩数据

```
1 select * from score where s_score like '_9%';
```

- 查找s\_id中含1的数据

```
1 select * from score where s_id rlike '[1]'; # like '%1%'
```

## 4.7. 逻辑运算符

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

- 查询成绩大于80，并且s\_id是01的数据

```
1 select * from score where s_score >80 and s_id = '01';
```

- 查询成绩大于80，或者s\_id 是01的数据

```
1 select * from score where s_score > 80 or s_id = '01';
```

- 查询s\_id 不是 01和02的学生

```
1 select * from score where s_id not in ('01','02');
```

## 4.8. 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：

- 计算每个学生的平均分数

```
1 select s_id ,avg(s_score) from score group by s_id;
```



- 计算每个学生最高成绩

```
1 select s_id ,max(s_score) from score group by s_id;
```

## HAVING 语句

- 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

```
1 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

```
1 select s_id ,avg(s_score) avgscore from score group by s_id having avgscore > 85;
```

## 4.9. JOIN 语句

### 4.9.1. 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：查询分数对应的姓名

```
1 select s.s_id,s.s_score,stu.s_name,stu.s_birth  from score s  join student stu on s.s_id = stu.s_id;
```

### 4.9.2. 表的别名

- 好处

- 使用别名可以简化查询。
- 使用表名前缀可以提高执行效率。



- 案例实操

- 合并老师与课程表

```
1 select * from teacher t join course c on t.t_id = c.t_id;
```

### 4.9.3. 内连接

内连接：只有进行连接的两个表中都存在与连接条件相匹配的数据才会被保留下。

```
1 select * from teacher t inner join course c on t.t_id = c.t_id;
```

### 4.9.4. 左外连接

左外连接：JOIN操作符左边表中符合WHERE子句的所有记录将会被返回。查询老师对应的课程

```
1 select * from teacher t left join course c on t.t_id = c.t_id;
```

### 4.9.5. 右外连接

右外连接：JOIN操作符右边表中符合WHERE子句的所有记录将会被返回。

```
1 select * from teacher t right join course c on t.t_id = c.t_id;
```

### 4.9.6. 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

```
1 select * from teacher t
2 left join course c
3 on t.t_id = c.t_id
4 left join score s
5 on s.c_id = c.c_id
6 left join student stu
7 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teacher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。



## 4.10. 排序

### 4.10.1. 全局排序

Order By：全局排序，一个reduce

1. 使用 ORDER BY 子句排序 ASC (ascend) :升序 (默认) DESC (descend) :降序
2. ORDER BY 子句在SELECT语句的结尾。
3. 案例实操
  1. 查询学生的成绩，并按照分数降序排列

```
1   SELECT * FROM student s LEFT JOIN score sco ON s.s_id = sco.s_id
      ORDER BY sco.s_score DESC;
```

1. 查询学生的成绩，并按照分数升序排列

```
1   SELECT * FROM student s LEFT JOIN score sco ON s.s_id = sco.s_id
      ORDER BY sco.s_score asc;
```

### 4.10.2. 按照别名排序

按照分数的平均值排序

```
1   select s_id ,avg(s_score) avg from score group by s_id order by avg;
```

### 4.10.3. 多个列排序

按照学生id和平均成绩进行排序

```
1   select s_id ,avg(s_score) avg from score group by s_id order by s_id,avg;
```

### 4.10.4. 每个MapReduce内部排序 (Sort By) 局部排序

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

1. 设置reduce个数

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



### 1. 查看设置reduce个数

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

### 1. 查询成绩按照成绩降序排列

```
1 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）

```
1 insert overwrite local directory '/export/servers/hivedatas/sort' select *
  from score sort by s_score;
```

## 4.10.5. 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：先按照学生id进行分区，再按照学生成绩进行排序。

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

```
1 set mapreduce.job.reduces=7;
```

### 1. 通过distribute by 进行数据的分区

```
1 insert overwrite local directory '/export/servers/hivedatas/sort' select *
  from score distribute by s_id sort by s_score;
```

## 4.10.6. 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。

以下两种写法等价



```
1 select * from score cluster by s_id;
2 select * from score distribute by s_id sort by s_id;
```

## 5.Hive Shell参数

### 5.1 Hive命令行

#### 语法结构

```
1 bin/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** 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

### 5.2 Hive参数配置方式

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。

#### 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：

```
1 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：

```
1 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)

## 6. Hive 函数

### 6.1. 内置函数

内容较多，见《Hive官方文档》

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

#### 1. 查看系统自带的函数



```
1    hive> show functions;
```

## 2. 显示自带的函数的用法

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

## 3. 详细显示自带的函数的用法

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

## 4:常用内置函数

```
1 #字符串连接函数: concat
2   select concat('abc','def','gh');
3 #带分隔符字符串连接函数: concat_ws
4   select concat_ws(',',$abc','$def','$gh');
5 #cast类型转换
6   select cast(1.5 as int);
7 #get_json_object(json 解析函数, 用来处理json, 必须是json格式)
8   select get_json_object('{"name":"jack","age":20}', '$.name');
9 #URL解析函数
10  select parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1',
11    'HOST');
12 #explode: 把map集合中每个键值对或数组中的每个元素都单独生成一行的形式
```

## 6.2. 自定义函数

### 6.2.1 概述:

1. Hive自带了一些函数，比如：max/min等，当Hive提供的内置函数无法满足你的业务处理需要时，此时就可以考虑使用用户自定义函数(UDF).

2. 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()`

#### 3. 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；

#### 4. 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；

## 6.2.2 UDF 开发实例

### Step 1 创建 Maven 工程

```
1 <dependencies>
2     <!-- https://mvnrepository.com/artifact/org.apache.hive/hive-exec -->
3     <dependency>
4         <groupId>org.apache.hive</groupId>
5         <artifactId>hive-exec</artifactId>
6         <version>2.7.5</version>
7     </dependency>
8     <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-
common -->
9     <dependency>
10        <groupId>org.apache.hadoop</groupId>
11        <artifactId>hadoop-common</artifactId>
12        <version>2.7.5</version>
13    </dependency>
14 </dependencies>
15
16 <build>
17     <plugins>
18         <plugin>
19             <groupId>org.apache.maven.plugins</groupId>
20             <artifactId>maven-compiler-plugin</artifactId>
21             <version>3.0</version>
22             <configuration>
23                 <source>1.8</source>
24                 <target>1.8</target>
25                 <encoding>UTF-8</encoding>
26             </configuration>
```

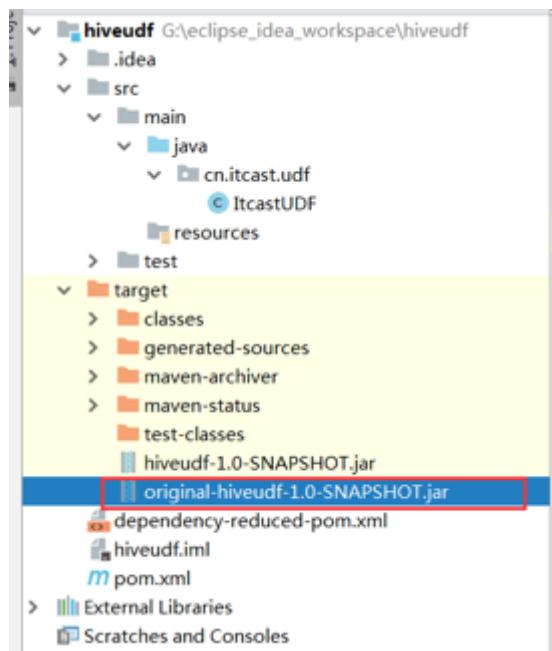


```
27          </plugin>
28      </plugins>
29  </build>
```

## Step 2 开发 Java 类集成 UDF

```
1  public class MyUDF extends UDF{
2      public Text evaluate(final Text str){
3          String tmp_str = str.toString();
4          if(str != null && !tmp_str.equals("")){
5              String str_ret = tmp_str.substring(0, 1).toUpperCase() +
6              tmp_str.substring(1);
7              return new Text(str_ret);
8          }
9      }
10 }
11
```

## Step 3 项目打包，并上传到hive的lib目录下



## Step 4 添加jar包

重命名我们的jar包名称

```
1  cd /export/servers/apache-hive-2.7.5-bin/lib
2  mv original-day_10_hive_udf-1.0-SNAPSHOT.jar my_upper.jar
```



hive的客户端添加我们的jar包

```
1 add jar /export/servers/apache-hive-2.7.5-bin/lib/my_upper.jar;
```

### Step 5 设置函数与我们的自定义函数关联

```
1 create temporary function my_upper as 'cn.itcast.udf.IItcastUDF';
```

### Step 6 使用自定义函数

```
1 select my_upper('abc');
```

## 7.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

### 7.1 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     | lzip  | 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.

## 7.2 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,<br>org.apache.hadoop.io.compress.GzipCodec,<br>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 |

## 7.3 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

```
1 set hive.exec.compress.intermediate=true;
```



## 2) 开启mapreduce中map输出压缩功能

```
1 set mapreduce.map.output.compress=true;
```

## 3) 设置mapreduce中map输出数据的压缩方式

```
1 set mapreduce.map.output.compress.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

## 4) 执行查询语句

```
1 select count(1) from score;
```

## 7.4 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

```
1 set hive.exec.compress.output=true;
```

## 2) 开启mapreduce最终输出数据压缩

```
1 set mapreduce.output.fileoutputformat.compress=true;
```

## 3) 设置mapreduce最终数据输出压缩方式

```
1 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

```
1 set mapreduce.output.fileoutputformat.compress.type=BLOCK;
```

## 5) 测试一下输出结果是否是压缩文件

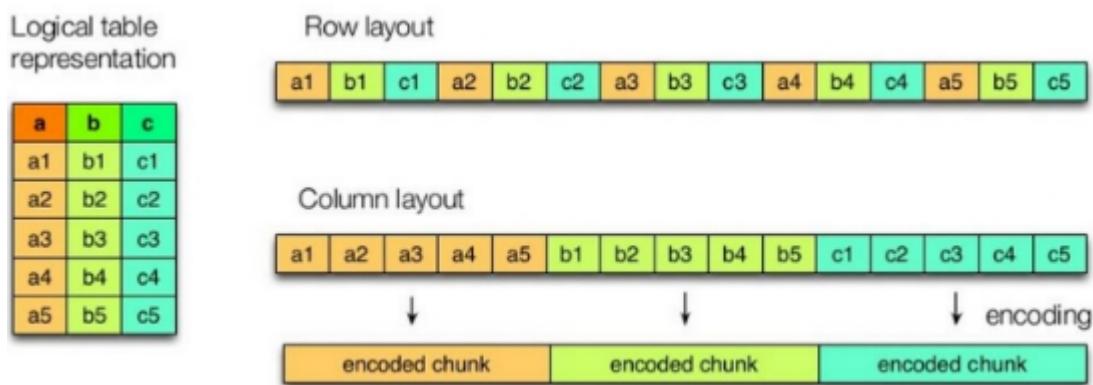


```
1 insert overwrite local directory '/export/servers/snappy' select * from
score distribute by s_id sort by s_id desc;
```

## 8.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）。

### 8.1 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。

### 8.2 常用数据存储格式

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。

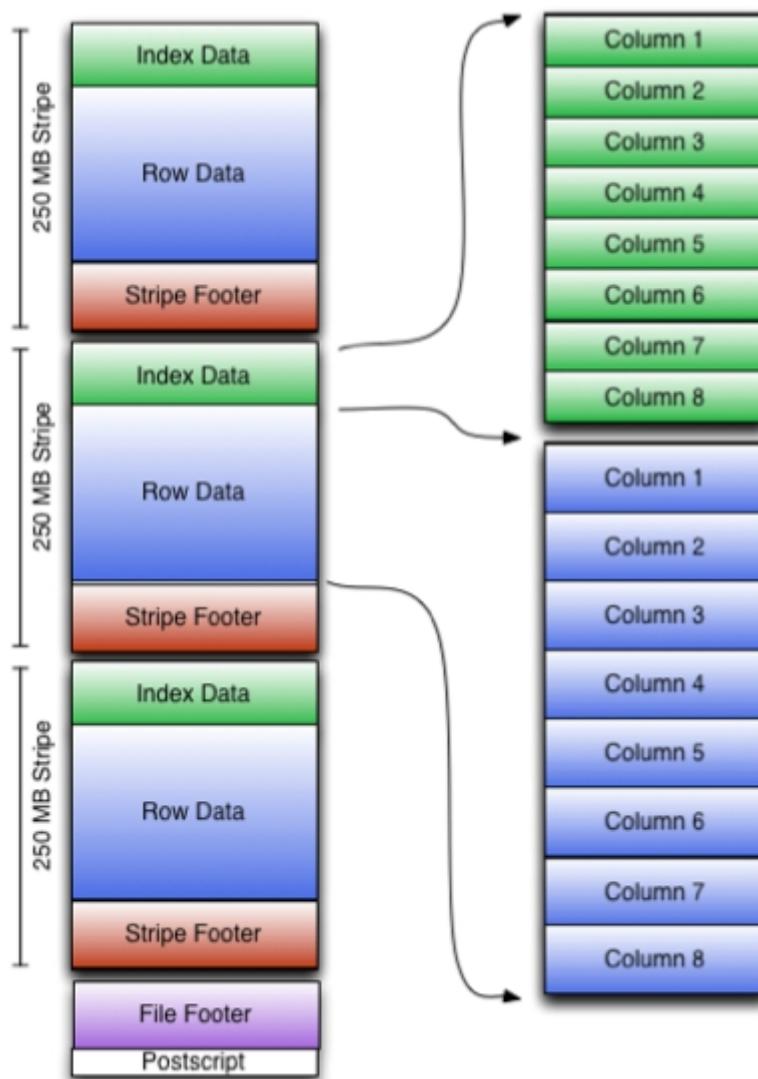
ORC格式



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，每个Stripe里有三部分组成，分别是Index Data,Row Data,Stripe Footer：

- **indexData**：某些列的索引数据
- **rowData** :真正的数据存储
- **StripFooter**：stripe的元数据信息

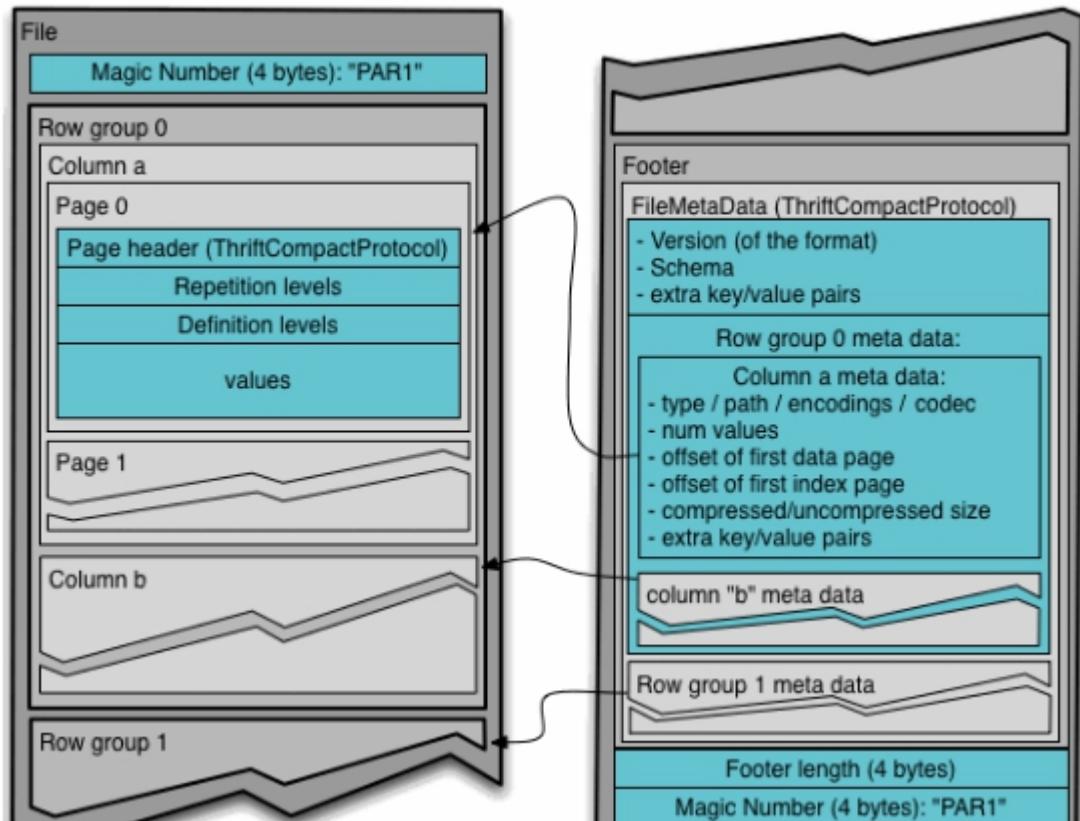


### PARQUET格式

Parquet是面向分析型业务的列式存储格式，由Twitter和Cloudera合作开发，

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。



## 9. 文件存储格式与数据压缩结合

### 9.1 压缩比和查询速度对比

#### 1) TextFile

(1) 创建表，存储数据格式为TEXTFILE



```
1  create table log_text (
2    track_time string,
3    url string,
4    session_id string,
5    referer string,
6    ip string,
7    end_user_id string,
8    city_id string
9  )
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
11 STORED AS TEXTFILE ;
```

## (2) 向表中加载数据

```
1 load data local inpath '/export/servers/hivedatas/log.data' into table
log_text ;
```

## (3) 查看表中数据大小

```
1 dfs -du -h /user/hive/warehouse/myhive.db/log_text;
```

## 2) ORC

### (1) 创建表，存储数据格式为ORC

```
1  create table log_orc(
2    track_time string,
3    url string,
4    session_id string,
5    referer string,
6    ip string,
7    end_user_id string,
8    city_id string
9  )
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
11 STORED AS orc ;
```

### (2) 向表中加载数据

```
1 insert into table log_orc select * from log_text ;
```

### (3) 查看表中数据大小

```
1 dfs -du -h /user/hive/warehouse/myhive.db/log_orc;
```

## 3) Parquet

### (1) 创建表，存储数据格式为parquet

```
1 create table log_parquet(
2   track_time string,
3   url string,
4   session_id string,
5   referer string,
6   ip string,
7   end_user_id string,
8   city_id string
9 )
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
11 STORED AS PARQUET ;
```

### (2) 向表中加载数据

```
1 insert into table log_parquet select * from log_text ;
```

### (3) 查看表中数据大小

```
1 dfs -du -h /user/hive/warehouse/myhive.db/log_parquet;
```

## 存储文件的压缩比总结：

ORC > Parquet > textFile

## 4) 存储文件的查询速度测试：

### 1) TextFile

```
hive (default)> select count(*) from log_text;
```

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

### 2) ORC



```
hive (default)> select count(*) from log_orc;
```

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

### 3) Parquet

```
hive (default)> select count(*) from log_parquet;
```

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

## 存储文件的查询速度总结：

ORC > TextFile > Parquet

## 9.2 ORC存储指定压缩方式

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 $\geq 1000$ )                   |
| 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



```
1  create table log_orc_none(
2    track_time string,
3    url string,
4    session_id string,
5    referer string,
6    ip string,
7    end_user_id string,
8    city_id string
9  )
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
11 STORED AS orc tblproperties ("orc.compress"="NONE");
```

## (2) 插入数据

```
1  insert into table log_orc_none select * from log_text ;
```

## (3) 查看插入后数据

```
1  dfs -du -h /user/hive/warehouse/myhive.db/log_orc_none;
```

## 2) 创建一个SNAPPY压缩的ORC存储方式

### (1) 建表语句

```
1  create table log_orc_snappy(
2    track_time string,
3    url string,
4    session_id string,
5    referer string,
6    ip string,
7    end_user_id string,
8    city_id string
9  )
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
11 STORED AS orc tblproperties ("orc.compress"="SNAPPY");
```

### (2) 插入数据

```
1  insert into table log_orc_snappy select * from log_text ;
```

### (3) 查看插入后数据



```
1  dfs -du -h /user/hive/warehouse/myhive.db/log Orc_snappy ;
```

## 9.3 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。

# 10.hive调优

## 10.1 Fetch抓取

Hive中对某些情况的查询可以不必使用MapReduce计算。例如：SELECT \* FROM score;在这种情况下，Hive可以简单地读取score对应的存储目录下的文件，然后输出查询结果到控制台。通过设置hive.fetch.task.conversion参数，可以控制查询语句是否走MapReduce.

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。

```
1  set hive.fetch.task.conversion=none;
2
3  select * from score;
4  select s_score from score;
5  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。

```
1  set hive.fetch.task.conversion=more;
2
3  select * from score;
4  select s_score from score;
5  select s_score from score limit 3;
```

## 10.2 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。

案例实操：

1) 开启本地模式，并执行查询语句

```
1 set hive.exec.mode.local.auto=true;
2 select * from score cluster by s_id;
```

2) 关闭本地模式，并执行查询语句

```
1 set hive.exec.mode.local.auto=false;
2 select * from score cluster by s_id;
```

## 10.3 MapJoin

如果不指定MapJoin或者不符合MapJoin的条件，那么Hive解析器会在Reduce阶段完成join，容易发生数据倾斜。可以用MapJoin把小表全部加载到内存中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

```
1 set hive.auto.convert.join = true;
```

(2) 大表小表的阈值设置（默认25M以下认为是小表）：

```
1 set hive.mapjoin.smalltable.filesize=25123456;
```

## 10.4 Group By

默认情况下，Map阶段同一Key数据分发给一个reduce，当一个key数据过大时就倾斜了。并不是所有的聚合操作都需要在Reduce端完成，很多聚合操作都可以先在Map端进行部分聚合，最后在Reduce端得出最终结果。



## 开启Map端聚合参数设置

(1) 是否在Map端进行聚合，默认为True

```
1 set hive.map.aggr = true;
```

(2) 在Map端进行聚合操作的条目数目

```
1 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）

```
1 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。

## 10.5 Count(distinct)

数据量小的时候无所谓，数据量大的情况下，由于COUNT DISTINCT操作需要用一个Reduce Task来完成，这一个Reduce需要处理的数据量太大，就会导致整个Job很难完成，一般COUNT DISTINCT使用先GROUP BY再COUNT的方式替换：

```
1 select count(distinct s_id) from score;
2 select count(s_id) from (select id from score group by s_id) a;
```

虽然会多用一个Job来完成，但在数据量大的情况下，这个绝对是值得的。

## 10.6 笛卡尔积

尽量避免笛卡尔积，即避免join的时候不加on条件，或者无效的on条件，Hive只能使用1个reducer来完成笛卡尔积。

## 10.7 动态分区调整



往hive分区表中插入数据时，hive提供了一个动态分区功能，其可以基于查询参数的位置去推断分区的名称，从而建立分区。使用Hive的动态分区，需要进行相应的配置。

Hive的动态分区是以第一个表的分区规则，来对应第二个表的分区规则，将第一个表的所有分区，全部拷贝到第二个表中来，第二个表在加载数据的时候，不需要指定分区了，直接用第一个表的分区即可

### 10.7.1 开启动态分区参数设置

(1) 开启动态分区功能（默认true，开启）

```
1 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。）

```
1 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。

```
1 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。

```
1 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

```
1 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。

```
1 set hive.error.on.empty.partition=false;
```

### 10.7.2 案例操作



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

```
1 create table ori_partitioned(id bigint, time bigint, uid string, keyword
2   string, url_rank int, click_num int, click_url string)
3   PARTITIONED BY (p_time bigint)
4   row format delimited fields terminated by '\t';
5
6   load data local inpath '/export/servers/hivedatas/small_data' into table
7   ori_partitioned partition (p_time='20111230000010');
8
9   load data local inpath '/export/servers/hivedatas/small_data' into table
10  ori_partitioned partition (p_time='20111230000011');
```

#### (2) 创建目标分区表

```
1 create table ori_partitioned_target(id bigint, time bigint, uid string,
2   keyword string, url_rank int, click_num int, click_url string) PARTITIONED
3   BY (p_time STRING) row format delimited fields terminated by '\t'
```

#### (3) 向目标分区表加载数据

如果按照之前介绍的往指定一个分区中Insert数据，那么这个需求很不容易实现。这时候就需要使用动态分区来实现。

```
1 INSERT overwrite TABLE ori_partitioned_target PARTITION (p_time)
2   SELECT id, time, uid, keyword, url_rank, click_num, click_url, p_time
3   FROM ori_partitioned;
```

注意：在SELECT子句的最后几个字段，必须对应前面PARTITION (p\_time)中指定的分区字段，包括顺序。

#### (4) 查看分区

```
1 show partitions ori_partitioned_target;
```

## 10.8 并行执行



Hive会将一个查询转化成一个或者多个阶段。这样的阶段可以是MapReduce阶段、抽样阶段、合并阶段、limit阶段。或者Hive执行过程中可能需要的其他阶段。默认情况下，Hive一次只会执行一个阶段。不过，某个特定的job可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个job的执行时间缩短。不过，如果有更多的阶段可以并行执行，那么job可能就越快完成。

通过设置参数hive.exec.parallel值为true，就可以开启并发执行。不过，在共享集群中，需要注意下，如果job中并行阶段增多，那么集群利用率就会增加。

```
1 set hive.exec.parallel = true;
```

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。

## 10.9 严格模式

Hive提供了一个严格模式，可以防止用户执行那些可能意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。

```
1 set hive.mapred.mode = strict; #开启严格模式
2 set hive.mapred.mode = nonstrict; #开启非严格模式
```

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

## 10.10 JVM重用

JVM重用是Hadoop调优参数的内容，其对Hive的性能具有非常大的影响，特别是对于很难避免小文件的场景或task特别多的场景，这类场景大多数执行时间都很短。



Hadoop的默认配置通常是使用派生JVM来执行map和Reduce任务的。这时JVM的启动过程可能会造成相当大的开销，尤其是执行的job包含有成百上千task任务的情况。JVM重用可以使得JVM实例在同一个job中重新使用N次。N的值可以在Hadoop的mapred-site.xml文件中进行配置。通常在10-20之间，具体多少需要根据具体业务场景测试得出。

我们也可以在hive当中通过

```
1 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。

## 10.11 推测执行

在分布式集群环境下，因为程序Bug（包括Hadoop本身的bug），负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致，有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。为了避免这种情况发生，Hadoop采用了推测执行（Speculative Execution）机制，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：

```
1 set mapred.map.tasks.speculative.execution=true
2 set mapred.reduce.tasks.speculative.execution=true
3 set hive.mapred.reduce.tasks.speculative.execution=true;
```

关于调优这些推测执行变量，还很难给一个具体的建议。如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的map或者Reduce task的话，那么启动推测执行造成的浪费是非常巨大大。