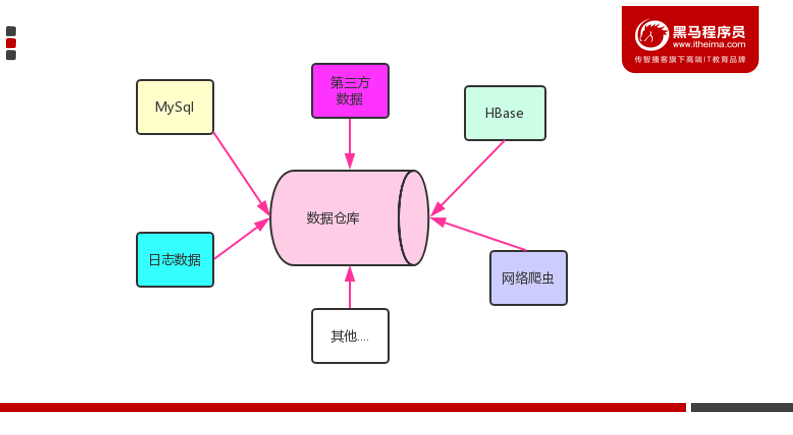
## 数据仓库的介绍

### 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。数据仓库顾名思义，是一个很大的数据存储集合，出于企业的分析性报告和决策支持目的而创建，对多样的业务数据进行筛选与整合。它为企业提供一定的BI（商业智能）能力，指导业务流程改进、监视时间、成本、质量以及控制。

数据仓库的输入方是各种各样的数据源，最终的输出用于企业的数据分析、数据挖掘、数据报表等方向。



### 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）、稳定的（Non-Volatile）和时变的（Time-Variant ）数据集合，用以支持管理决策。

**1.主题性**

不同于传统数据库对应于某一个或多个项目，数据仓库根据使用者实际需求，将不同数据源的数据在一个较高的抽象层次上做整合，所有数据都围绕某一主题来组织。

这里的主题怎么来理解呢？比如对于城市，“天气湿度分析”就是一个主题，对于淘宝，“用户点击行为分析”就是一个主题。

**2.集成性**

数据仓库中存储的数据是来源于多个数据源的集成，原始数据来自不同的数据源，存储方式各不相同。要整合成为最终的数据集合，需要从数据源经过一系列抽取、清洗、转换的过程。

**3.稳定性**

数据仓库中保存的数据是一系列历史快照，不允许被修改。用户只能通过分析工具进行查询和分析。这里说明一点，数据仓库基本上是不许允许用户进行修改，删除操作的。大多数的场景是用来查询分析数据。

**4.时变性**

数据仓库会定期接收新的集成数据，反应出最新的数据变化。这和稳定特点并不矛盾。

### 数据仓库与数据库区别

#### 数据库

数据库是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，也被称为联机事务处理 OLTP（On-Line Transaction Processing）。

#### 数据仓库

数据仓库一般针对某些主题的历史数据进行分析，支持管理决策，又被称为联机分析处理 OLAP（On-Line Analytical Processing）。

首先要明白，数据仓库的出现，并不是要取代数据库。

#### 两者区别

* 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。

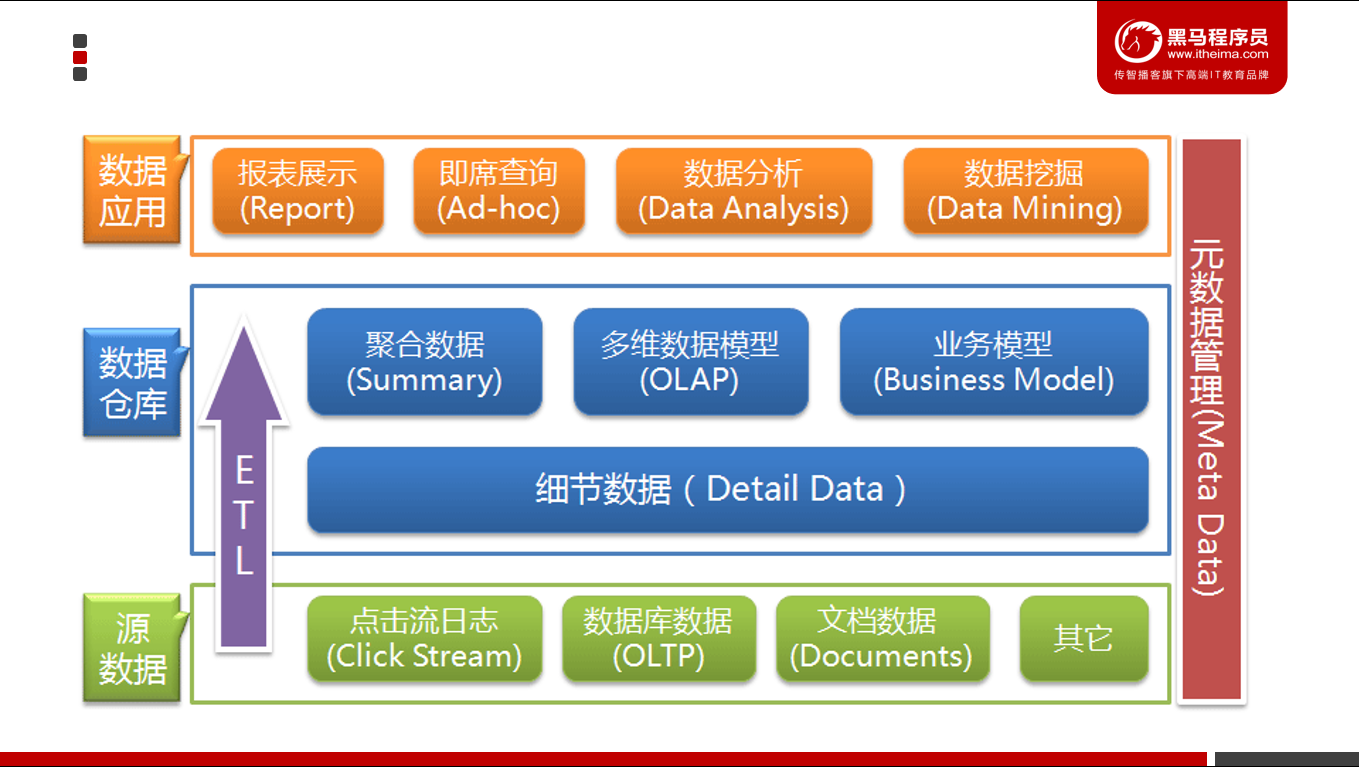
数据仓库是分析系统的数据平台，它从事务系统获取数据，并做汇总、加工，为决策者提供决策的依据。比如，某银行某分行一个月发生多少交易，该分行当前存款余额是多少。如果存款又多，消费交易又多，那么该地区就有必要设立ATM了。

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”。

### 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。**



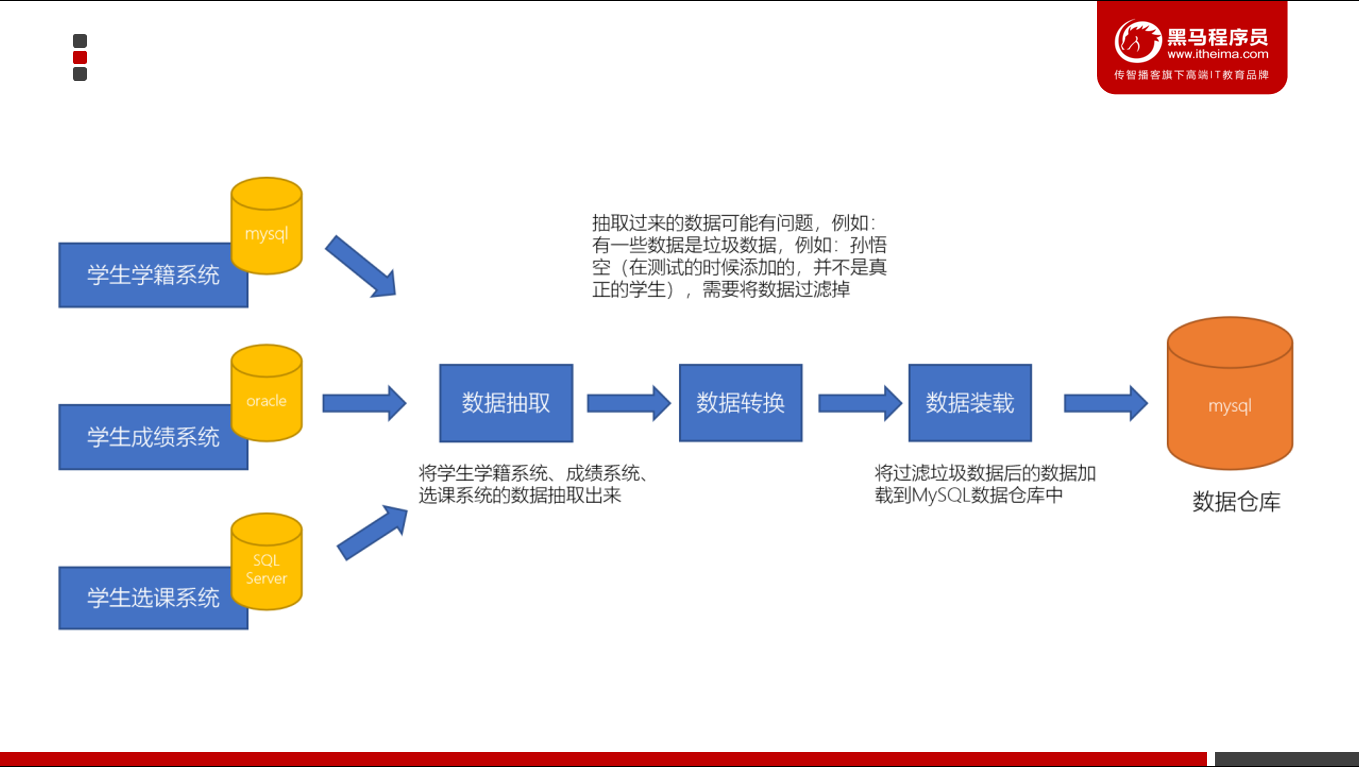
数据仓库的数据来源于不同的源数据，并提供多样的数据应用，数据自下而上流入数据仓库后向上层开放应用，而数据仓库只是中间集成化数据管理的一个平台。

* **源数据层（ODS）:** 操作性数据(Operational Data Store) ，是作为数据库到数据仓库的一种过渡，ODS的数据结构一般与数据来源保持一致，而且ODS的数据周期一般比较短。ODS的数据为后一步的数据处理做准备。
* **数据仓库层（DW）：**数据仓库(Data Warehouse)，是数据的归宿，这里保持这所有的从ODS到来的数据，并长期报错，而且这些数据不会被修改,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）：**数据应用(Data[application](https://www.baidu.com/link?url=plA-knPJLjt9bk3FO7E-okaPeM8Q9YR_gnlNDQQmxm3kNIrizg_Jej4Mzd1XDzLrNTzZfDl8xel5yMJ8ElInUcdl2ut9LaFjpuBbdUpkSUu&wd=&eqid=98d9dd3500017950000000025eeb7392)),为了特定的应用目的或应用范围，而从数据仓库中独立出来的一部分数据，也可称为部门数据或主题数据，该数据面向应用。如根据报表、专题分析需求而计算生成的数据。

### 数据仓库之ETL

ETL，是英文Extract-Transform-Load的缩写，用来描述将数据从来源端经过抽取（extract）、转换（transform）、加载（load）至目的端的过程。ETL是将业务系统的数据经过抽取、清洗、转换之后加载到数据仓库的过程，目的是将企业中分散、零乱、标准不统一的数据整合到一起。

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。



## Hive基本概念

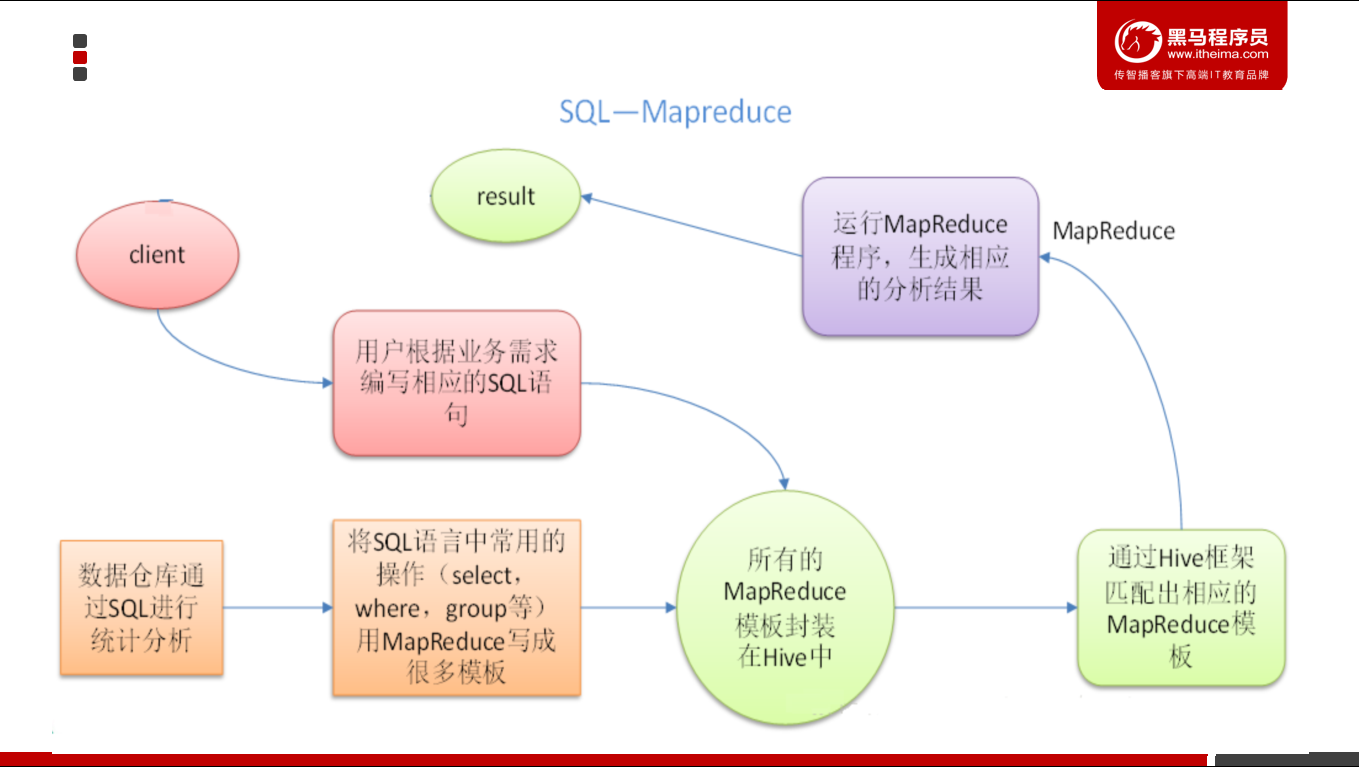
### Hive介绍

#### 什么是Hive

Hive是一个构建在Hadoop上的数据仓库框架。最初，Hive是由Facebook开发，后来移交由Apache软件基金会开发，并作为一个Apache开源项目。

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

* **直接使用hadoop所面临的问题**

1. 人员学习成本太高
2. 项目周期要求太短
3. MapReduce实现复杂查询逻辑开发难度太大

* **为什么要使用Hive**

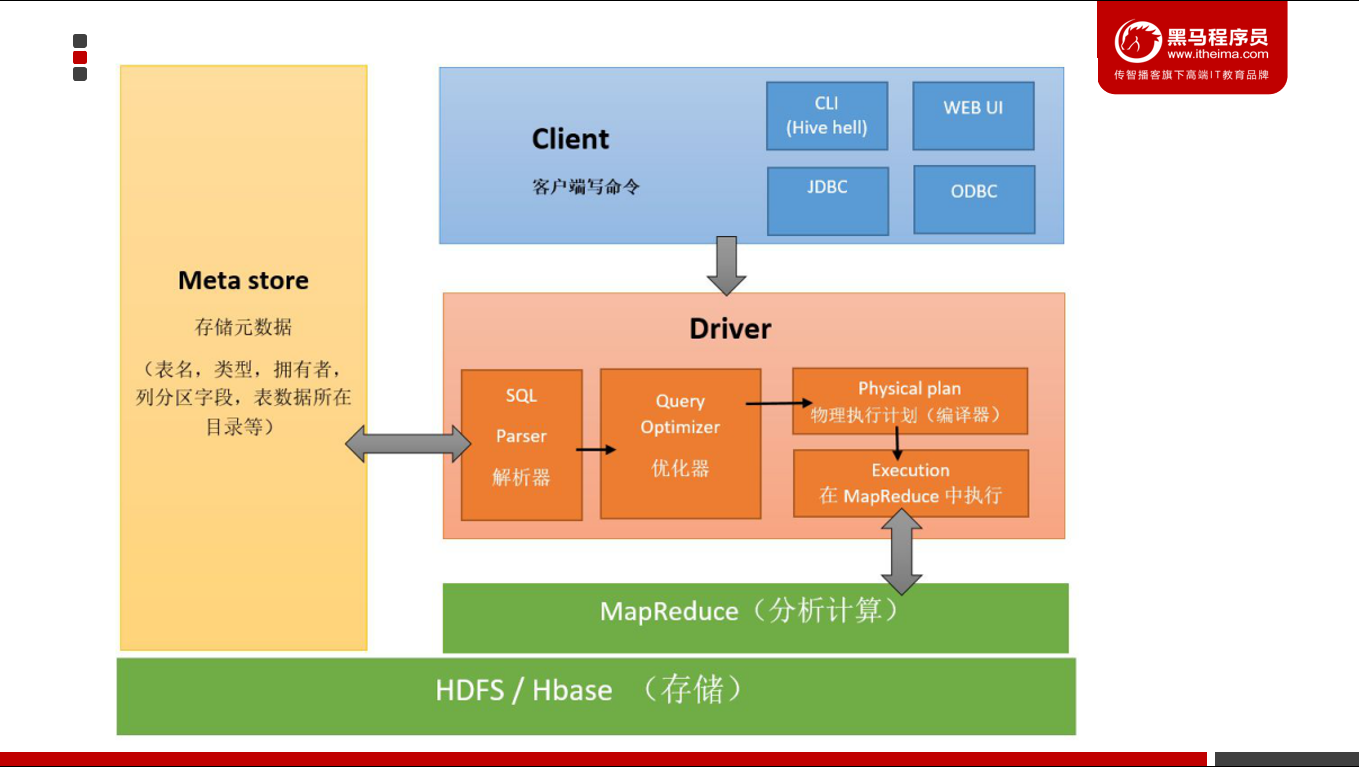
1. 操作接口采用类SQL语法，提供快速开发的能力
2. 避免了去写MapReduce，减少开发人员的学习成本
3. 功能扩展很方便

#### Hive的特点

1. Hive最大的特点是通过类SQL来分析大数据，而避免了写MapReduce程序来分析数据，这样使得分析数据更容易。
2. 数据是存储在HDFS上的，Hive本身并不提供数据的存储功能，它可以使已经存储的数据结构化。
3. Hive是将数据映射成数据库和一张张的表，库和表的元数据信息一般存在关系型数据库上（比如MySQL）。
4. 数据存储方面：它能够存储很大的数据集，可以直接访问存储在Apache HDFS或其他数据存储系统（如Apache HBase）中的文件。
5. 数据处理方面：因为Hive语句最终会生成MapReduce任务去计算，所以不适用于实时计算的场景，它适用于离线分析。
6. Hive除了支持MapReduce计算引擎，还支持Spark和Tez这两种分布式计算引擎；
7. 数据的存储格式有多种，比如数据源是二进制格式，普通文本格式等等；

### Hive架构

#### 架构图



#### 基本组成

**客户端:**Client CLI(hive shell 命令行),JDBC/ODBC(java访问hive),WEBUI(浏览器访问hive)

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

默认存储在自带的derby数据库中,推荐使用MySQL存储Metastore

**驱动器:Driver**

(1)解析器(SQL Parser):将SQL字符转换成抽象语法树AST,这一步一般使用都是第三方工具库完成,比如antlr,对AST进行语法分析,比如表是否存在,字段是否存在,SQL语句是否有误

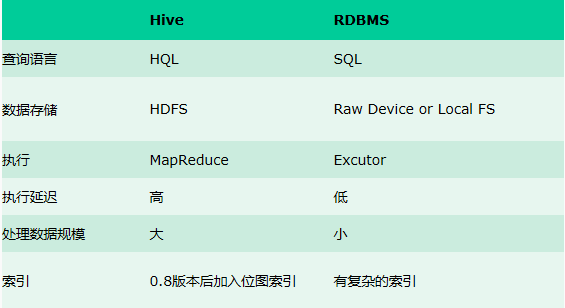
(2)编译器(Physical Plan):将AST编译生成逻辑执行计划

(3)优化器(Query Optimizer):对逻辑执行计划进行优化

(4)执行器(Execution):把逻辑执行计划转换成可以运行的物理计划,对于Hive来说,就是MR/Spark

**存储和执行：**Hive使用HDFS进行存储,使用MapReduce进行计算

#### Hive与传统数据库对比



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

### Hive的元数据

**元数据(metadata)：**本质上只是用来存储hive中有哪些数据库，哪些表，表的字段，分区，索引以及命名空间等元信息。元数据存储在关系型数据库中。如hive内置的Derby、第三方数据库如MySQL等。

**元数据服务(metastore）**，作用是：客户端连接metastore服务，metastore再去连接MySQL数据库来存取元数据。有了metastore服务，就可以有多个客户端同时连接，而且这些客户端不需要知道MySQL数据库的用户名和密码，只需要连接metastore 服务即可。

## Hive的安装

### Hive的安装方式

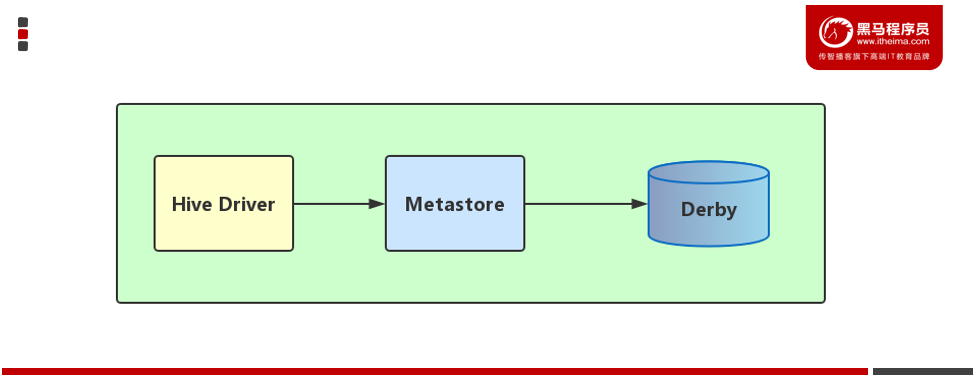
hive的安装一共有三种方式**:内嵌模式、本地模式、远程模式**

#### 内嵌模式

**内嵌模式**使用的是内嵌的Derby数据库来存储元数据，也不需要额外起Metastore服务。数据库和Metastore服务都嵌入在主Hive Server进程中。这个是默认的，配置简单，但是一次只能一个客户端连接，适用于用来实验，不适用于生产环境。

解压hive安装包 bin/hive 启动即可使用

缺点：不同路径启动hive，每一个hive拥有一套自己的元数据，无法共享。



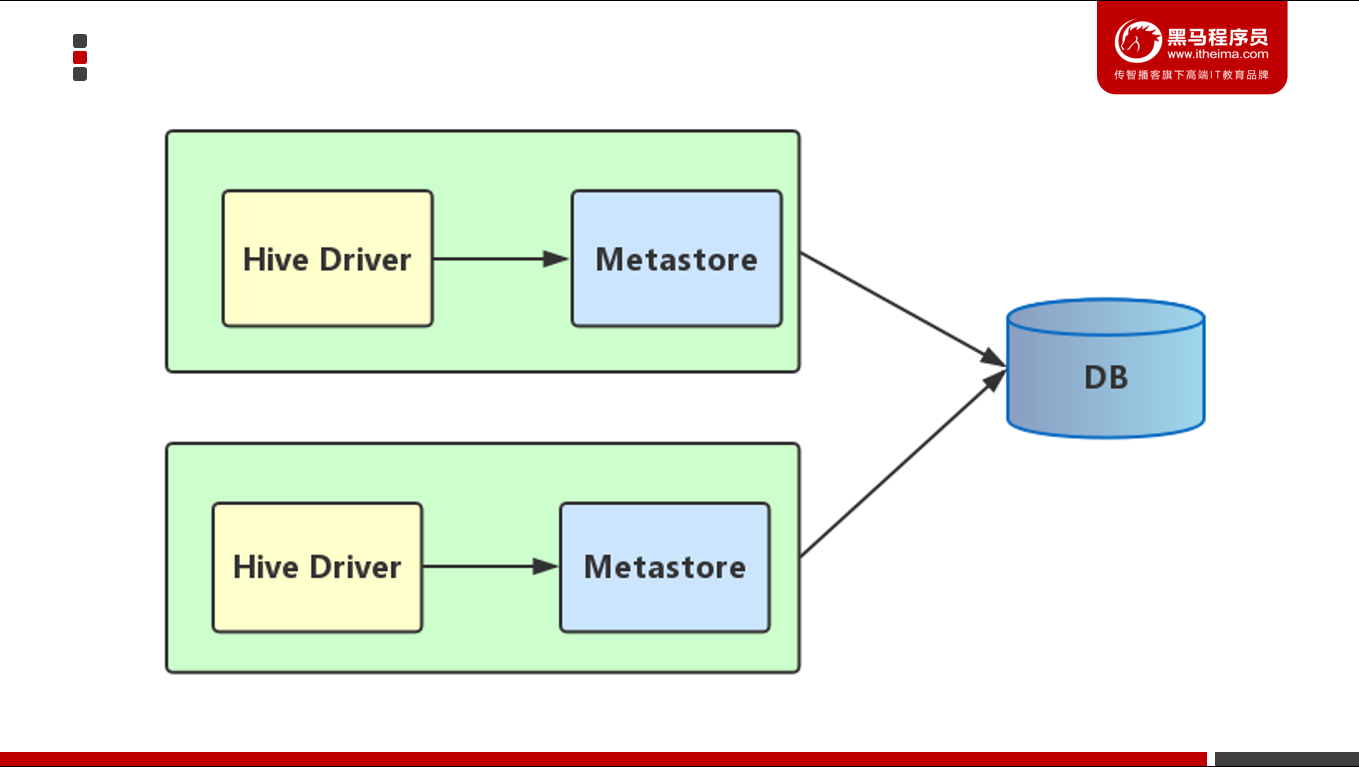
#### 本地模式

**本地模式**采用外部数据库来存储元数据，目前支持的数据库有：MySQL、Postgres、Oracle、MS SQL Server.在这里我们使用MySQL。

本地模式不需要单独起metastore服务，用的是跟hive在同一个进程里的metastore服务。也就是说当你启动一个hive 服务，里面默认会帮我们启动一个metastore服务。

hive根据hive.metastore.uris 参数值来判断，如果为空，则为本地模式。

缺点是：每启动一次hive服务，都内置启动了一个metastore。

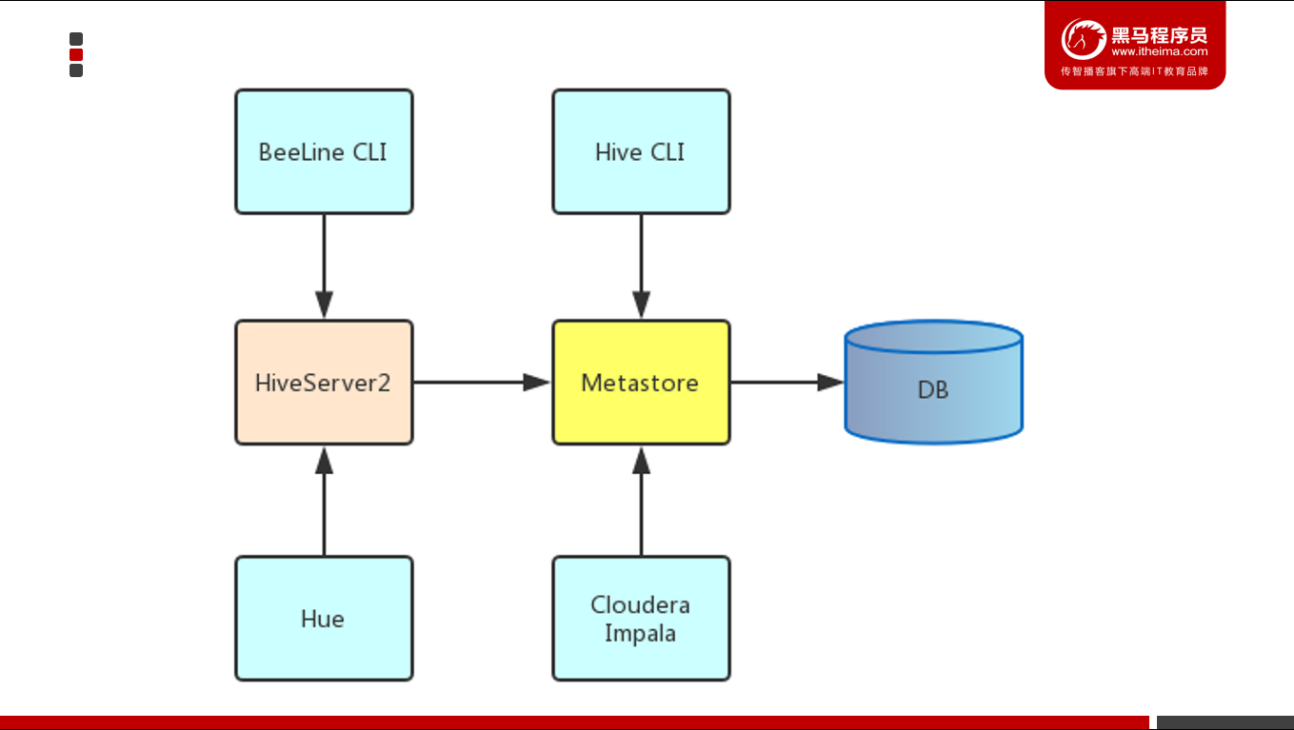


#### 远程模式

**远程模式**下，需要单独起metastore服务，然后每个客户端都在配置文件里配置连接到该metastore服务。远程模式的metastore服务和hive运行在不同的进程里。

在生产环境中，建议用远程模式来配置Hive Metastore。

在这种情况下，其他依赖hive的软件都可以通过Metastore访问hive。



远程模式下，需要配置hive.metastore.uris 参数来指定metastore服务运行的机器ip和端口，并且需要单独手动启动metastore服务。

### Hive的安装

我们在此处选择第三台机器node3作为我们hive的安装机器，安装方式使用远程方式。

#### 准备工作

1、下载hive的安装包，这里我们选用hive的版本是2.1.1,软件包为：apache-hive-2.1.0-bin.tar.gz

Hive下载地址:<http://archive.apache.org/dist/hive/>

2、下载mysql的安装包，我们使用的mysql版本是5.7.29,软件包为:mysql-5.7.29-linux-glibc2.12-x86\_64.tar.gz

下载地址:<https://downloads.mysql.com/archives/community/>

3、将apache-hive-2.1.0-bin.tar.gz上传到/export/software目录

1. 将mysql-5.7.29-linux-glibc2.12-x86\_64.tar.gz上传到/export/software目录

#### 安装mysql数据库

在这里，我们使用mysql数据库作为Hive的元数据存储，所以在安装Hive之前，必须安装好mysql

1. **解压mysql安装包**

|  |
| --- |
| **cd** **/**export**/**software  **tar** **-**zxvf mysql-5.7.29-linux-glibc2.12-x86\_64.tar.gz -C **/export/server/** |

1. **重命名**

|  |
| --- |
| **cd** **/export/server**  **mv** mysql-5.7.29-linux-glibc2.12-x86\_64 mysql-5.7.29 |

**3、添加用户组与用户**

|  |
| --- |
| groupadd mysql  **useradd** -r -g mysql mysql |

**4、修改目录权限**

|  |
| --- |
| **chown** -R mysql**:**mysql **/export/server/**mysql-5.7.29**/** |

**5、配置mysql服务**

|  |
| --- |
| **cp** **/export/server/**mysql-5.7.29**/**support-files**/**mysql.server **/**etc**/**init.d**/**mysql |

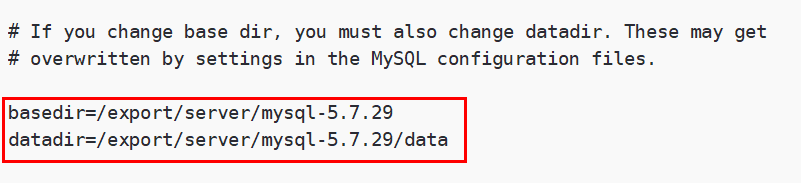
**6、修改mysql服务**

修改/etc/init.d/mysql文件

|  |
| --- |
| **vi** **/**etc**/**init.d**/**mysql |

将该文件的basedir和datadir路径修改为以下内容

|  |
| --- |
| basedir**=/export/server/**mysql-5.7.29  datadir**=/export/server/**mysql-5.7.29**/**data |



1. **修改配置文件my.cnf**

修改/etc/my.cnf文件

|  |
| --- |
| **vi** **/**etc**/**my.cnf |

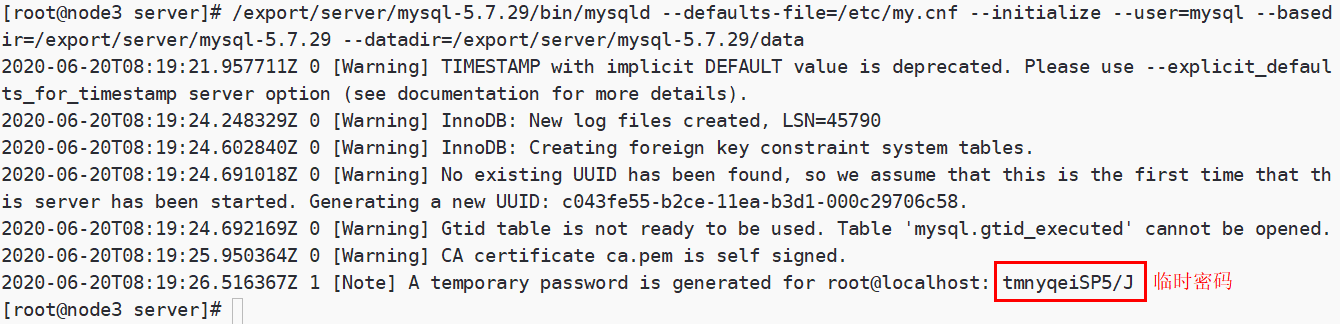
将/etc/my.cnf文件修改为以下内容

|  |
| --- |
| **[**client**]**  port**=**3306  default-character-set**=**utf8  **[**mysqld**]**  basedir**=/export/server/**mysql-5.7.29  datadir**=/export/server/**mysql-5.7.29**/**data  port**=**3306  character-set-server**=**utf8  default\_storage\_engine**=**InnoDB |

1. **初始化mysql**

|  |
| --- |
| **/export/server/**mysql-5.7.29**/**bin**/**mysqld **--**defaults-file**=/**etc**/**my.cnf **--**initialize **--**user**=**mysql **--**basedir**=/export/server/**mysql-5.7.29 **--**datadir**=/export/server/**mysql-5.7.29**/**data |

执行该命令之后，会生成一个mysql的临时密码,这个密码后边要使用。



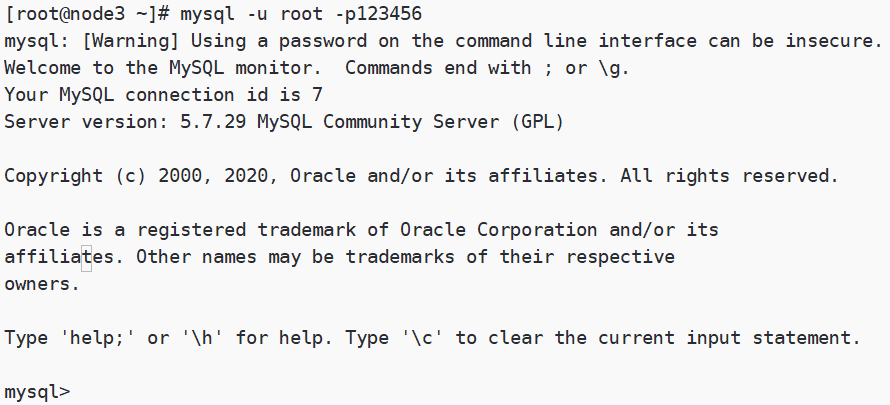
1. **启动服务**

|  |
| --- |
| **service** mysql start |

1. **登录mysql**

使用第7步生成的临时密码

|  |
| --- |
| mysql -uroot -p临时密码 |



1. **修改密码**

注意这条命令是在登录mysql之后执行

|  |
| --- |
| **set** password**=**password**(**'123456'**);** |

1. **开启远程访问权限**

注意这条命令是在登录mysql之后执行

|  |
| --- |
| GRANT ALL PRIVILEGES ON **\*.\*** TO 'root'**@**'%' IDENTIFIED BY '123456'**;**  flush privileges**;** |

1. **修改环境变量**

退出mysql,然后修改Linux的/etc/profile文件

|  |
| --- |
| **vi** **/**etc**/**profile |

在该文件末尾最后添加以下内容

|  |
| --- |
| **export** MYSQL\_HOME**=/export/server/**mysql-5.7.29  **export** PATH**=$PATH:$MYSQL\_HOME/**bin |

保存修改之后，让该文件的修改生效

|  |
| --- |
| source **/**etc**/**profile |

1. **将mysql设置为开机启动**

|  |
| --- |
| chkconfig **--**add mysql #mysql服务到自启服务  chkconfig mysql on #设置自启 |

#### 安装Hive

1. **解压Hive安装包并重命名**

|  |
| --- |
| cd **/**export**/**software  tar **-**zxvf apache**-**hive**-**2.1.0**-**bin**.**tar**.**gz **-**C **/export/server**  cd **/export/server**  mv apache**-**hive**-**2.1.0**-**bin hive**-**2.1.0 |

##### 1、修改hive的配置文件

**hive-env.sh**

添加我们的hadoop的环境变量

|  |
| --- |
| **cd** **/export/server/**hive-2.1.0**/**conf  **cp** hive-env.sh.template hive-env.sh  **vim** hive-env.sh |

修改内容如下：

|  |
| --- |
| HADOOP\_HOME**=/export/server/**hadoop-2.7.5  **export** HIVE\_CONF\_DIR**=/export/server/**hive-2.1.0**/**conf |

##### 2、修改hive-site.xml

|  |
| --- |
| **cd** **/export/server/**hive-2.1.0**/**conf  **vim** hive-site.xml |

在该文件中添加以下内容

|  |
| --- |
| <?xml version=**"1.0"** encoding=**"UTF-8"** standalone=**"no"**?>  <?xml-stylesheet type=**"text/xsl"** href=**"configuration.xsl"**?>  <configuration>  <property>  <name>**javax.jdo.option.ConnectionUserName**</name>  <value>**root**</value>  </property>  <property>  <name>**javax.jdo.option.ConnectionPassword**</name>  <value>**123456**</value>  </property>  <property>  <name>**javax.jdo.option.ConnectionURL**</name>  <value>**jdbc:mysql://node3:3306/hive?createDatabaseIfNotExist=true&amp;useSSL=false**</value>  </property>  <property>  <name>**javax.jdo.option.ConnectionDriverName**</name>  <value>**com.mysql.jdbc.Driver**</value>  </property>  <property>  <name>**hive.metastore.schema.verification**</name>  <value>**false**</value>  </property>  <property>  <name>**datanucleus.schema.autoCreateAll**</name>  <value>**true**</value>  </property>  <property>  <name>**hive.server2.thrift.bind.host**</name>  <value>**node3**</value>  </property>  </configuration> |

##### 3、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/server/**hive-2.1.0**/**lib |

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

##### 4、拷贝相关jar包

将hive-2.1.0/jdbc/目录下的hive-jdbc-2.1.0-standalone.jar 拷贝到hive-2.1.0/lib/目录

|  |
| --- |
| cp **/export/server/**hive-2.1.0**/**jdbc**/**hive**-**jdbc-2.1.0**-**standalone**.**jar **/export/server/**hive-2.1.0**/**lib**/** |

##### 5、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

|  |
| --- |
| **vim** **/**etc**/**profile |

添加以下内容:

|  |
| --- |
| **export** HIVE\_HOME**=/export/server/**hive-2.1.0  **export** PATH**=:$HIVE\_HOME/**bin**:$PATH** |

### Hive的交互方式

##### 第一种交互方式：bin/hive

|  |
| --- |
| **cd** **/export/server/**hive-2.1.0**/**  bin**/**hive |

创建一个数据库

|  |
| --- |
| **create** database mytest**;**  show databases**;** |

此处需要注意: 如果启动后在mysql中没有发现构建hive库及其相关的表, 建议执行一下操作:

|  |
| --- |
| schematool -dbType mysql -initSchema  手动初始化元数据信息 |

##### 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

|  |
| --- |
| cd **/export/server/**hive-2.1.0**/**  **bin/**hive **-**e "create database 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

|  |
| --- |
| cd **/export/server**  vim hive**.sql** |

脚本内容如下:

|  |
| --- |
| **create** database mytest2**;**  **use** mytest2**;**  **create** **table** stu**(**id **int,**name string**);** |

通过hive -f 来执行我们的sql脚本

|  |
| --- |
| **bin/**hive **-**f **/export/server/**hive**.sql** |

##### 第三种交互方式：Beeline Client

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

在hive运行的服务器上，首先启动metastore服务，然后启动hiveserver2服务。

|  |
| --- |
| nohup **/export/server/**hive-2.1.0**/bin/**hive --service metastore &  nohup **/export/server/**hive-2.1.0**/bin/**hive --service hiveserver2 & |

nohup 和 & 表示后台启动

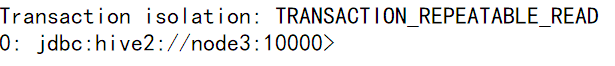
在node3上使用beeline客户端进行连接访问。

|  |
| --- |
| /export/server/hive-2.1.0/bin/beeline |

根据提醒进行以下操作:

|  |
| --- |
| **[root@node3 ~]# /export/server/hive-2.1.0/bin/beeline**  which: no hbase in (:/export/server/hive-2.1.0/bin::/export/server/hadoop-2.7.5/bin:/export/server/hadoop-2.7.5/sbin::/export/server/jdk1.8.0\_241/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/export/server/mysql-5.7.29/bin:/root/bin)  Beeline version 2.1.0 by Apache Hive  **beeline>** !connect jdbc:hive2://node3:10000  Connecting to jdbc:hive2://node3:10000  Enter username for jdbc:hive2://node3:10000: root  Enter password for jdbc:hive2://node3:10000:123456 |

连接成功之后，出现以下内容，可以在提示符后边输入hive sql命令



注意: 如果报出以下, 请修改 hadoop中 core-site.xml文件

错误信息为: User: root is not allowed to impersonate root

解决方案: 在node1的 hadoop的 core-site.xml文件中添加一下内容:

|  |
| --- |
| <property>  <name>hadoop.proxyuser.root.hosts</name>  <value>\*</value>  </property>  <property>  <name>hadoop.proxyuser.root.groups</name>  <value>\*</value>  </property> |

添加后, 将 core-site.xml 发送到其他两台机子:

|  |
| --- |
| cd /export/servers/hadoop-2.7.5/etc/hadoop  scp core-site.xml node2:$PWD  scp core-site.xml node3:$PWD |

然后重启hadoop即可

### Hive一键启动脚本

这里，我们写一个expect脚本，可以一键启动beenline，并登录到hive。expect是建立在tcl基础上的一个自动化交互套件, 在一些需要交互输入指令的场景下, 可通过脚本设置自动进行交互通信。

#### 安装expect

|  |
| --- |
| **yum** install expect |

#### 创建脚本

|  |
| --- |
| **cd** **/export/server/**hive-2.1.0/bin  **vim** beenline.exp |

添加以下内容:

|  |
| --- |
| #!/bin/expect  spawn beeline  **set** timeout 5  expect "beeline>"  send "!connect jdbc:hive2://node3:10000\r"  expect "Enter username for jdbc:hive2://node3:10000:"  send "root\r"  expect "Enter password for jdbc:hive2://node3:10000:"  send "123456\r"  interact |

#### 修改脚本权限

|  |
| --- |
| **chmod** 777 beenline.exp |

#### 启动beeline

|  |
| --- |
| expect beenline.exp |

## Hive数据库和表操作

### 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>** |

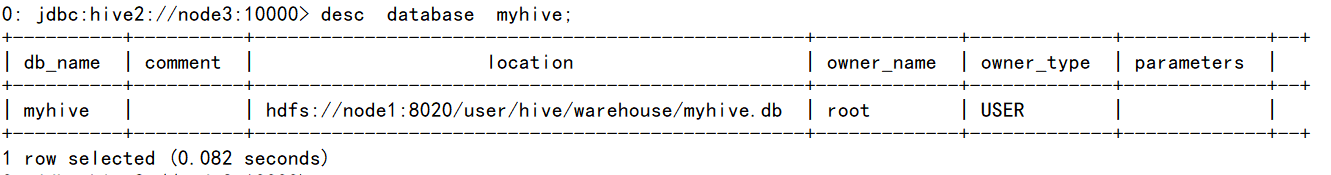
#### 创建数据库并指定hdfs存储位置

|  |
| --- |
| **create** database myhive2 location '/myhive2'**;** |

#### 查看数据库详细信息

查看数据库基本信息

|  |
| --- |
| **desc** database myhive**;** |



#### 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 myhive**;** |

强制删除数据库，包含数据库下面的表一起删除

|  |
| --- |
| **drop** database myhive2 **cascade;** |

### 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 可用来指定行分隔符
5. STORED AS SEQUENCEFILE|TEXTFILE|RCFILE 来指定该表数据的存储格式，hive中，表的默认存储格式为TextFile。
6. CLUSTERED BY

对于每一个表（table）进行分桶(MapReuce中的分区），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。

7、LOCATION: 指定表在HDFS上的存储位置。

#### Hive建表时候的字段类型

| **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 -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，毫秒值精度 | 122327493795 |
|  | DATE | 日期 | ‘2016-03-29’ |
|  | [INTERVAL](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types#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) |

#### 内部表操作

未被external修饰的是内部表（managed table）,内部表又称管理表,内部表数据存储的位置由hive.metastore.warehouse.dir参数决定（默认：/user/hive/warehouse），删除内部表会直接删除元数据（metadata）及存储数据，因此内部表不适合和其他工具共享数据。

##### hive建表初体验

|  |
| --- |
| **create** database myhive**;**  **use** myhive**;**  **create** **table** stu**(**id **int,**name string**);**  **insert** **into** stu **values** **(**1**,**"zhangsan"**);**  **select** **\*** **from** stu**;** |

##### 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/stu2'**;** |

##### 根据查询结果创建表

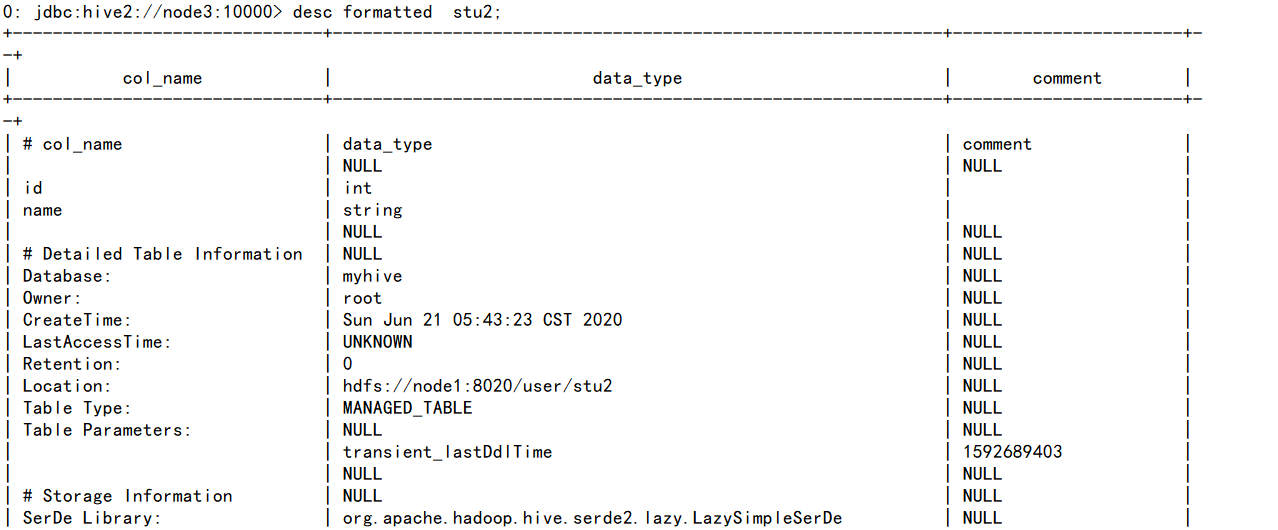
|  |
| --- |
| **create** **table** stu3 **as** **select** **\*** **from** stu2**;** |

##### 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**;** |



##### 删除表

|  |
| --- |
| **drop** **table** stu2**;** |

查看数据库和HDFS，发现删除内部表之后，所有的内容全部删除

#### 外部表操作

在创建表的时候可以指定external关键字创建外部表,外部表对应的文件存储在location指定的hdfs目录下,向该目录添加新文件的同时，该表也会读取到该文件(当然文件格式必须跟表定义的一致)。

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外部表的时候，数据仍然存放在hdfs当中，不会删掉。

##### 数据装载载命令Load

Load命令用于将外部数据加载到Hive表中

**语法:**

|  |
| --- |
| load **data** **[local]** inpath '/export/server/datas/student.txt' overwrite **|** **into** **table** student **[partition** **(**partcol1**=**val1**,**…**)];** |

**参数:**

1. load data:表示加载数据
2. local:表示从本地加载数据到hive表；否则从HDFS加载数据到hive表
3. inpath:表示加载数据的路径
4. overwrite:表示覆盖表中已有数据，否则表示追加
5. into table:表示加载到哪张表
6. student:表示具体的表
7. partition:表示上传到指定分区

##### 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

源数据如下:

student.txt

|  |
| --- |
| 01 赵雷 1990-01-01 男  02 钱电 1990-12-21 男  03 孙风 1990-05-20 男  04 李云 1990-08-06 男  05 周梅 1991-12-01 女  06 吴兰 1992-03-01 女  07 郑竹 1989-07-01 女  08 王菊 1990-01-20 女 |

teacher.txt

|  |
| --- |
| 01 张三  02 李四  03 王五 |

###### 1、创建老师表：

|  |
| --- |
| **create** **external** **table** teacher **(**t\_id string**,**t\_name string**)** **row** **format** delimited fields terminated **by** '\t'**;** |

###### 2、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'**;** |

###### 3、从本地文件系统向表中加载数据

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/student.txt' **into** **table** student**;** |

###### 4、加载数据并覆盖已有数据

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/student.txt' overwrite **into** **table** student**;** |

###### 5、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/server/**hivedatas  hadoop fs **-**mkdir **-**p **/**hivedatas  hadoop fs **-**put teacher**.**txt **/**hivedatas**/**  load **data** inpath '/hivedatas/teacher.txt' **into** **table** teacher**;** |

注意,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

#### 复杂类型操作

##### Array类型

**源数据:**

说明:name与locations之间制表符分隔，locations中元素之间逗号分隔

|  |
| --- |
| zhangsan beijing,shanghai,tianjin,hangzhou  wangwu changchun,chengdu,wuhan,beijin |

**建表语句**

|  |
| --- |
| **create** **table** hive\_array**(**name string**,** work\_locations **array<**string**>)**  **row** **format** delimited fields terminated **by** '\t'  COLLECTION ITEMS TERMINATED **BY** ','**;** |

**导入数据**（从本地导入，同样支持从HDFS导入）

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/work\_locations.txt' overwrite **into** **table** hive\_array**;** |

**常用查询：**

|  |
| --- |
| -- 查询所有数据  **select** **\*** **from** hive\_array**;**  -- 查询loction数组中第一个元素  **select** name**,** work\_locations**[**0**]** location **from** hive\_array**;**  -- 查询location数组中元素的个数  **select** name**,** **size(**work\_locations**)** location **from** hive\_array**;**  -- 查询location数组中包含tianjin的信息  **select** **\*** **from** hive\_array **where** array\_contains**(**work\_locations**,**'tianjin'**);** |

##### map类型

**源数据:**

说明：字段与字段分隔符: “,”；需要map字段之间的分隔符："#"；map内部k-v分隔符：":"

|  |
| --- |
| 1,zhangsan,father:xiaoming#mother:xiaohuang#brother:xiaoxu,28  2,lisi,father:mayun#mother:huangyi#brother:guanyu,22  3,wangwu,father:wangjianlin#mother:ruhua#sister:jingtian,29  4,mayun,father:mayongzhen#mother:angelababy,26 |

**建表语句**

|  |
| --- |
| **create** **table** hive\_map**(**  id **int,** name string**,** members **map<**string**,**string**>,** age **int**  **)**  **row** **format** delimited  fields terminated **by** ','  COLLECTION ITEMS TERMINATED **BY** '#'  **MAP** KEYS TERMINATED **BY** ':'**;** |

**导入数据**

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/hive\_map.txt' overwrite **into** **table** hive\_map**;** |

**常用查询**

|  |
| --- |
| **select** **\*** **from** hive\_map**;**  **select** id**,** name**,** members**[**'father'**]** father**,** members**[**'mother'**]** mother**,** age **from** hive\_map**;**  **select** id**,** name**,** map\_keys**(**members**)** **as** relation **from** hive\_map**;**  **select** id**,** name**,** map\_values**(**members**)** **as** relation **from** hive\_map**;**  **select** id**,**name**,size(**members**)** num **from** hive\_map**;**  **select** **\*** **from** hive\_map **where** array\_contains**(**map\_keys**(**members**),** 'brother'**);**  **select** id**,**name**,** members**[**'brother'**]** brother **from** hive\_map **where** array\_contains**(**map\_keys**(**members**),** 'brother'**);** |

##### struct类型

**源数据：**

说明：字段之间#分割，第二个字段之间冒号分割

|  |
| --- |
| 192.168.1.1#zhangsan**:**40  192.168.1.2#lisi**:**50  192.168.1.3#wangwu**:**60  192.168.1.4#zhaoliu**:**70 |

**建表语句**

|  |
| --- |
| **create** **table** hive\_struct**(**  ip string**,** info struct**<**name**:**string**,** age**:int>**  **)**  **row** **format** delimited  fields terminated **by** '#'  COLLECTION ITEMS TERMINATED **BY** ':'**;** |

**导入数据**

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/hive\_struct.txt' **into** **table** hive\_struct**;** |

**常用查询**

|  |
| --- |
| **select** **\*** **from** hive\_struct**;**  **select** ip**,** info**.**name **from** hive\_struct**;** |

#### 内部表和外部表之间的转换

1. 查询表的类型

|  |
| --- |
| **desc** formatted student**;**  Table Type: MANAGED\_TABLE |

1. 修改内部表student为外部表

|  |
| --- |
| **alter** **table** student **set** tblproperties**(**'EXTERNAL'**=**'TRUE'**);** |

1. 查询表的类型

|  |
| --- |
| **desc** formatted student**;**  Table Type: EXTERNAL\_TABLE |

1. 修改外部表student为内部表

|  |
| --- |
| **alter** **table** student **set** tblproperties**(**'EXTERNAL'**=**'FALSE'**);** |

1. 查询表的类型

|  |
| --- |
| **desc** formatted student**;**  Table Type: MANAGED\_TABLE |

注意：('EXTERNAL'='TRUE')和('EXTERNAL'='FALSE')为固定写法，区分大小写！

#### 分区表：

##### 基本操作

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/server/hivedatas/score.txt' **into** **table** score **partition** **(month=**'202006'**);** |

加载数据到一个多分区的表中去

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/score.txt' **into** **table** score2 **partition(year=**'2020'**,month=**'06'**,day=**'01'**);** |

查看分区

|  |
| --- |
| show partitions score**;** |

添加一个分区

|  |
| --- |
| **alter** **table** score **add** **partition(month=**'202005'**);** |

同时添加多个分区

|  |
| --- |
| **alter** **table** score **add** **partition(month=**'202004'**)** **partition(month** **=** '202003'**);** |

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** **=** '202006'**);** |

##### 外部分区表综合练习

需求描述：现在有一个文件score.txt文件，存放在集群的这个目录下/scoredatas/month=202006，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。

**1、数据准备：**

|  |
| --- |
| hadoop fs **-**mkdir **-**p **/**scoredatas**/month=**202006  hadoop fs **-**put score**.**txt**/**scoredatas**/month=**202006**/** |

**2、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'**;** |

1. **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：

|  |
| --- |
| hadoop fs **-**mkdir **-**p **/**scoredatas**/month=**202005  hadoop fs **-**put score**.**txt**/**scoredatas**/month=**202005 |

修改表，进行手动添加方式

|  |
| --- |
| **alter** **table** score4 **add** **partition(month=**'202005'**);** |

#### 分桶表

##### 基本操作

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，由于桶表的数据加载通过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/server/hivedatas/course.txt' **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  **alter** **table** score5 change **column** mysco mysconew **int;**  -- 5:查询表结构  **desc** score5**;** |

##### 删除表

|  |
| --- |
| **drop** **table** score5**;** |

##### 清空表数据

只能清空管理表，也就是内部表

|  |
| --- |
| **truncate** **table** score6**;** |

#### hive表中加载数据

##### 直接向分区表中插入数据

通过insert into方式加载数据

|  |
| --- |
| **create** **table** score3 **like** score**;**  **insert** **into** **table** score3 **partition(month** **=**'202007'**)** **values** **(**'001'**,**'002'**,**'100'**);** |

通过查询方式加载数据

|  |
| --- |
| **create** **table** score4 **like** score**;**  **insert** overwrite **table** score4 **partition(month** **=** '202006'**)** **select** s\_id**,**c\_id**,**s\_score **from** score**;** |

##### 通过查询插入数据

通过load方式加载数据

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/score.txt' overwrite **into** **table** score **partition(month=**'202006'**);** |

###### 多插入模式

给score表加载数据

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/score.txt' overwrite **into** **table** score **partition(month=**'202006'**);** |

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=**'202006'**)** **select** s\_id**,**c\_id **insert** overwrite **table** score\_second **partition(month** **=** '202006'**)** **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'**;** |

1. 上传数据到hdfs上

|  |
| --- |
| hadoop fs **-**mkdir **-**p **/**myscore6  hadoop fs **-**put score**.**txt**/**myscore6**;** |

1. 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** teacher2 **like** teacher**;**  export **table** teacher **to** '/export/teacher'**;**  import **table** teacher2 **from** '/export/teacher'**;** |

#### 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/server/exporthive' **select** **\*** **from** score**;** |

1. 将查询的结果格式化导出到本地

|  |
| --- |
| **insert** overwrite **local** directory '/export/server/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/server/exporthive' **row** **format** delimited fields terminated **by** '\t' **select** **\*** **from** score**;** |

##### 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/server/**exporthive**/**score**.**txt |

##### export导出到HDFS上

|  |
| --- |
| export **table** score **to** '/export/exporthive/score'**;** |

##### sqoop导出

后续单独讲。

## hive查询语法

### SELECT语句

#### 语句结构

**基本语法:**

|  |
| --- |
| **SELECT** **[ALL** **|** **DISTINCT]**select\_expr**,** select\_expr**,** **...**  **FROM** table\_reference  **[WHERE** where\_condition**]**  **[GROUP** BYcol\_list**]**  **[HAVING** where\_condition**]**  **[ORDER** BYcol\_list**]**  **[CLUSTER** BYcol\_list  **|** **[**DISTRIBUTE **BY** col\_list**]** **[**SORT **BY** col\_list**]**  **]**  **[LIMIT** **number]** |

**解释:**

1、**ORDER BY**用于全局排序，就是对指定的所有排序键进行全局排序，使用ORDER BY的查询语句，最后会用一个Reduce Task来完成全局排序。

2、**sort by**用于分区内排序，即每个Reduce任务内排序**。**，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，还兼具sort by的排序功能。。

因此，如果分桶和sort字段是同一个时，此时，cluster by = distribute by + sort by

#### 全表查询

|  |
| --- |
| **select** **\*** **from** score**;** |

#### 选择特定列查询

|  |
| --- |
| **select** s\_id **,**c\_id **from** score**;** |

#### 列别名

|  |
| --- |
| **select** s\_id **as** myid **,**c\_id **from** score**;** |

#### 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**;** |

#### 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**;** |

#### 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**;** |

### 运算符

##### 比较运算符

**1、操作符**

|  |  |  |
| --- | --- | --- |
| **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, 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、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

|  |  |
| --- | --- |
| 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'**);** |

### 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。注意使用group by分组之后，select后面的字段只能是分组字段和聚合函数。

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**;** |

1. 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**;** |

### JOIN语句

#### 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** teacher t **right** **join** course c **on** t**.**t\_id **=** c**.**t\_id**;** |

#### 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** teacher t **inner** **join** course c **on** t**.**t\_id **=** c**.**t\_id**;** |

#### 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** teacher t **left** **join** course c **on** t**.**t\_id **=** c**.**t\_id**;** |

#### 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** teacher t FULL **JOIN** course c **ON** t**.**t\_id **=** c**.**t\_id **;** |

#### 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** teacher 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。

### 排序

#### Order By-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

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

(2)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)按照学生id和平均成绩进行排序

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

#### Sort By-每个MapReduce内部局部排序

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

|  |
| --- |
| 1**)**设置reduce**个**数  **set** mapreduce**.**job**.**reduces**=**3**;**  2**)**查看设置reduce**个**数  **set** mapreduce**.**job**.**reduces**;**  3）查询成绩按照成绩降序排列  **select** **\*** **from** score sort **by** s\_score**;**  4**)**将查询结果导入到文件**中**（按照成绩降序排列）  **insert** overwrite **local** directory '/export/server/hivedatas/sort' **select** **\*** **from** score sort **by** s\_score**;** |

#### Distribute By-分区排序

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  **set** mapreduce**.**job**.**reduces**=**7**;**  2**)**通过distribute **by**进行数据的分**区**  **insert** overwrite **local** directory '/export/server/hivedatas/sort' **select** **\*** **from** score distribute **by** s\_id sort **by** s\_score**;** |

#### 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:

|  |
| --- |
| **select** **\*** **from** score **cluster** **by** s\_id**;**  **select** **\*** **from** score distribute **by** s\_id sort **by** s\_id**;** |

## Hive查询练习

### 数据准备

**创建表，并将资料中表数据加载到表中(略)**

### 操作实例

|  |
| --- |
| -- 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** teacher **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** teacher 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** teacher 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;** |

## Hive Shell参数

### 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 |

### 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。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。

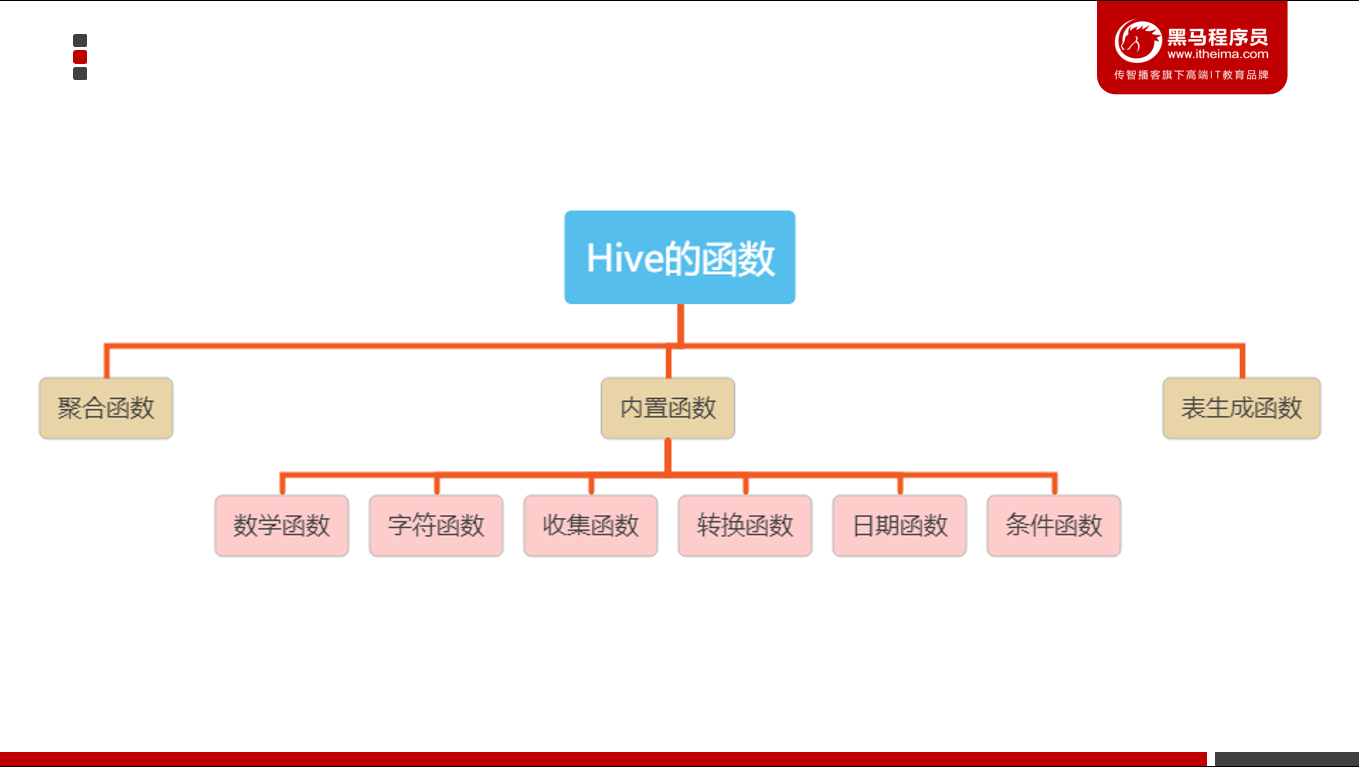
#### 参数设置优先级

**参数声明 > 命令行参数 > 配置文件参数（hive）**

## Hive函数

Hive的函数分为三类： 聚合函数、内置函数，表生成函数，聚合函数之前已经学习过了，接下来学习内置函数和表生成函数.

### Hive的内置函数



#### 数学函数

##### 取整函数: round

语法: round(double a)

返回值: BIGINT

说明:返回double类型的整数值部分（遵循四舍五入）

**举例：**

|  |
| --- |
| hive**>** select round**(**3**.**1415926**);**  3 |

##### 指定精度取整函数: round

语法: round(double a, int d)

返回值: DOUBLE

说明:返回指定精度d的double类型

**举例：**

|  |
| --- |
| hive**>** select round**(**3**.**1415926**,**4**);**  3**.**1416 |

##### 向下取整函数: floor

语法: floor(double a)

返回值: BIGINT

说明:返回等于或者小于该double变量的最大的整数

**举例：**

|  |
| --- |
| hive**>** select floor**(**3**.**1415926**);**  3 |

##### 向上取整函数: ceil

语法: ceil(double a)

返回值: BIGINT

说明:返回等于或者大于该double变量的最小的整数

**举例：**

|  |
| --- |
| hive**>** select ceil**(**3**.**1415926**)** **;**  4 |

##### 取随机数函数: rand

语法: rand(),rand(int seed)

返回值: double

说明:返回一个0到1范围内的随机数。如果指定种子seed，则会返回固定的随机数

**举例：**

|  |
| --- |
| hive**>** select rand**();**  0**.**5577432776034763  hive**>** select rand**();**  0**.**6638336467363424  hive**>** select rand**(**100**);**  0**.**7220096548596434  hive**>** select rand**(**100**);**  0**.**7220096548596434 |

##### 幂运算函数: pow

语法: pow(double a, double p)

返回值: double

说明:返回a的p次幂

**举例：**

|  |
| --- |
| hive**>** select pow**(**2**,**4**)** **;**  16**.**0 |

##### 绝对值函数: abs

语法: abs(double a)  abs(int a)

返回值: double       int

说明:返回数值a的绝对值

**举例：**

|  |
| --- |
| hive**>** select abs**(-**3**.**9**);**  3**.**9  hive**>** select abs**(**10**.**9**);**  10**.**9 |

#### 字符串函数

##### 字符串长度函数：length

语法: length(string A)

返回值: int

说明：返回字符串A的长度

举例：

|  |
| --- |
| hive**>** **select** **length(**'abcedfg'**)**l**;**  7 |

##### 字符串反转函数：reverse

语法: reverse(string A)

返回值: string

说明：返回字符串A的反转结果

举例：

|  |
| --- |
| hive**>** **select** **reverse(**abcedfg’**);**  gfdecba |

##### 字符串连接函数：concat

语法: concat(string A, string B…)

返回值: string

说明：返回输入字符串连接后的结果，支持任意个输入字符串

**举例：**

|  |
| --- |
| hive**>** **select** **concat(**‘abc’**,**'def’,'gh’**);;**  abcdefgh |

##### 字符串连接函数-带分隔符：concat\_ws

语法: concat\_ws(string SEP, string A, string B…)

返回值: string

说明：返回输入字符串连接后的结果，SEP表示各个字符串间的分隔符

**举例：**

|  |
| --- |
| hive**>** **select** **concat\_ws(**','**,**'abc'**,**'def'**,**'gh'**);**  abc**,**def**,**gh |

##### 字符串截取函数：substr,substring

语法: substr(string A, int start),substring(string A, int start)

返回值: string

说明：返回字符串A从start位置到结尾的字符串

**举例：**

|  |
| --- |
| hive**>** **select** **substr(**'abcde'**,**3**);**  cde  hive**>** **select** **substring(**'abcde'**,**3**);**  cde  hive**>select** **substr(**'abcde'**,**-1**);**  e |

##### 字符串截取函数：substr,substring

语法: substr(string A, int start, int len),substring(string A, intstart, int len)

返回值: string

说明：返回字符串A从start位置开始，长度为len的字符串

举例：

|  |
| --- |
| hive**>** **select** **substr(**'abcde'**,**3**,**2**);**  cd  hive**>** **select** **substring(**'abcde'**,**3**,**2**);**  cd  hive**>select** **substring(**'abcde'**,**-2**,**2**);**  de |

##### 字符串转大写函数：upper,ucase

语法: upper(string A) ucase(string A)

返回值: string

说明：返回字符串A的大写格式

举例：

|  |
| --- |
| hive**>** **select** **upper(**'abSEd'**);**  ABSED  hive**>** **select** **ucase(**'abSEd'**);**  ABSED |

##### 字符串转小写函数：lower,lcase

语法: lower(string A) lcase(string A)

返回值: string

说明：返回字符串A的小写格式

**举例：**

|  |
| --- |
| hive**>** **select** **lower(**'abSEd'**);**  absed  hive**>** **select** **lcase(**'abSEd'**);**  absed |

##### 去空格函数：trim

语法: trim(string A)

返回值: string

说明:去除字符串两边的空格

举例：

|  |
| --- |
| hive**>** **select** **trim(**' abc '**);**  abc |

##### 左边去空格函数：ltrim

语法: ltrim(string A)

返回值: string

说明：去除字符串左边的空格

**举例：**

|  |
| --- |
| hive**>** **select** **ltrim(**' abc '**);**  abc |

##### 右边去空格函数：rtrim

语法: rtrim(string A)

返回值: string

说明：去除字符串右边的空格

**举例：**

|  |
| --- |
| hive**>** **select** **rtrim(**' abc '**);**  abc |

##### 正则表达式替换函数：regexp\_replace

语法: regexp\_replace(string A, string B, string C)

返回值: string

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

**举例：**

|  |
| --- |
| hive**>** **select** regexp\_replace**(**'foobar'**,** 'oo|ar'**,** ''**);**  fb |

##### URL解析函数：parse\_url

语法: parse\_url(string urlString, string partToExtract [, stringkeyToExtract])

返回值: string

说明：返回URL中指定的部分。partToExtract的有效值为：HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO.

**举例：**

|  |
| --- |
| hive**>** **select** parse\_url**(**'http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1'**,** 'HOST'**);**  facebook**.**com  hive**>** **select** parse\_url**(**'http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1'**,** 'PATH'**);**  **/**path1**/**p**.**php  hive**>** **select** parse\_url**(**'http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1'**,** 'QUERY'**,**'k1'**);**  v1 |

##### 分割字符串函数: split

语法: split(string str, stringpat)

返回值: array

说明:按照pat字符串分割str，会返回分割后的字符串数组

**举例：**

|  |
| --- |
| hive**>** **select** split**(**'abtcdtef'**,**'t'**);**  **[**"ab"**,**"cd"**,**"ef"**]** |

#### 日期函数

##### 获取当前UNIX时间戳函数:unix\_timestamp

语法: unix\_timestamp()

返回值: bigint

说明:获得当前时区的UNIX时间戳

**举例：**

|  |
| --- |
| hive**>** **select** unix\_timestamp**();**  1323309615 |

##### UNIX时间戳转日期函数:from\_unixtime

语法: from\_unixtime(bigint unixtime[, string format])

返回值: string

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

**举例：**

|  |
| --- |
| hive**>** **select** from\_unixtime**(**1323308943**,**'yyyyMMdd'**);**  20111208 |

##### 日期转UNIX时间戳函数:unix\_timestamp

语法: unix\_timestamp(string date)

返回值: bigint

说明:转换格式为"yyyy-MM-ddHH:mm:ss"的日期到UNIX时间戳。如果转化失败，则返回0。

**举例：**

|  |
| --- |
| hive**>** **select** unix\_timestamp**(**'2011-12-07 13:01:03'**)**;  1323234063 |

##### 指定格式日期转UNIX时间戳函数:unix\_timestamp

语法: unix\_timestamp(string date, string pattern)

返回值: bigint

说明:转换pattern格式的日期到UNIX时间戳。如果转化失败，则返回0。

**举例：**

|  |
| --- |
| hive**>** **select** unix\_timestamp**(**'20111207 13:01:03'**,**'yyyyMMddHH:mm:ss'**)**;  1323234063 |

##### 日期时间转日期函数:to\_date

语法: to\_date(string timestamp)

返回值: string

说明:返回日期时间字段中的日期部分。

举例：

|  |
| --- |
| hive**>** **select** **to\_date(**'2011-12-08 10:03:01'**);**  2011**-**12**-**08 |

##### 日期转年函数: year

语法: year(string date)

返回值: int

说明:返回日期中的年。

举例：

|  |
| --- |
| hive**>** **select** **year(**'2011-12-08 10:03:01'**);**  2011  hive**>** **select** **year(**'2012-12-08'**);**  2012 |

##### 日期转月函数: month

语法: month (string date)

返回值: int

说明:返回日期中的月份。

举例：

|  |
| --- |
| hive**>** **select** **month(**'2011-12-08 10:03:01'**);**  12  hive**>** **select** **month(**'2011-08-08'**);**  8 |

##### 日期转天函数: day

语法: day (string date)

返回值: int

说明:返回日期中的天。

举例：

|  |
| --- |
| hive**>** **select** **day(**'2011-12-08 10:03:01'**);**  8  hive**>** **select** **day(**'2011-12-24'**);**  24 |

同样的，还有 hour,minute,second函数，分别是获取小时，分钟和秒，使用方式和以上类似，这里就不再讲述。

##### 日期转周函数:weekofyear

语法: weekofyear (string date)

返回值: int

说明:返回日期在当前的周数。

举例：

|  |
| --- |
| hive**>** **select** weekofyear**(**'2011-12-08 10:03:01'**);**  49 |

##### 日期比较函数: datediff

语法: datediff(string enddate, string startdate)

返回值: int

说明:返回结束日期减去开始日期的天数。

举例：

|  |
| --- |
| hive**>** **select** **datediff(**'2012-12-08'**,**'2012-05-09'**);**  213 |

##### 日期增加函数: date\_add

语法: date\_add(string startdate, int days)

返回值: string

说明:返回开始日期startdate增加days天后的日期。

举例：

|  |
| --- |
| hive**>** **select** **date\_add(**'2012-12-08'**,**10**);**  2012**-**12**-**18 |

##### 日期减少函数: date\_sub

语法: date\_sub (string startdate, int days)

返回值: string

说明:返回开始日期startdate减少days天后的日期。

举例：

|  |
| --- |
| hive**>** **select** **date\_sub(**'2012-12-08'**,**10**);**  2012**-**11**-**28 |

#### 条件函数

##### if函数: if

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

返回值: T

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

举例：

|  |
| --- |
| hive**>** select **if(**1**=**2**,**100**,**200**)** **;**  200  hive**>** select **if(**1**=**1**,**100**,**200**)** **;**  100 |

##### 条件判断函数：CASE

语法: CASE a WHEN b THEN c [WHEN d THEN e]\* [ELSE f] END

返回值: T

说明：如果a等于b，那么返回c；如果a等于d，那么返回e；否则返回f

举例：

|  |
| --- |
| hive**>** select **case** 100 when 50 then 'tom' when 100 then 'mary'**else** 'tim' end **;**  mary  hive**>** select **case** 200 when 50 then 'tom' when 100 then 'mary'**else** 'tim' end **;**  tim |

##### 条件判断函数：CASE

语法: CASE WHEN a THEN b [WHEN c THEN d]\* [ELSE e] END

返回值: T

说明：如果a为TRUE,则返回b；如果c为TRUE，则返回d；否则返回e

举例：

|  |
| --- |
| hive**>** select **case** when 1**=**2 then 'tom' when 2**=**2 then 'mary' **else**'tim' end **;**  mary  hive**>** select **case** when 1**=**1 then 'tom' when 2**=**2 then 'mary' **else**'tim' end **;**  tom |

#### 转换函数

hive有两个类型转换函数。

1，cast()函数。

公式：

cast(表达式 as 数据类型)

cast函数，可以将"20190607"这样类型的时间数据转化成int类型数据。

cast("20190607" as int)

select cast('2017-06-12' as date) filed;

#### Hive的行转列

##### 介绍

1、行转列是指多行数据转换为一个列的字段。

2、Hive行转列用到的函数：

concat(str1,str2,...) --字段或字符串拼接

concat\_ws(sep, str1,str2) --以分隔符拼接每个字符串

collect\_set(col) --将某字段的值进行去重汇总，产生array类型字段

##### 测试数据:

字段: deptno ename

|  |
| --- |
| 20 SMITH  30 ALLEN  30 WARD  20 JONES  30 MARTIN  30 BLAKE  10 CLARK  20 SCOTT  10 KING  30 TURNER  20 ADAMS  30 JAMES  20 FORD  10 MILLER |

##### 操作步骤

**1:建表**

|  |
| --- |
| **create** **table** emp**(**  deptno **int,**  ename string  **)** **row** **format** delimited fields terminated **by** '\t'**;** |

**2:插入数据：**

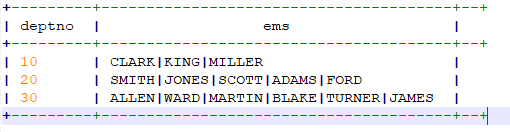
|  |
| --- |
| load **data** **local** inpath "/opt/data/emp.txt" **into** **table** emp**;** |

**3:转换**

|  |
| --- |
| **select** deptno**,concat\_ws(**"|"**,**collect\_set**(**ename**))** **as** ems **from** emp **group** **by** deptno**;** |

行转列，COLLECT\_SET**(**col**)**：函数只接受基本数据类型，它的主要作用是将某字段的值进行去重汇总，产生**array**类型字段。

**4:结果查看**



### Hive的表生成函数

#### explode函数

explode(col)：将hive一列中复杂的array或者map结构拆分成多行。

explode(ARRAY) 列表中的每个元素生成一行

explode(MAP) map中每个key-value对，生成一行，key为一列，value为一列

**数据：**

|  |
| --- |
| 10 CLARK**|**KING**|**MILLER  20 SMITH**|**JONES**|**SCOTT**|**ADAMS**|**FORD  30 ALLEN**|**WARD**|**MARTIN**|**BLAKE**|**TURNER**|**JAMES |

**建表：**

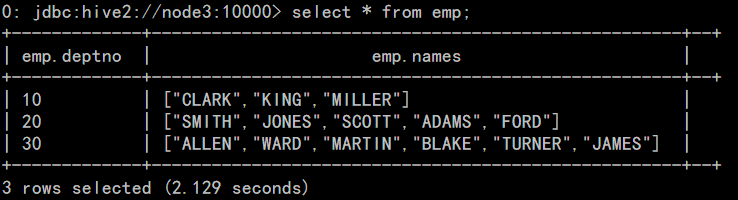
|  |
| --- |
| **create** **table** emp**(**  deptno **int,**  **names** **array<**string**>**  **)**  **row** **format** delimited fields terminated **by** '\t'  collection items terminated **by** '|'**;** |

**插入数据**

|  |
| --- |
| load **data** **local** inpath "/server/data/hivedatas/emp3.txt" **into** **table** emp**;** |

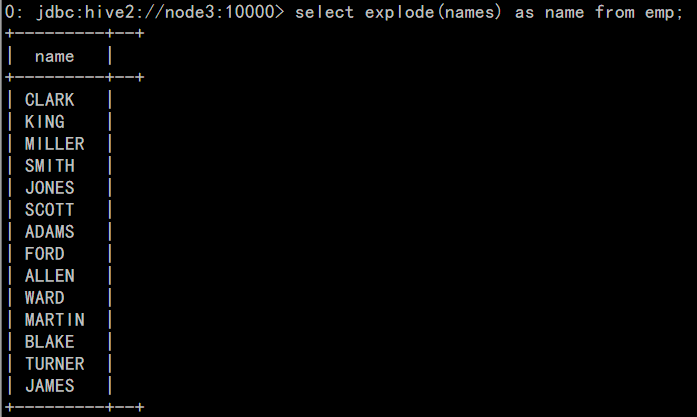
**查询数据**

|  |
| --- |
| **select** **\*** **from** emp**;** |



**使用expload查询**

|  |
| --- |
| **select** explode**(names)** **as** name **from** emp**;** |



#### LATERAL VIEW侧视图

**LATERAL VIEW**  
用法：LATERAL VIEW udtf(expression) tableAlias AS columnAlias  
解释：用于和split, explode等UDTF一起使用，它能够将一列数据拆成多行数据，在此基础上可以对拆分后的数据进行聚合。

**列转行**

|  |
| --- |
| **select** deptno**,**name **from** emp **lateral** **view** explode**(names)** tmp\_tb **as** name**;** |



#### Reflect函数

**reflect函数可以支持在sql中调用java中的自带函数**

##### 使用java.lang.Math当中的Max求两列中最大值

|  |
| --- |
| --创建hive表  **create** **table** test\_udf**(**col1 **int,**col2 **int)** **row** **format** delimited fields terminated **by** ','**;**  --准备数据 test\_udf.txt  1**,**2  4**,**3  6**,**4  7**,**5  5**,**6  --加载数据  load **data** **local** inpath '/root/hivedata/test\_udf.txt' **into** **table** test\_udf**;**  --使用java.lang.Math当中的Max求两列当中的最大值  **select** reflect**(**"java.lang.Math"**,**"max"**,**col1**,**col2**)** **from** test\_udf**;** |

##### 不同记录执行不同的java内置函数

|  |
| --- |
| --创建hive表  **create** **table** test\_udf2**(**class\_name string**,**method\_name string**,**col1 **int** **,** col2 **int)** **row** **format** delimited fields terminated **by** ','**;**  --准备数据 test\_udf2.txt  **java.**lang**.**Math**,min,**1**,**2  **java.**lang**.**Math**,max,**2**,**3  --加载数据  load **data** **local** inpath '/root/hivedata/test\_udf2.txt' **into** **table** test\_udf2**;**  --执行查询  **select** reflect**(**class\_name**,**method\_name**,**col1**,**col2**)** **from** test\_udf2**;** |

### Hive的开窗函数

#### 窗口函数(一) NTILE,ROW\_NUMBER,RANK,DENSE\_RANK

##### 数据准备

|  |
| --- |
| cookie1**,**2018**-**04**-**10**,**1  cookie1**,**2018**-**04**-**11**,**5  cookie1**,**2018**-**04**-**12**,**7  cookie1**,**2018**-**04**-**13**,**3  cookie1**,**2018**-**04**-**14**,**2  cookie1**,**2018**-**04**-**15**,**4  cookie1**,**2018**-**04**-**16**,**4  cookie2**,**2018**-**04**-**10**,**2  cookie2**,**2018**-**04**-**11**,**3  cookie2**,**2018**-**04**-**12**,**5  cookie2**,**2018**-**04**-**13**,**6  cookie2**,**2018**-**04**-**14**,**3  cookie2**,**2018**-**04**-**15**,**9  cookie2**,**2018**-**04**-**16**,**7    **CREATE** **TABLE** as\_t2 **(**  cookieid string**,**  createtime string**,** --day  pv **INT**  **)** **ROW** **FORMAT** DELIMITED  FIELDS TERMINATED **BY** ','  stored **as** textfile**;**    -- 加载数据：  load **data** **local** inpath '/root/hivedata/as\_t2.dat' **into** **table** as\_t2**;** |

##### ROW\_NUMBER

ROW\_NUMBER() 从1开始，按照顺序，生成分组内记录的序列

|  |
| --- |
| **SELECT**  cookieid**,**  createtime**,**  pv**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** pv **desc)** **AS** rn  **FROM** as\_t2**;** |

##### RANK 和 DENSE\_RANK

RANK() 生成数据项在分组中的排名，排名相等会在名次中留下空位

DENSE\_RANK() 生成数据项在分组中的排名，排名相等会在名次中不会留下空位

|  |
| --- |
| **SELECT**  cookieid**,**  createtime**,**  pv**,**  **RANK()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** pv **desc)** **AS** rn1**,**  **DENSE\_RANK()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** pv **desc)** **AS** rn2**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** pv **DESC)** **AS** rn3  **FROM** as\_t2  **WHERE** cookieid **=** 'cookie1'**;** |

#### Hive分析窗口函数(2) SUM,AVG,MIN,MAX

##### 数据准备

|  |
| --- |
| --建表语句:  **create** **table** as\_t1**(**  cookieid string**,**  createtime string**,** --day  pv **int**  **)** **row** **format** delimited  fields terminated **by** ','**;**  --加载数据：  load **data** **local** inpath '/root/hivedata/as\_t1.dat' **into** **table** as\_t1**;**  cookie1**,**2018**-**04**-**10**,**1  cookie1**,**2018**-**04**-**11**,**5  cookie1**,**2018**-**04**-**12**,**7  cookie1**,**2018**-**04**-**13**,**3  cookie1**,**2018**-**04**-**14**,**2  cookie1**,**2018**-**04**-**15**,**4  cookie1**,**2018**-**04**-**16**,**4  --开启智能本地模式  **SET** hive**.exec.mode.local.**auto**=true;** |

##### SUM（结果和ORDER BY相关,默认为升序）

|  |
| --- |
| **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid **order** **by** createtime**)** **as** pv1  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** unbounded preceding **and** **current** **row)** **as** pv2  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid**)** **as** pv3  **from** as\_t1**;** --如果每天order by排序语句 默认把分组内的所有数据进行sum操作  **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** 3 preceding **and** **current** **row)** **as** pv4  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** 3 preceding **and** 1 following**)** **as** pv5  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **sum(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** **current** **row** **and** unbounded following**)** **as** pv6  **from** as\_t1**;**  --pv1: 分组内从起点到当前行的pv累积，如，11号的pv1=10号的pv+11号的pv, 12号=10号+11号+12号  --pv2: 同pv1  --pv3: 分组内(cookie1)所有的pv累加  --pv4: 分组内当前行+往前3行，如，11号=10号+11号， 12号=10号+11号+12号，  13号**=**10号+11号+12号+13号， 14号**=**11号+12号+13号+14号  --pv5: 分组内当前行+往前3行+往后1行，如，14号=11号+12号+13号+14号+15号=5+7+3+2+4=21  --pv6: 分组内当前行+往后所有行，如，13号=13号+14号+15号+16号=3+2+4+4=13，  14号**=**14号+15号+16号**=**2**+**4**+**4**=**10  /\*  - 如果不指定rows between,默认为从起点到当前行;  - 如果不指定order by，则将分组内所有值累加;  - 关键是理解rows between含义,也叫做window子句：  - preceding：往前  - following：往后  - current row：当前行  - unbounded：起点  - unbounded preceding 表示从前面的起点  - unbounded following：表示到后面的终点  \*/ |

##### AVG，MIN，MAX

AVG,MIN,MAX和SUM用法一样

|  |
| --- |
| **select** cookieid**,**createtime**,**pv**,**  **avg(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** unbounded preceding **and** **current** **row)** **as** pv2  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **max(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** unbounded preceding **and** **current** **row)** **as** pv2  **from** as\_t1**;**  **select** cookieid**,**createtime**,**pv**,**  **min(**pv**)** over**(partition** **by** cookieid **order** **by** createtime **rows** **between** unbounded preceding **and** **current** **row)** **as** pv2  **from** as\_t1**;** |

#### Hive分析窗口函数(3) LAG,LEAD,FIRST\_VALUE,LAST\_VALUE

##### 准备数据

|  |
| --- |
| cookie1**,**2018**-**04**-**10 10**:**00**:**02**,**url2  cookie1**,**2018**-**04**-**10 10**:**00**:**00**,**url1  cookie1**,**2018**-**04**-**10 10**:**03**:**04**,**1url3  cookie1**,**2018**-**04**-**10 10**:**50**:**05**,**url6  cookie1**,**2018**-**04**-**10 11**:**00**:**00**,**url7  cookie1**,**2018**-**04**-**10 10**:**10**:**00**,**url4  cookie1**,**2018**-**04**-**10 10**:**50**:**01**,**url5  cookie2**,**2018**-**04**-**10 10**:**00**:**02**,**url22  cookie2**,**2018**-**04**-**10 10**:**00**:**00**,**url11  cookie2**,**2018**-**04**-**10 10**:**03**:**04**,**1url33  cookie2**,**2018**-**04**-**10 10**:**50**:**05**,**url66  cookie2**,**2018**-**04**-**10 11**:**00**:**00**,**url77  cookie2**,**2018**-**04**-**10 10**:**10**:**00**,**url44  cookie2**,**2018**-**04**-**10 10**:**50**:**01**,**url55    **CREATE** **TABLE** as\_t4 **(**  cookieid string**,**  createtime string**,** --页面访问时间  url STRING --被访问页面  **)** **ROW** **FORMAT** DELIMITED  FIELDS TERMINATED **BY** ','  stored **as** textfile**;**  --加载数据：  load **data** **local** inpath '/root/hivedata/as\_t4.dat' **into** **table** as\_t4**;** |

##### LAG

LAG(col,n,DEFAULT) 用于统计窗口内往上第n行值第一个参数为列名，第二个参数为往上第n行（可选，默认为1），第三个参数为默认值（当往上第n行为NULL时候，取默认值，如不指定，则为NULL）

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** rn**,**  **LAG(**createtime**,**1**,**'1970-01-01 00:00:00'**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** last\_1\_time**,**  **LAG(**createtime**,**2**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** last\_2\_time  **FROM** as\_t4**;**  ​  --last\_1\_time: 指定了往上第1行的值，default为'1970-01-01 00:00:00'              cookie1**第**一行，往上1行**为NULL,**因此取默认**值** 1970**-**01**-**01 00**:**00**:**00              cookie1**第**三行，往上1行**值为第**二行**值**，2015**-**04**-**10 10**:**00**:**02              cookie1**第**六行，往上1行**值为第**五行**值**，2015**-**04**-**10 10**:**50**:**01  --last\_2\_time: 指定了往上第2行的值，为指定默认值  cookie1**第**一行，往上2行**为NULL**  cookie1**第**二行，往上2行**为NULL**  cookie1**第**四行，往上2行**为第**二行**值**，2015**-**04**-**10 10**:**00**:**02  cookie1**第**七行，往上2行**为第**五行**值**，2015**-**04**-**10 10**:**50**:**01 |

##### LEAD

与LAG相反LEAD(col,n,DEFAULT) 用于统计窗口内往下第n行值第一个参数为列名，第二个参数为往下第n行（可选，默认为1），第三个参数为默认值（当往下第n行为NULL时候，取默认值，如不指定，则为NULL）

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** rn**,**  **LEAD(**createtime**,**1**,**'1970-01-01 00:00:00'**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** next\_1\_time**,**  **LEAD(**createtime**,**2**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** next\_2\_time  **FROM** as\_t4**;** |

##### FIRST\_VALUE

取分组内排序后，截止到当前行，第一个值

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** rn**,**  **FIRST\_VALUE(**url**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** first1  **FROM** as\_t4**;** |

##### LAST\_VALUE

取分组内排序后，截止到当前行，最后一个值

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** rn**,**  **LAST\_VALUE(**url**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** last1  **FROM** as\_t4**;** |

如果想要取分组内排序后最后一个值，则需要变通一下：

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **ROW\_NUMBER()** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** rn**,**  **LAST\_VALUE(**url**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime**)** **AS** last1**,**  **FIRST\_VALUE(**url**)** OVER**(PARTITION** **BY** cookieid **ORDER** **BY** createtime **DESC)** **AS** last2  **FROM** as\_t4  **ORDER** **BY** cookieid**,**createtime**;** |

##### 特别注意order by

如果不指定ORDER BY，则进行排序混乱，会出现错误的结果

|  |
| --- |
| **SELECT** cookieid**,**  createtime**,**  url**,**  **FIRST\_VALUE(**url**)** OVER**(PARTITION** **BY** cookieid**)** **AS** first2  **FROM** as\_t4**;** |

### Hive自定义函数

#### 概述

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。

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）。

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()

#### 自定义UDF

**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；

**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；

##### 简单

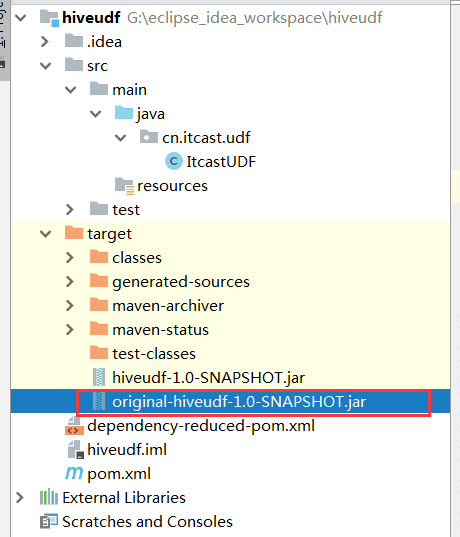
##### 第一步：创建maven java 工程，导入jar包

|  |
| --- |
| **<**dependencies**>**  **<**dependency**>**  **<**groupId**>**org**.**apache**.**hive**</**groupId**>**  **<**artifactId**>**hive**-exec</**artifactId**>**  **<**version**>**2.7.5**</**version**>**  **</**dependency**>**  **<**dependency**>**  **<**groupId**>**org**.**apache**.**hadoop**</**groupId**>**  **<**artifactId**>**hadoop**-**common**</**artifactId**>**  **<**version**>**2.7.5**</**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**>**  **</**plugins**>**  **</**build**>** |

##### 第二步：开发java类继承UDF，并重载evaluate 方法

|  |
| --- |
| **public** **class** MyUDF **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/server/**hive-2.7.5**/**lib  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

|  |
| --- |
| **add** jar **/export/server/**hive-2.7.5**/**lib**/**my\_upper**.**jar**;** |

##### 第五步：设置函数与我们的自定义函数关联

|  |
| --- |
| **create** **temporary** **function** my\_upper **as** 'com.as.udf.asUDF'**;** |

##### 第六步：使用自定义函数

|  |
| --- |
| **select** my\_upper**(**'abc'**);** |

#### 自定义UDTF

##### 需求

自定义一个UDTF，实现将一个任意分隔符的字符串切割成独立的单词,例如:

|  |
| --- |
| **源数据：**  "zookeeper,hadoop,hdfs,hive,MapReduce"  **目标数据:**  zookeeper  hadoop  hdfs  hive  MapReduce |

##### 代码实现

|  |
| --- |
| import org**.**apache**.**hadoop**.**hive**.**ql**.exec.**UDFArgumentException**;**  import org**.**apache**.**hadoop**.**hive**.**ql**.**metadata**.**HiveException**;**  import org**.**apache**.**hadoop**.**hive**.**ql**.**udf**.**generic**.**GenericUDTF**;**  import org**.**apache**.**hadoop**.**hive**.**serde2**.**objectinspector**.**ObjectInspector**;**  import org**.**apache**.**hadoop**.**hive**.**serde2**.**objectinspector**.**ObjectInspectorFactory**;**  import org**.**apache**.**hadoop**.**hive**.**serde2**.**objectinspector**.**StructObjectInspector**;**  import org**.**apache**.**hadoop**.**hive**.**serde2**.**objectinspector**.**primitive**.**PrimitiveObjectInspectorFactory**;**  import **java.**util**.**ArrayList**;**  import **java.**util**.**List**;**  import **java.**util**.function.**ObjDoubleConsumer**;**  **public** **class** MyUDTF **extends** GenericUDTF **{**  **private** final transient **Object[]** forwardListObj **=** **new** **Object[**1**];**  @Override  **public** StructObjectInspector **initialize(**StructObjectInspector argOIs**)** throws UDFArgumentException **{**  **//**设置列名的类型  List**<**String**>** fieldNames **=** **new** ArrayList**<>();**  **//**设置列名  fieldNames**.add(**"column\_01"**);**  List**<**ObjectInspector**>** fieldOIs **=** **new** ArrayList**<**ObjectInspector**>()** **;//**检查器列表  **//**设置输出的列的值类型  fieldOIs**.add(**PrimitiveObjectInspectorFactory**.**javaStringObjectInspector**);**    **return** ObjectInspectorFactory**.**getStandardStructObjectInspector**(**fieldNames**,** fieldOIs**);**  **}**  @Override  **public** void process**(Object[]** objects**)** throws HiveException **{**  **//**1**:**获取原始数据  String args **=** objects**[**0**].**toString**();**  **//**2**:**获取数据传入的第二个参数，此处为分隔符  String splitKey **=** objects**[**1**].**toString**();**  **//**3.将原始数据按照传入的分隔符进行切分  String**[]** fields **=** args**.**split**(**splitKey**);**  **//**4**:**遍历切分后的结果，并写出  **for** **(**String field **:** fields**)** **{**  **//**将每一个单词添加值对象数组  forwardListObj**[**0**]** **=** field**;**  **//**将对象数组内容写出  forward**(**forwardListObj**);**  **}**  **}**  @Override  **public** void **close()** throws HiveException **{**  **}**  **}** |

##### 添加我们的jar包

将打包的jar包上传到node3主机**/export/server/**hive-2.7.5**/**lib目录,并重命名我们的jar包名称

|  |
| --- |
| cd **/export/server/**hive-2.7.5**/**lib  mv original**-**day\_10\_hive\_udtf-1.0**-**SNAPSHOT**.**jar my\_udtf**.**jar |

hive的客户端添加我们的jar包,将jar包添加到hive的classpath下

|  |
| --- |
| **hive>add** jar **/export/server/**hive-2.7.5**/**lib**/**my\_udtf**.**jar |

##### 创建临时函数与开发后的udtf代码关联

|  |
| --- |
| **hive>create** **temporary** **function** my\_udtf **as** 'com.as.udf.AsUDF'**;** |

##### 使用自定义udtf函数

|  |
| --- |
| **hive>select** myudtf**(**"zookeeper,hadoop,hdfs,hive,MapReduce"**,**","**)** word**;** |

## 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

### 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.

### 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 |

### 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;** |

1. 开启mapreduce中map输出压缩功能

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

1. 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**;** |

1. 执行查询语句

|  |
| --- |
| select count**(**1**)** from score**;** |

### 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。

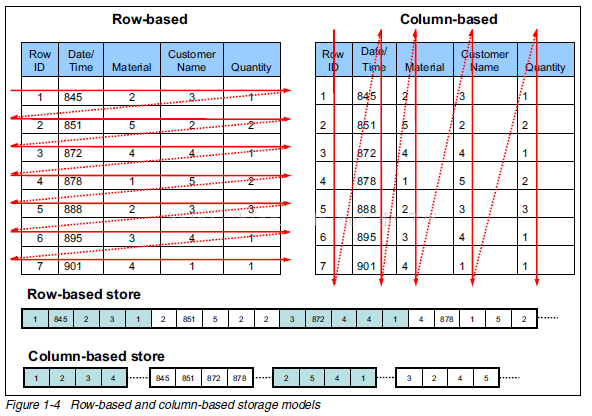
**案例实操：**

|  |
| --- |
| -- 1）开启hive最终输出数据压缩功能  **set** hive**.exec.compress.output=true;**  -- 2）开启mapreduce最终输出数据压缩  **set** mapreduce**.output.**fileoutputformat**.compress=true;**  -- 3）设置mapreduce最终数据输出压缩方式  **set** mapreduce**.output.**fileoutputformat**.compress.**codec **=** org**.**apache**.**hadoop**.**io**.compress.**SnappyCodec**;**  -- 4）设置mapreduce最终数据输出压缩为块压缩  **set** mapreduce**.output.**fileoutputformat**.compress.type=**BLOCK**;**  -- 5）测试一下输出结果是否是压缩文件  **insert** overwrite **local** directory '/export/server/snappy' **select** **\*** **from** score distribute **by** s\_id sort **by** s\_id **desc;** |

## 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）。

### 列式存储和行式存储



**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。

**相比于行式存储，列式存储在分析场景下有着许多优良的特性:**

1）分析场景中往往需要读大量行但是少数几个列。在行存模式下，数据按行连续存储，所有列的数据都存储在一个block中，不参与计算的列在IO时也要全部读出，读取操作被严重放大。而列存模式下，只需要读取参与计算的列即可，极大的减低了IO开销，加速了查询。

2）同一列中的数据属于同一类型，压缩效果显著。列存储往往有着高达十倍甚至更高的压缩比，节省了大量的存储空间，降低了存储成本。

3）更高的压缩比意味着更小的数据空间，从磁盘中读取相应数据耗时更短。

4）自由的压缩算法选择。不同列的数据具有不同的数据类型，适用的压缩算法也就不尽相同。可以针对不同列类型，选择最合适的压缩算法。

**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。**

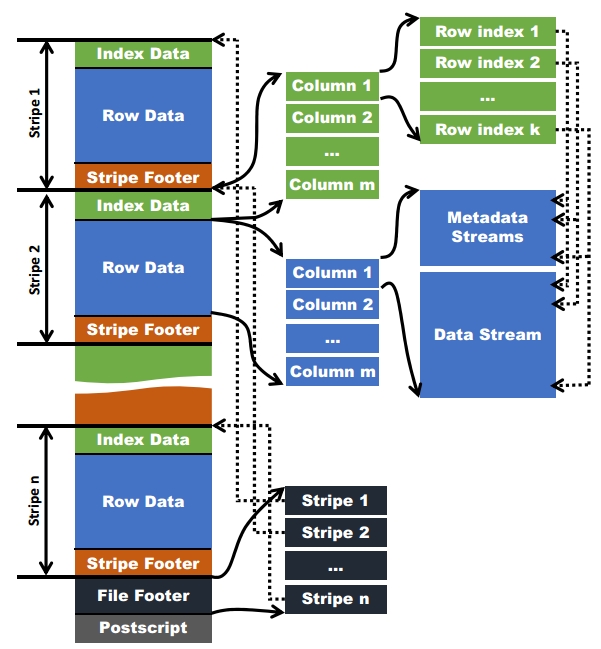
### TEXTFILE格式

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。

### ORC格式

ORC的全称是(Optimized Record Columnar)，使用ORC文件格式可以提高hive读、写和处理数据的能力。

　在ORC格式的hive表中，记录首先会被横向的切分为多个stripes，然后在每一个stripe内数据以列为单位进行存储，所有列的内容都保存在同一个文件中。每个stripe的默认大小为256MB，相对于RCFile每个4MB的stripe而言，更大的stripe使ORC的数据读取更加高效。



ORC在RCFile的基础上进行了一定的改进，所以与RCFile相比，具有以下一些优势：

1、ORC中的特定的序列化与反序列化操作可以使ORC file writer根据数据类型进行写出。

2、提供了多种RCFile中没有的indexes，这些indexes可以使ORC的reader很快的读到需要的数据，并且跳过无用数据，这使得ORC文件中的数据可以很快的得到访问。

3、由于ORC file writer可以根据数据类型进行写出，所以ORC可以支持复杂的数据结构（比如Map等）。

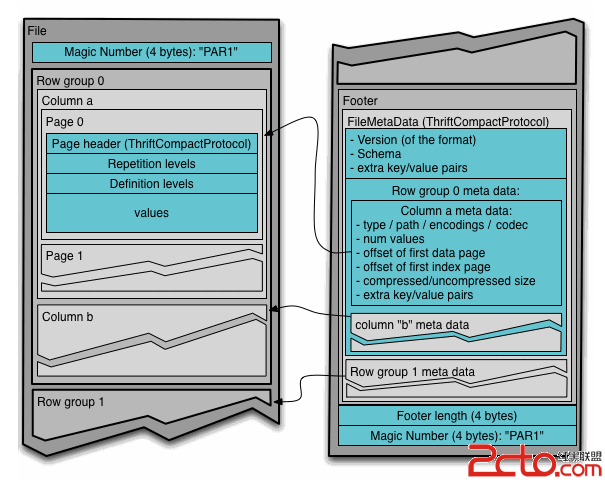
4、除了上面三个理论上就具有的优势之外，ORC的具体实现上还有一些其他的优势，比如ORC的stripe默认大小更大，为ORC writer提供了一个memory manager来管理内存使用情况。

### 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。

Parquet 在同一个数据文件中保存一行中的所有数据，以确保在同一个节点上处理时一行的所有列都可用。Parquet 所做的是设置 HDFS 块大小和最大数据文件大小为 1GB，以确保 I/O 和网络传输请求适用于大批量数据。



Parquet文件在磁盘所有数据分成多个RowGroup 和 Footer。

1. RowGroup: 真正存储数据区域，每一个RowGroup存储多个ColumnChunk的数据。
2. ColumnChunk就代表当前RowGroup某一列的数据，因为可能这一列还在其他RowGroup有数据。ColumnChunk可能包含一个Page。
3. Page是压缩和编码的单元，主要包括PageHeader，RepetitionLevel,DefinitionLevel和Values.
4. PageHeader： 包含一些元数据，诸如编码和压缩类型，有多少数据，当前page第一个数据的偏移量，当前Page第一个索引的偏移量，压缩和解压的大小
5. DefinitionLevel: 当前字段在路径中的深度
6. RepetitionLevel: 当前字段是否可以重复
7. Footer:主要当前文件的元数据和一些统计信息

### 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：**

**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 **;** |

(3)向表中加载数据

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

1. 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OR

|  |
| --- |
| **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 **;** |

1. 向表中加载数据

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

1. 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 **;** |

1. 向表中加载数据

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

1. 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**

### 存储和压缩结合

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"**);** |

1. 插入数据

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

1. 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"**);** |

1. 插入数据

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

1. 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。

## hive调优

### 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。

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**;** |

1. **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**;** |

### 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**;** |

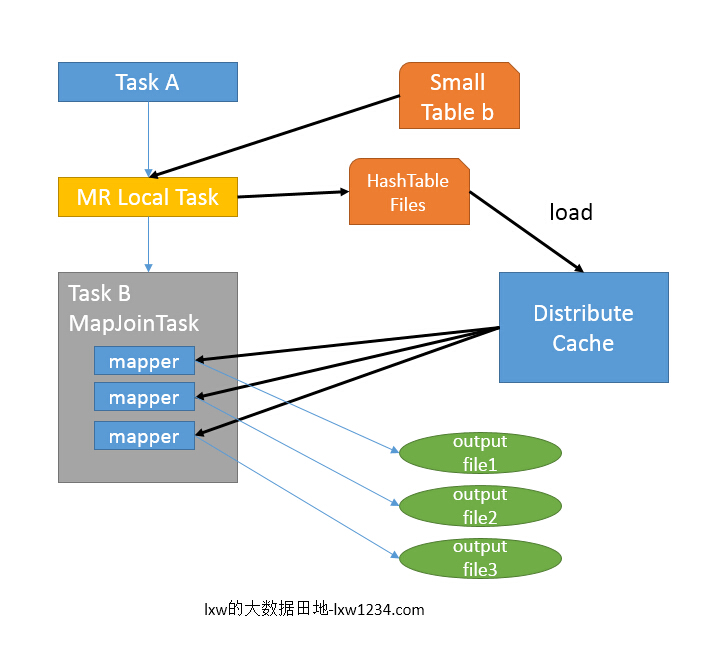
**案例实操：**

|  |
| --- |
| --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**)** |

### Join的优化

#### Map端Join

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。



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。

**map端join的参数设置：**

1. 设置自动选择mapjoin

|  |
| --- |
| **set** hive**.**auto**.convert.join** **=** **true;** -- 默认为true |

1. 大表小表的阈值设置：

|  |
| --- |
| **set** hive**.**mapjoin**.**smalltable**.**filesize**=** 25000000**;** |

小表的输入文件大小的阈值（以字节为单位）;如果文件大小小于此阈值，它将尝试将common join转换为map join。

因此在实际使用中，只要根据业务把握住小表的阈值标准即可，hive会自动帮我们完成mapjoin，提高执行的效率。

#### 大表Join大表

##### 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/server/hivedatas/hive\_big\_table/\*' **into** **table** ori**;**  load **data** **local** inpath '/export/server/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**)** |

##### 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**)** |

##### 案例实操

（1）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）

（2）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'**;** |

1. 分别向大表和小表中导入数据

|  |
| --- |
| load **data** **local** inpath '/export/server/hivedatas/big\_data' **into** **table** bigtable**;**  load **data** **local** inpath '/export/server/hivedatas/small\_data' **into** **table** smalltable**;** |

1. 关闭mapjoin功能（默认是打开的）

|  |
| --- |
| **set** hive**.**auto**.convert.join** **=** **false;** |

1. 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

1. 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）

### SQL优化

#### 列裁剪

Hive在读数据的时候，可以只读取查询中所需要用到的列，而忽略其他列。例如，若有以下查询：

|  |
| --- |
| **SELECT** a**,**b **FROM** q **WHERE** e**<**10**;** |

在实施此项查询中，Q表有5列（a，b，c，d，e），Hive只读取查询逻辑中真实需要的3列a、b、e， 而忽略列c，d；这样做节省了读取开销，中间表存储开销和数据整合开销。

裁剪对应的参数项为：hive.optimize.cp=true（默认值为真）

#### 分区裁剪

　可以在查询的过程中减少不必要的分区。例如，若有以下查询：

|  |
| --- |
| **SELECT** **\*** **FROM** **(SELECT** a1**,** **COUNT(**1**)** **FROM** T **GROUP** **BY** a1**)** subq **WHERE** subq**.**prtn**=**100**;** --（多余分区）  **SELECT** **\*** **FROM** T1 **JOIN** **(SELECT** **\*** **FROM** T2**)** subq **ON** **(**T1**.**a1**=**subq**.**a2**)** **WHERE** subq**.**prtn**=**100**;** |

查询语句若将"subq.prtn=100"条件放入子查询中更为高效，可以减少读入的分区数目。Hive自动执行这种裁剪优化。

　分区参数为：hive.optimize.pruner=true（默认值为真）

#### 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  **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。

#### 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**;** |

测试:

**方式1：**

|  |
| --- |
| **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**)** |

**方式2**

|  |
| --- |
| **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。

#### 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。

### 动态分区调整

#### 参数设置

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

1. 开启动态分区功能（默认true，开启）

|  |
| --- |
| **set** hive**.exec.dynamic.partition=true;** |

（2）设置为非严格模式（动态分区的模式，默认strict，表示必须指定至少一个分区为静态分区，nonstrict模式表示允许所有的分区字段都可以使用动态分区。）

|  |
| --- |
| **set** hive**.exec.dynamic.partition.mode=**nonstrict**;** |

（3）在所有执行MR的节点上，最大一共可以创建多少个动态分区。

|  |
| --- |
| **set** hive**.exec.max.dynamic.**partitions**=**1000**;** |

（4）在每个执行MR的节点上，最大可以创建多少个动态分区。该参数需要根据实际的数据来设定。

|  |
| --- |
| **set** hive**.exec.max.dynamic.**partitions**.**pernode**=**100 |

（5）整个MR Job中，最大可以创建多少个HDFS文件。

​ 在linux系统当中，每个linux用户最多可以开启1024个进程，每一个进程最多可以打开2048个文件，即持有2048个文件句柄，下面这个值越大，就可以打开文件句柄越大

|  |
| --- |
| **set** hive**.exec.max.**created**.**files**=**100000**;** |

1. 当有空分区生成时，是否抛出异常。一般不需要设置。

|  |
| --- |
| **set** hive**.**error**.on.**empty**.partition=false;** |

#### 案例操作

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

|  |
| --- |
| **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'**);** |

1. 创建目标分区表

|  |
| --- |
| **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）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。

|  |
| --- |
| **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。

1. 查看分区

|  |
| --- |
| show partitions ori\_partitioned\_target**;** |

p\_time=20111230000010

p\_time=20111230000011

### 数据倾斜

#### Map数

##### 概述

**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；

##### 减少Map数

|  |
| --- |
| --假设一个SQL任务：  **select** **count(**1**)** **from** tab\_info **where** pt **=** '2020-07-04'**;**  --该任务共有194个文件，其中很多是远远小于128M的小文件，总大小9G，正常执行会用194个map任务。  --Map总共消耗的计算资源：SLOTS\_MILLIS\_MAPS= 623,020  --通过以下方法来在map执行前合并小文件，减少map数：  **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**;**  --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,最终生成了74个块。  **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，  --再执行上面的语句，用了74个map任务，map消耗的计算资源：SLOTS\_MILLIS\_MAPS= 333,500  --对于这个简单SQL任务，执行时间上可能差不多，但节省了一半的计算资源。 |

##### 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** mapred**.**reduce**.**tasks**=**10**;**  **create** **table** a\_1 **as**  **select** **\*** **from** tab\_info 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。  \*/ |

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；

#### Reduce数

Reduce的个数对整个作业的运行性能有很大影响。如果Reduce设置的过大，那么将会产生很多小文件，对NameNode会产生一定的影响，而且整个作业的运行时间未必会减少；如果Reduce设置的过小，那么单个Reduce处理的数据将会加大，很可能会引起OOM异常。

如果设置了mapred.reduce.tasks/mapreduce.job.reduces参数，那么Hive会直接使用它的值作为Reduce的个数；如果mapred.reduce.tasks/mapreduce.job.reduces的值没有设置（也就是-1），那么Hive会根据输入文件的大小估算出Reduce的个数。根据输入文件估算Reduce的个数可能未必很准确，因为Reduce的输入是Map的输出，而Map的输出可能会比输入要小，所以最准确的数根据Map的输出估算Reduce的个数。

**1. Hive自己如何确定reduce数：**

　　reduce个数的设定极大影响任务执行效率，不指定reduce个数的情况下，Hive会猜测确定一个reduce个数，基于以下两个设定：

　hive.exec.reducers.bytes.per.reducer（每个reduce任务处理的数据量，默认为1000^3=1G）

　hive.exec.reducers.max（每个任务最大的reduce数，默认为999）

　计算reducer数的公式很简单N=min（参数2，总输入数据量/参数1），即如果reduce的输入（map的输出）总大小不超过1G，那么只会有一个reduce任务；

如：

|  |
| --- |
| **select** pt**,count(**1**)** **from** tab\_info **where** pt **=** '2020-07-04' **group** **by** pt**;** |

如果源文件总大小为9G多，那么这句会有10个reduce

**2. 调整reduce个数方法一：**

调整hive.exec.reducers.bytes.per.reducer参数的值；

|  |
| --- |
| **set** hive**.exec.**reducers**.**bytes**.**per**.**reducer**=**524288000**;** --（500M）  **select** pt**,** **count(**1**)** **from** tab\_info **where** pt **=** '2020-07-04' **group** **by** pt**;** |

如果源文件总大小为9G多,这次有20个reduce

1. **调整reduce个数方法二：**

|  |
| --- |
| **set** mapred**.**reduce**.**tasks**=**15**;**  **select** pt**,count(**1**)** **from** tab\_info **where** pt **=** '2020-07-04' **group** **by** pt**;** |

这次有15个reduce

**4. reduce个数并不是越多越好；**

　同map一样，启动和初始化reduce也会消耗时间和资源；

　另外，有多少个reduce，就会有个多少个输出文件，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；

**5. 什么情况下只有一个reduce；**

很多时候你会发现任务中不管数据量多大，不管你有没有调整reduce个数的参数，任务中一直都只有一个reduce任务；其实只有一个reduce任务的情况，除了数据量小于hive.exec.reducers.bytes.per.reducer参数值的情况外，还有以下原因：

* 没有group by的汇总，

比如

|  |
| --- |
| **select** pt**,count(**1**)** **from** tab\_info **where** pt **=** ‘2020**-**07**-**04’ **group** **by** pt**;** |

写成

|  |
| --- |
| **select** **count(**1**)** **from** tab\_info **where** pt **=** ‘2020**-**07**-**04’**;** |

这点非常常见

* 用了Order by
* 有笛卡尔积。

　　注意：在设置reduce个数的时候也需要考虑这两个原则：使大数据量利用合适的reduce数；是单个reduce任务处理合适的数据量；

’

### 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。

### 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。

|  |
| --- |
| **set** hive**.**mapred**.mode** **=** strict**;** --开启严格模式  **set** hive**.**mapred**.mode** **=** nostrict**;** --开启非严格模式 |

**配置文件修改:**

|  |
| --- |
| **<property>**  **<name>hive.mapred.mode</name>**  **<value>strict</value>**  **</property>** |

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。

### 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**;** |

### 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>** |

**hive设置开启推测执行参数：**

|  |
| --- |
| **set** mapred**.map.**tasks**.**speculative**.**execution**=true**  **set** mapred**.**reduce**.**tasks**.**speculative**.**execution**=true**  **set** hive**.**mapred**.**reduce**.**tasks**.**speculative**.**execution**=true;** |

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

### 存储方式和压缩方式

大数据场景下存储格式压缩格式尤为关键，可以提升计算速度，减少存储空间，降低网络io，磁盘io，所以要选择合适的压缩格式和存储格式,存储方式和压缩方式之前已经讲过，这里不再描述。