**HIVE学习笔记**

**版本：V1.0**

|  |  |  |  |
| --- | --- | --- | --- |
| 版本管理 | 变化日期 | 变化内容 | 撰写人 |
| V1.0 | 2020-1-1 | 起草 | 王欢 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1. HIVE结构 6](#_Toc280790998)

[1.1 HIVE架构 6](#_Toc280790999)

[1.2 Hive 和 Hadoop 关系 7](#_Toc280791000)

[1.3 Hive 和普通关系数据库的异同 8](#_Toc280791001)

[1.4 HIVE元数据库 9](#_Toc280791002)

[1.4.1 DERBY 9](#_Toc280791003)

[1.4.2 Mysql 10](#_Toc280791004)

[1.5 HIVE的数据存储 11](#_Toc280791005)

[1.6 其它HIVE操作 11](#_Toc280791006)

[2. HIVE 基本操作 12](#_Toc280791007)

[2.1 create table 12](#_Toc280791008)

[2.1.1 总述 12](#_Toc280791009)

[2.1.2 语法 12](#_Toc280791010)

[2.1.3 基本例子 14](#_Toc280791011)

[2.1.4 创建分区 15](#_Toc280791012)

[2.1.5 其它例子 16](#_Toc280791013)

[2.2 Alter Table 17](#_Toc280791014)

[2.2.1 Add Partitions 17](#_Toc280791015)

[2.2.2 Drop Partitions 17](#_Toc280791016)

[2.2.3 Rename Table 17](#_Toc280791017)

[2.2.4 Change Column 18](#_Toc280791018)

[2.2.5 Add/Replace Columns 18](#_Toc280791019)

[2.3 Create View 18](#_Toc280791020)

[2.4 Show 19](#_Toc280791021)

[2.5 Load 19](#_Toc280791022)

[2.6 Insert 21](#_Toc280791023)

[2.6.1 Inserting data into Hive Tables from queries 21](#_Toc280791024)

[2.6.2 Writing data into filesystem from queries 21](#_Toc280791025)

[2.7 Cli 22](#_Toc280791026)

[2.7.1 Hive Command line Options 22](#_Toc280791027)

[2.7.2 Hive interactive Shell Command 24](#_Toc280791028)

[2.7.3 Hive Resources 24](#_Toc280791029)

[2.7.4 调用python、shell等语言 25](#_Toc280791030)

[2.8 DROP 26](#_Toc280791031)

[2.9 其它 27](#_Toc280791032)

[2.9.1 Limit 27](#_Toc280791033)

[2.9.2 Top k 27](#_Toc280791034)

[2.9.3 REGEX Column Specification 27](#_Toc280791035)

[3. Hive Select 27](#_Toc280791036)

[3.1 Group By 28](#_Toc280791037)

[3.2 Order /Sort By 28](#_Toc280791038)

[4. Hive Join 29](#_Toc280791039)

[5. HIVE参数设置 31](#_Toc280791040)

[6. HIVE UDF 33](#_Toc280791041)

[6.1 基本函数 33](#_Toc280791042)

[6.1.1 关系操作符 33](#_Toc280791043)

[6.1.2 代数操作符 34](#_Toc280791044)

[6.1.3 逻辑操作符 35](#_Toc280791045)

[6.1.4 复杂类型操作符 35](#_Toc280791046)

[6.1.5 内建函数 36](#_Toc280791047)

[6.1.6 数学函数 36](#_Toc280791048)

[6.1.7 集合函数 36](#_Toc280791049)

[6.1.8 类型转换 36](#_Toc280791050)

[6.1.9 日期函数 36](#_Toc280791051)

[6.1.10 条件函数 37](#_Toc280791052)

[6.1.11 字符串函数 37](#_Toc280791053)

[6.2 UDTF 39](#_Toc280791054)

[6.2.1 Explode 39](#_Toc280791055)

[7. HIVE 的MAP/REDUCE 41](#_Toc280791056)

[7.1 JOIN 41](#_Toc280791057)

[7.2 GROUP BY 42](#_Toc280791058)

[7.3 DISTINCT 42](#_Toc280791059)

[8. 使用HIVE注意点 43](#_Toc280791060)

[8.1 字符集 43](#_Toc280791061)

[8.2 压缩 43](#_Toc280791062)

[8.3 count(distinct) 43](#_Toc280791063)

[8.4 JOIN 43](#_Toc280791064)

[8.5 DML操作 44](#_Toc280791065)

[8.6 HAVING 44](#_Toc280791066)

[8.7 子查询 44](#_Toc280791067)

[8.8 Join中处理null值的语义区别 44](#_Toc280791068)

[9. 优化与技巧 47](#_Toc280791069)

[9.1 全排序 47](#_Toc280791070)

[9.1.1 例1 48](#_Toc280791071)

[9.1.2 例2 51](#_Toc280791072)

[9.2 怎样做笛卡尔积 54](#_Toc280791073)

[9.3 怎样写exist/in子句 54](#_Toc280791074)

[9.4 怎样决定reducer个数 55](#_Toc280791075)

[9.5 合并MapReduce操作 55](#_Toc280791076)

[9.6 *Bucket 与 sampling* 56](#_Toc280791077)

[9.7 Partition 57](#_Toc280791078)

[9.8 JOIN 58](#_Toc280791079)

[9.8.1 JOIN原则 58](#_Toc280791080)

[9.8.2 Map Join 58](#_Toc280791081)

[9.8.3 大表Join的数据偏斜 60](#_Toc280791082)

[9.9 合并小文件 62](#_Toc280791083)

[9.10 Group By 62](#_Toc280791084)

[10. HIVE FAQ： 62](#_Toc280791085)

# HIVE结构

Hive 是建立在 Hadoop 上的数据仓库基础构架。它提供了一系列的工具，可以用来进行数据提取转化加载（ETL），这是一种可以存储、查询和分析存储在 Hadoop 中的大规模数据的机制。Hive 定义了简单的类 SQL 查询语言，称为 QL，它允许熟悉 SQL 的用户查询数据。同时，这个语言也允许熟悉 MapReduce 开发者的开发自定义的 mapper 和 reducer 来处理内建的 mapper 和 reducer 无法完成的复杂的分析工作。

## HIVE架构

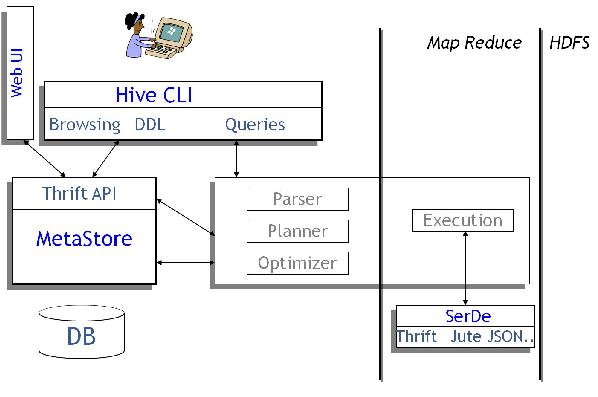
[](http://baike.corp.taobao.com/index.php/File:Hive-arch.JPG)

Hive 的结构可以分为以下几部分：

* 用户接口：包括 CLI, Client, WUI
* 元数据存储。通常是存储在关系数据库如 mysql, derby 中
* 解释器、编译器、优化器、执行器
* Hadoop：用 HDFS 进行存储，利用 MapReduce 进行计算

1. 用户接口主要有三个：CLI，Client 和 WUI。其中最常用的是 CLI，Cli 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在节点，并且在该节点启动 Hive Server。 WUI 是通过浏览器访问 Hive。
2. Hive 将元数据存储在数据库中，如 mysql、derby。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
3. 解释器、编译器、优化器完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后有 MapReduce 调用执行。
4. Hive 的数据存储在 HDFS 中，大部分的查询由 MapReduce 完成（包含 \* 的查询，比如 select \* from tbl 不会生成 MapRedcue 任务）。

## Hive 和 Hadoop 关系

[](http://baike.corp.taobao.com/index.php/File:Hive-hadoop.JPG)

Hive 构建在 Hadoop 之上，

* HQL 中对查询语句的解释、优化、生成查询计划是由 Hive 完成的
* 所有的数据都是存储在 Hadoop 中
* 查询计划被转化为 MapReduce 任务，在 Hadoop 中执行（有些查询没有 MR 任务，如：select \* from table）
* Hadoop和Hive都是用UTF-8编码的

## Hive 和普通关系数据库的异同

|  |  |  |
| --- | --- | --- |
|  | Hive | RDBMS |
| 查询语言 | HQL | SQL |
| 数据存储 | HDFS | Raw Device or Local FS |
| 索引 | 无 | 有 |
| 执行 | MapReduce | Excutor |
| 执行延迟 | 高 | 低 |
| 处理数据规模 | 大 | 小 |

1. 查询语言。由于 SQL 被广泛的应用在数据仓库中，因此，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。
2. 数据存储位置。Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。
3. 数据格式。Hive 中没有定义专门的数据格式，数据格式可以由用户指定，用户定义数据格式需要指定三个属性：列分隔符（通常为空格、”\t”、”\x001″）、行分隔符（”\n”）以及读取文件数据的方法（Hive 中默认有三个文件格式 TextFile，SequenceFile 以及 RCFile）。由于在加载数据的过程中，不需要从用户数据格式到 Hive 定义的数据格式的转换，因此，Hive 在加载的过程中不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的 HDFS 目录中。而在数据库中，不同的数据库有不同的存储引擎，定义了自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。
4. 数据更新。由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不支持对数据的改写和添加，所有的数据都是在加载的时候中确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO ...  VALUES 添加数据，使用 UPDATE ... SET 修改数据。
5. 索引。之前已经说过，Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入， Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。
6. 执行。Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的（类似 select \* from tbl 的查询不需要 MapReduce）。而数据库通常有自己的执行引擎。
7. 执行延迟。之前提到，Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。
8. 可扩展性。由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。
9. 数据规模。由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

## HIVE元数据库

Hive 将元数据存储在 RDBMS 中,一般常用的有MYSQL和DERBY。

### DERBY

启动HIVE的元数据库

进入到hive的安装目录

Eg:

1、启动derby数据库

/home/admin/caona/hive/build/dist/

运行startNetworkServer -h 0.0.0.0

2、连接Derby数据库进行测试

查看/home/admin/caona/hive/build/dist/conf/hive-default.xml。

找到<property>

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

<value>jdbc:derby://hadoop1:1527/metastore\_db;create=true</value>

<description>JDBC connect string for a JDBC metastore</description>

</property>

进入derby安装目录

/home/admin/caona/hive/build/dist/db-derby-10.4.1.3-bin/bin

输入./ij

Connect 'jdbc:derby://hadoop1:1527/metastore\_db;create=true';

3、元数据库数据字典

|  |  |  |
| --- | --- | --- |
| 表名 | 说明 | 关联键 |
| BUCKETING\_COLS |  |  |
| COLUMNS | Hive表字段信息(字段注释，字段名，字段类型，字段序号) | SD\_ID |
| DBS | 元数据库信息，存放HDFS路径信息 | DB\_ID |
| PARTITION\_KEYS | Hive分区表分区键 | PART\_ID |
| SDS | 所有hive表、表分区所对应的hdfs数据目录和数据格式。 | SD\_ID,SERDE\_ID |
| SD\_PARAMS | 序列化反序列化信息，如行分隔符、列分隔符、NULL的表示字符等 | SERDE\_ID |
| SEQUENCE\_TABLE | SEQUENCE\_TABLE表保存了hive对象的下一个可用ID，如’org.apache.hadoop.hive.metastore.model.MTable’, 21，则下一个新创建的hive表其TBL\_ID就是21，同时SEQUENCE\_TABLE表中271786被更新为26(这里每次都是+5?)。同样，COLUMN，PARTITION等都有相应的记录 |  |
| SERDES |  |  |
| SERDE\_PARAMS |  |  |
| SORT\_COLS |  |  |
| TABLE\_PARAMS | 表级属性，如是否外部表，表注释等 | TBL\_ID |
| TBLS | 所有hive表的基本信息 | TBL\_ID,SD\_ID |

从上面几张表的内容来看，hive整个创建表的过程已经比较清楚了

1. 解析用户提交hive语句，对其进行解析，分解为表、字段、分区等hive对象
2. 根据解析到的信息构建对应的表、字段、分区等对象，从SEQUENCE\_TABLE中获取构建对象的最新ID，与构建对象信息(名称，类型等)一同通过DAO方法写入到元数据表中去，成功后将SEQUENCE\_TABLE中对应的最新ID+5。

实际上我们常见的RDBMS都是通过这种方法进行组织的，典型的如postgresql，其系统表中和hive元数据一样裸露了这些id信息(oid,cid等),而Oracle等商业化的系统则隐藏了这些具体的ID。通过这些元数据我们可以很容易的读到数据诸如创建一个表的数据字典信息，比如导出建表语名等。

导出建表语句的shell脚本见*附一 待完成*

### Mysql

将存放元数据的Derby数据库迁移到Mysql数据库

步骤：

## HIVE的数据存储

首先，Hive 没有专门的数据存储格式，也没有为数据建立索引，用户可以非常自由的组织 Hive 中的表，只需要在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。

其次，Hive 中所有的数据都存储在 HDFS 中，Hive 中包含以下数据模型：Table，External Table，Partition，Bucket。

1. Hive 中的 Table 和数据库中的 Table 在概念上是类似的，每一个 Table 在 Hive 中都有一个相应的目录存储数据。例如，一个表 xiaojun，它在 HDFS 中的路径为：/ warehouse /xiaojun，其中，wh 是在 hive-site.xml 中由 ${hive.metastore.warehouse.dir} 指定的数据仓库的目录，所有的 Table 数据（不包括 External Table）都保存在这个目录中。
2. Partition 对应于数据库中的 Partition 列的密集索引，但是 Hive 中 Partition 的组织方式和数据库中的很不相同。在 Hive 中，表中的一个 Partition 对应于表下的一个目录，所有的 Partition 的数据都存储在对应的目录中。例如：xiaojun 表中包含 dt 和 city 两个 Partition，则对应于 dt = 20100801, ctry = US 的 HDFS 子目录为：/ warehouse /xiaojun/dt=20100801/ctry=US；对应于 dt = 20100801, ctry = CA 的 HDFS 子目录为；/ warehouse /xiaojun/dt=20100801/ctry=CA
3. Buckets 对指定列计算 hash，根据 hash 值切分数据，目的是为了并行，每一个 Bucket 对应一个文件。将 user 列分散至 32 个 bucket，首先对 user 列的值计算 hash，对应 hash 值为 0 的 HDFS 目录为：/ warehouse /xiaojun/dt =20100801/ctry=US/part-00000；hash 值为 20 的 HDFS 目录为：/ warehouse /xiaojun/dt =20100801/ctry=US/part-00020
4. External Table 指向已经在 HDFS 中存在的数据，可以创建 Partition。它和 Table 在元数据的组织上是相同的，而实际数据的存储则有较大的差异。

* Table 的创建过程和数据加载过程（这两个过程可以在同一个语句中完成），在加载数据的过程中，实际数据会被移动到数据仓库目录中；之后对数据对访问将会直接在数据仓库目录中完成。删除表时，表中的数据和元数据将会被同时删除。
* External Table 只有一个过程，加载数据和创建表同时完成（CREATE EXTERNAL TABLE ……LOCATION），实际数据是存储在 LOCATION 后面指定的 HDFS 路径中，并不会移动到数据仓库目录中。当删除一个 External Table 时，仅删除元数据，不删除表数据

## Hive metastore服务方式（三种）

一、 metadata 、metastore

Apache hive作为一款基于hadoop的数据仓库工具，其最大的魅力在于可以将位于hdfs的结构化数据映射成为一张表，然后提供了类sql的语法让用户使用来开展对数据的分析，是一款非常棒的离线批处理数仓工具。

在hive的具体使用中，首先面临的问题便是如何定义表结构信息，跟结构化的数据映射成功。所谓的映射指的是一种对应关系。在hive中我们需要描述清楚表跟文件之间的映射关系、列和字段之间的关系等等信息。

我们把这些描述映射关系的数据的称之为hive的元数据。该数据十分重要，因为只有通过查询它才可以确定用户编写sql和最终操作文件之间的关系。

为此我们需要搞清楚下述两个知识：

Metadata即元数据。元数据包含用Hive创建的database、table、表的字段等元信息。元数据存储在关系型数据库中。如hive内置的Derby、第三方如MySQL等。

Metastore即元数据服务，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。

二、 metastore三种配置方式

1. 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。

2. 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。

本地模式下hive的配置主需要指定mysql的相关信息即可。

hive安装包conf/hive-site.xml文件中配置，如果没有，自己创建该文件。

<configuration>

<property>

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

<value>jdbc:mysql://node-1:3306/hive?createDatabaseIfNotExist=true</value>

</property>

<property>

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

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

</property>

<property>

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

<value>root</value>

</property>

<property>

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

<value>hadoop</value>

</property>

</configuration>

3. 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。

三、 Hive Client、Beeline Client

课程中采用远程模式部署hive的metastore服务。使用hive自带的客户端进行连接访问。

在node-1机器上，解压hive的安装包。

注意：以下两件事在启动hive之前必须确保正常完成。

1、选择某台机器提前安装mysql,确保具有远程访问的权限。

2、启动hadoop集群 确保集群正常健康

上传mysql jdbc的驱动包到hive的安装包lib目录下。

修改hive的配置文件。具体如下：

tar zxvf hive-1.1.0-cdh5.14.0.tar.gz

mv hive-1.1.0-cdh5.14.0 hive

cd hive/conf/

mv hive-env.sh.template hive-env.sh

vim conf/hive-env.sh

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

vim hive-site.xml 内容如下：

<configuration>

<property>

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

<value>jdbc:mysql://node-1:3306/hive?createDatabaseIfNotExist=true</value>

</property>

<property>

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

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

</property>

<property>

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

<value>root</value>

</property>

<property>

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

<value>hadoop</value>

</property>

<property>

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

<value>node-1</value>

</property>

<property>

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

<value>thrift://node-1:9083</value>

</property>

</configuration>

第二代客户端Hive 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/servers/hive/bin/hive --service metastore &

nohup /export/servers/hive/bin/hive --service hiveserver2 &

# HIVE 基本操作

## create table

### 总述

* 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 EXIST 选项来忽略这个异常。
* EXTERNAL 关键字可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。
* LIKE 允许用户复制现有的表结构，但是不复制数据。
* 用户在建表的时候可以自定义 SerDe 或者使用自带的 SerDe。如果没有指定 ROW FORMAT 或者 ROW FORMAT DELIMITED，将会使用自带的 SerDe。在建表的时候，用户还需要为表指定列，用户在指定表的列的同时也会指定自定义的 SerDe，Hive 通过 SerDe 确定表的具体的列的数据。
* 如果文件数据是纯文本，可以使用 STORED AS TEXTFILE。如果数据需要压缩，使用 STORED AS SEQUENCE 。
* 有分区的表可以在创建的时候使用 PARTITIONED BY 语句。一个表可以拥有一个或者多个分区，每一个分区单独存在一个目录下。而且，表和分区都可以对某个列进行 CLUSTERED BY 操作，将若干个列放入一个桶（bucket）中。也可以利用SORT BY 对数据进行排序。这样可以为特定应用提高性能。
* 表名和列名不区分大小写，SerDe 和属性名区分大小写。表和列的注释是字符串。

### 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]

| STORED BY 'storage.handler.class.name' [ WITH SERDEPROPERTIES (...) ] (Note: only available starting with 0.6.0)

]

[LOCATION hdfs\_path]

[TBLPROPERTIES (property\_name=property\_value, ...)] (Note: only available starting with 0.6.0)

[AS select\_statement] (Note: this feature is only available starting with 0.5.0.)

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

LIKE existing\_table\_name

[LOCATION hdfs\_path]

data\_type

: primitive\_type

| array\_type

| map\_type

| struct\_type

primitive\_type

: TINYINT

| SMALLINT

| INT

| BIGINT

| BOOLEAN

| FLOAT

| DOUBLE

| STRING

array\_type

: ARRAY < data\_type >

map\_type

: MAP < primitive\_type, data\_type >

struct\_type

: STRUCT < col\_name : data\_type [COMMENT col\_comment], ...>

row\_format

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

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

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

file\_format:

: SEQUENCEFILE

| TEXTFILE

| RCFILE (Note: only available starting with 0.6.0)

| INPUTFORMAT input\_format\_classname OUTPUTFORMAT output\_format\_classname

目前在hive中常用的数据类型有:

BIGINT – 主要用于状态,类别,数量的字段, 如status/option/type/quantity

DOUBLE – 主要用于金额的字段, 如fee/price/bid

STRING – 除上述之外的字段基本都使用String, 尤其是id和日期时间这样的字段

### 基本例子

1、如果一个表已经存在，可以使用if not exists

2、 create table xiaojun(id int,cont string) row format delimited fields terminated by '\005' stored as textfile;

terminated by：关于来源的文本数据的字段间隔符

如果要将自定义间隔符的文件读入一个表，需要通过创建表的语句来指明输入文件间隔符，然后load data到这个表。

4、Alibaba数据库常用间隔符的读取

我们的常用间隔符一般是Ascii码5，Ascii码7等。在hive中Ascii码5用’\005’表示， Ascii码7用’\007’表示，依此类推。

5、装载数据

查看一下：Hadoop fs -ls

LOAD DATA INPATH '/user/admin/xiaojun/a.txt' OVERWRITE INTO TABLE xiaojun;

6、如果使用external建表和普通建表区别

A、指定一个位置，而不使用默认的位置。如：

create EXTERNAL table xiaojun(id int,cont string) row format delimited fields terminated by '\005' stored as textfile location '/user/admin/xiaojun/';

--------------check结果

ij> select LOCATION from tbls a,sds b where a.sd\_id=b.sd\_id and tbl\_name='xiaojun';

-----

LOCATION

--------------------------------------------------------------------------------------------------------------------------------

hdfs://hadoop1:7000/user/admin/xiaojun

ij> select LOCATION from tbls a,sds b where a.sd\_id=b.sd\_id and tbl\_name='c';

----

LOCATION

--------------------------------------------------------------------------------------------------------------------------------

hdfs://hadoop1:7000/user/hive/warehouse/c

B、对于使用create table external建表完成后，再drop掉表，表中的数据还在文件系统中。

如：

hive> create EXTERNAL table xiaojun(id int,cont string) row format delimited fields terminated by '\005' stored as textfile;

----

OK

hive> LOAD DATA INPATH '/user/admin/xiaojun' OVERWRITE INTO TABLE xiaojun;

--------------------------------------------------

Loading data to table xiaojun

OK

hive> drop table xiaojun;

----

OK

[admin@hadoop1 bin]$ ./hadoop fs -ls hdfs://hadoop1:7000/user/hive/warehouse/xiaojun

Found 1 items

使用普通的建表DROP后则找不到

### 创建分区

HIVE的分区通过在创建表时启用partition by实现，用来partition的维度并不是实际数据的某一列，具体分区的标志是由插入内容时给定的。当要查询某一分区的内容时可以采用where语句，形似where tablename.partition\_key > a来实现。

创建含分区的表。

命令原型：

CREATE TABLE page\_view(viewTime INT, userid BIGINT,

page\_url STRING, referrer\_url STRING,

ip STRING COMMENT 'IP Address of the User')

COMMENT 'This is the page view table'

PARTITIONED BY(dt STRING, country STRING)

CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\001'

COLLECTION ITEMS TERMINATED BY '\002'

MAP KEYS TERMINATED BY '\003'

STORED AS SEQUENCEFILE;

Eg:

建表：

CREATE TABLE c02\_clickstat\_fatdt1

(yyyymmdd string,

id INT,

ip string,

country string,

cookie\_id string,

page\_id string ,

clickstat\_url\_id int,

query\_string string,

refer string

)PARTITIONED BY(dt STRING)

row format delimited fields terminated by '\005' stored as textfile;

装载数据：

LOAD DATA INPATH '/user/admin/SqlldrDat/CnClickstat/20101101/19/clickstat\_gp\_fatdt0/0' OVERWRITE INTO TABLE c02\_clickstat\_fatdt1

PARTITION(dt='20101101');

访问某一个分区

SELECT count(\*)

FROM c02\_clickstat\_fatdt1 a

WHERE a.dt >= '20101101' AND a.dt < '20101102';

### 其它例子

1、指定LOCATION位置

CREATE EXTERNAL TABLE page\_view(viewTime INT, userid BIGINT,

page\_url STRING, referrer\_url STRING,

ip STRING COMMENT 'IP Address of the User',

country STRING COMMENT 'country of origination')

COMMENT 'This is the staging page view table'

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\054'

STORED AS TEXTFILE

LOCATION '<hdfs\_location>';

1. 复制一个空表

CREATE TABLE empty\_key\_value\_store

LIKE key\_value\_store;

## Alter Table

### Add Partitions

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

partition\_spec:

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

Eg:

ALTER TABLE c02\_clickstat\_fatdt1 ADD

PARTITION (dt='20101202') location '/user/hive/warehouse/c02\_clickstat\_fatdt1/part20101202'

PARTITION (dt='20101203') location '/user/hive/warehouse/c02\_clickstat\_fatdt1/part20101203';

### Drop Partitions

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

ALTER TABLE c02\_clickstat\_fatdt1 DROP PARTITION (dt='20101202');

### Rename Table

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

这个命令可以让用户为表更名。数据所在的位置和分区名并不改变。换而言之，老的表名并未“释放”，对老表的更改会改变新表的数据。

### Change Column

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

这个命令可以允许改变列名、数据类型、注释、列位置或者它们的任意组合

Eg:

### Add/Replace Columns

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

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

Eg:

hive> desc xi;

OK

id int

cont string

dw\_ins\_date string

Time taken: 0.061 seconds

hive> create table xibak like xi;

OK

Time taken: 0.157 seconds

hive> alter table xibak replace columns (ins\_date string);

OK

Time taken: 0.109 seconds

hive> desc xibak;

OK

ins\_date string

## Create View

CREATE VIEW [IF NOT EXISTS] view\_name [ (column\_name [COMMENT column\_comment], ...) ]

[COMMENT view\_comment]

[TBLPROPERTIES (property\_name = property\_value, ...)]

AS SELECT ...

## Show

查看表名

SHOW TABLES;

查看表名，部分匹配

SHOW TABLES 'page.\*';

SHOW TABLES '.\*view';

查看某表的所有Partition，如果没有就报错：

SHOW PARTITIONS page\_view;

查看某表结构：

DESCRIBE *invites*;

查看分区内容

SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';

查看有限行内容，同Greenplum，用limit关键词

SELECT a.foo FROM invites a limit 3;

查看表分区定义

DESCRIBE EXTENDED page\_view PARTITION (ds='2008-08-08');

## Load

HIVE装载数据没有做任何转换加载到表中的数据只是进入相应的配置单元表的位置移动数据文件。纯加载操作复制/移动操作。

3.1 语法

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

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

* filepath 可以是：
  + 相对路径，例如：project/data1
  + 绝对路径，例如： /user/hive/project/data1
  + 包含模式的完整 URI，例如：hdfs://namenode:9000/user/hive/project/data1
* 加载的目标可以是一个表或者分区。如果表包含分区，必须指定每一个分区的分区名。
* filepath 可以引用一个文件（这种情况下，Hive 会将文件移动到表所对应的目录中）或者是一个目录（在这种情况下，Hive 会将目录中的所有文件移动至表所对应的目录中）。
* 如果指定了 LOCAL，那么：
  + load 命令会去查找本地文件系统中的 filepath。如果发现是相对路径，则路径会被解释为相对于当前用户的当前路径。用户也可以为本地文件指定一个完整的 URI，比如：file:///user/hive/project/data1.
  + load 命令会将 filepath 中的文件复制到目标文件系统中。目标文件系统由表的位置属性决定。被复制的数据文件移动到表的数据对应的位置。
* 如果没有指定 LOCAL 关键字，如果 filepath 指向的是一个完整的 URI，hive 会直接使用这个 URI。 否则：
  + 如果没有指定 schema 或者 authority，Hive 会使用在 hadoop 配置文件中定义的 schema 和 authority，fs.default.name 指定了 Namenode 的 URI。
  + 如果路径不是绝对的，Hive 相对于 /user/ 进行解释。
  + Hive 会将 filepath 中指定的文件内容移动到 table （或者 partition）所指定的路径中。
* 如果使用了 OVERWRITE 关键字，则目标表（或者分区）中的内容（如果有）会被删除，然后再将 filepath 指向的文件/目录中的内容添加到表/分区中。
* 如果目标表（分区）已经有一个文件，并且文件名和 filepath 中的文件名冲突，那么现有的文件会被新文件所替代。

从本地导入数据到表格并追加原表

LOAD DATA LOCAL INPATH `/tmp/pv\_2008-06-08\_us.txt` INTO TABLE c02 PARTITION(date='2008-06-08', country='US')

从本地导入数据到表格并追加记录

LOAD DATA LOCAL INPATH './examples/files/kv1.txt' INTO TABLE pokes;

从hdfs导入数据到表格并覆盖原表

LOAD DATA INPATH '/user/admin/SqlldrDat/CnClickstat/20101101/18/clickstat\_gp\_fatdt0/0' INTO table c02\_clickstat\_fatdt1 OVERWRITE PARTITION (dt='20101201');

关于来源的文本数据的字段间隔符

如果要将自定义间隔符的文件读入一个表，需要通过创建表的语句来指明输入文件间隔符，然后load data到这个表就ok了。

## Insert

### Inserting data into Hive Tables from queries

Standard syntax:

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

Hive extension (multiple inserts):

FROM from\_statement

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

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

Hive extension (dynamic partition inserts):

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

Insert时，from子句既可以放在select子句后，也可以放在insert子句前，下面两句是等价的

hive> FROM invites a INSERT OVERWRITE TABLE events SELECT a.bar, count(\*) WHERE a.foo > 0 GROUP BY a.bar;

hive> INSERT OVERWRITE TABLE events SELECT a.bar, count(\*) FROM invites a WHERE a.foo > 0 GROUP BY a.bar;

hive没有直接插入一条数据的sql，不过可以通过其他方法实现：  
假设有一张表B至少有一条数据，我们想向表A（int，string）中插入一条数据，可以用下面的方法实现：  
from B  
insert table A select  1，‘abc’ limit 1；

我觉得hive好像不能够插入一个记录，因为每次你写insert语句的时候都是要将整个表的值overwrite。我想这个应该是与hive的storage layer是有关系的，因为它的存储层是HDFS，插入一个数据要全表扫描，还不如用整个表的替换来的快些。

Hive不支持一条一条的用insert语句进行插入操作，也不支持update的操作。数据是以load的方式，加载到建立好的表中。数据一旦导入，则不可修改。要么drop掉整个表，要么建立新的表，导入新的数据。

### Writing data into filesystem from queries

Standard syntax:

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

Hive extension (multiple inserts):

FROM from\_statement

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select\_statement1

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

导出文件到本地

INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local\_out' SELECT a.\* FROM pokes a;

导出文件到HDFS

INSERT OVERWRITE DIRECTORY '/user/admin/SqlldrDat/CnClickstat/20101101/19/clickstat\_gp\_fatdt0/0' SELECT a.\* FROM c02\_clickstat\_fatdt1 a WHERE dt=’20101201’;

一个源可以同时插入到多个目标表或目标文件，多目标insert可以用一句话来完成

FROM src

INSERT OVERWRITE TABLE dest1 SELECT src.\* WHERE src.key < 100

INSERT OVERWRITE TABLE dest2 SELECT src.key, src.value WHERE src.key >= 100 and src.key < 200

INSERT OVERWRITE TABLE dest3 PARTITION(ds='2008-04-08', hr='12') SELECT src.key WHERE src.key >= 200 and src.key < 300

INSERT OVERWRITE LOCAL DIRECTORY '/tmp/dest4.out' SELECT src.value WHERE src.key >= 300;

Eg:

from xi

insert overwrite table test2 select '1,2,3' limit 1

insert overwrite table d select '4,5,6' limit 1;

## Cli

### Hive Command line Options

$HIVE\_HOME/bin/hive是一个shell工具，它可以用来运行于交互或批处理方式配置单元查询。

语法：

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

-i <filename> Initialization Sql from file (executed automatically and silently before any other commands)

-e 'quoted query string' Sql from command line

-f <filename> Sql from file

-S Silent mode in interactive shell where only data is emitted

-hiveconf x=y Use this to set hive/hadoop configuration variables.

-e and -f cannot be specified together. In the absence of these options, interactive shell is started.

However, -i can be used with any other options. Multiple instances of -i can be used to execute multiple init scripts.

To see this usage help, run hive -h

运行一个查询：

$HIVE\_HOME/bin/ hive -e 'select count(\*) from c02\_clickstat\_fatdt1'

Example of setting hive configuration variables

$HIVE\_HOME/bin/hive -e 'select a.col from tab1 a' -hiveconf hive.exec.scratchdir=/home/my/hive\_scratch -hiveconf mapred.reduce.tasks=32

将查询结果导出到一个文件

HIVE\_HOME/bin/hive -S -e ' select count(\*) from c02\_clickstat\_fatdt1' > a.txt

运行一个脚本

HIVE\_HOME/bin/hive -f /home/my/hive-script.sql

Example of running an initialization script before entering interactive mode

HIVE\_HOME/bin/hive -i /home/my/hive-init.sql

### Hive interactive Shell Command

|  |  |
| --- | --- |
| **Command** | **Description** |
| quit | 使用 quit or exit 退出 |
| set <key>=<value> | 使用这个方式来设置特定的配置变量的值。有一点需要注意的是，如果你拼错了变量名，CLI将不会显示错误。 |
| set | 这将打印的配置变量，如果没有指定变量则由显示HIVE和用户变量。如set I 则显示i的值,set则显示hive内部变量值 |
| set -v | This will give all possible hadoop/hive configuration variables. |
| add FILE <value> <value>\* | Adds a file to the list of resources. |
| list FILE | list all the resources already added |
| list FILE <value>\* | Check given resources are already added or not. |
| ! <cmd> | execute a shell command from hive shell |
| dfs <dfs command> | execute dfs command command from hive shell |
| <query string> | executes hive query and prints results to stdout |

Eg:

hive> set i=32;

hive> set i;

hive> select a.\* from xiaojun a;

hive> !ls;

hive> dfs -ls;

还可以这样用

hive> set $i='121.61.99.14.128160791368.5';

hive> select count(\*) from c02\_clickstat\_fatdt1 where cookie\_id=$i;

11

### Hive Resources

Hive can manage the addition of resources to a session where those resources need to be made available at query execution time. Any locally accessible file can be added to the session. Once a file is added to a session, hive query can refer to this file by its name (in map/reduce/transform clauses) and this file is available locally at execution time on the entire hadoop cluster. Hive uses Hadoop's Distributed Cache to distribute the added files to all the machines in the cluster at query execution time.

Usage:

* ADD { FILE[S] | JAR[S] | ARCHIVE[S] } <filepath1> [<filepath2>]\*
* LIST { FILE[S] | JAR[S] | ARCHIVE[S] } [<filepath1> <filepath2> ..]
* DELETE { FILE[S] | JAR[S] | ARCHIVE[S] } [<filepath1> <filepath2> ..]
* FILE resources are just added to the distributed cache. Typically, this might be something like a transform script to be executed.
* JAR resources are also added to the Java classpath. This is required in order to reference objects they contain such as UDF's.
* ARCHIVE resources are automatically unarchived as part of distributing them.

Example:

* hive> add FILE /tmp/tt.py;
* hive> list FILES;
* /tmp/tt.py
* hive> from networks a MAP a.networkid USING 'python tt.py' as nn where a.ds = '2009-01-04' limit 10;

It is not neccessary to add files to the session if the files used in a transform script are already available on all machines in the hadoop cluster using the same path name. For example:

* ... MAP a.networkid USING 'wc -l' ...: here wc is an executable available on all machines
* ... MAP a.networkid USING '/home/nfsserv1/hadoopscripts/tt.py' ...: here tt.py may be accessible via a nfs mount point that's configured identically on all the cluster nodes

### 调用python、shell等语言

如下面这句sql就是借用了weekday\_mapper.py对数据进行了处理

CREATE TABLE u\_data\_new (

userid INT,

movieid INT,

rating INT,

weekday INT)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t';

add FILE weekday\_mapper.py;

INSERT OVERWRITE TABLE u\_data\_new

SELECT

TRANSFORM (userid, movieid, rating, unixtime)

USING 'python weekday\_mapper.py'

AS (userid, movieid, rating, weekday)

FROM u\_data;

，其中weekday\_mapper.py内容如下

import sys

import datetime

for line in sys.stdin:

line = line.strip()

userid, movieid, rating, unixtime = line.split('\t')

weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()

print '\t'.join([userid, movieid, rating, str(weekday)])

如下面的例子则是使用了shell的cat命令来处理数据

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

## DROP

删除一个内部表的同时会同时删除表的元数据和数据。删除一个外部表，只删除元数据而保留数据。

## 其它

### Limit

Limit 可以限制查询的记录数。查询的结果是随机选择的。下面的查询语句从 t1 表中随机查询5条记录：

SELECT \* FROM t1 LIMIT 5

### Top k

下面的查询语句查询销售记录最大的 5 个销售代表。

SET mapred.reduce.tasks = 1

SELECT \* FROM sales SORT BY amount DESC LIMIT 5

### ****REGEX Column Specification****

SELECT 语句可以使用正则表达式做列选择，下面的语句查询除了 ds 和 hr 之外的所有列：

SELECT `(ds|hr)?+.+` FROM sales

# Hive Select

语法：

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[ CLUSTER BY col\_list

| [DISTRIBUTE BY col\_list] [SORT BY col\_list]

]

[LIMIT number]

## Group By

基本语法：

groupByClause: GROUP BY groupByExpression (, groupByExpression)\*

groupByExpression: expression

groupByQuery: SELECT expression (, expression)\* FROM src groupByClause?

高级特性：

* 聚合可进一步分为多个表，甚至发送到Hadoop的DFS的文件（可以进行操作，然后使用HDFS的utilitites）。例如我们可以根据性别划分，需要找到独特的页面浏览量按年龄划分。如下面的例子：

FROM pv\_users

INSERT OVERWRITE TABLE pv\_gender\_sum

SELECT pv\_users.gender, count(DISTINCT pv\_users.userid)

GROUP BY pv\_users.gender

INSERT OVERWRITE DIRECTORY '/user/facebook/tmp/pv\_age\_sum'

SELECT pv\_users.age, count(DISTINCT pv\_users.userid)

GROUP BY pv\_users.age;

* hive.map.aggr可以控制怎么进行汇总。默认为为true，配置单元会做的第一级聚合直接在MAP上的任务。这通常提供更好的效率，但可能需要更多的内存来运行成功。

set hive.map.aggr=true;

SELECT COUNT(\*) FROM table2;

PS:在要特定的场合使用可能会加效率。不过我试了一下，比直接使用False慢很多。

## Order /Sort By

Order by 语法：

colOrder: ( ASC | DESC )

orderBy: ORDER BY colName colOrder? (',' colName colOrder?)\*

query: SELECT expression (',' expression)\* FROM src orderBy

Sort By 语法：

Sort顺序将根据列类型而定。如果数字类型的列，则排序顺序也以数字顺序。如果字符串类型的列，则排序顺序将字典顺序。

colOrder: ( ASC | DESC )

sortBy: SORT BY colName colOrder? (',' colName colOrder?)\*

query: SELECT expression (',' expression)\* FROM src sortBy

# Hive Join

语法

join\_table:

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

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

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

table\_reference:

table\_factor

| join\_table

table\_factor:

tbl\_name [alias]

| table\_subquery alias

| ( table\_references )

join\_condition:

ON equality\_expression ( AND equality\_expression )\*

equality\_expression:

expression = expression

Hive 只支持等值连接（equality joins）、外连接（outer joins）和（left/right joins）。Hive 不支持所有非等值的连接，因为非等值连接非常难转化到 map/reduce 任务。另外，Hive 支持多于 2 个表的连接。

写 join 查询时，需要注意几个关键点：

1、只支持等值join

例如：

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

SELECT a.\* FROM a JOIN b

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

是正确的，然而:

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

是错误的。

1. 可以 join 多于 2 个表。

例如

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

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

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

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

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

ON (c.key = b.key1)

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

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

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

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

3．join 时，每次 map/reduce 任务的逻辑：

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

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

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

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

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

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

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

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

例如：

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

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

对应所有 a 表中的记录都有一条记录输出。输出的结果应该是 a.val, b.val，当 a.key=b.key 时，而当 b.key 中找不到等值的 a.key 记录时也会输出 a.val, NULL。“FROM a LEFT OUTER JOIN b”这句一定要写在同一行——意思是 a 表在 b 表的**左边**，所以 a 表中的所有记录都被保留了；“a RIGHT OUTER JOIN b”会保留所有 b 表的记录。OUTER JOIN 语义应该是遵循标准 SQL spec的。

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

SELECT a.val, b.val FROM a

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

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

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

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

ON (a.key=b.key AND

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

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

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

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

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

FROM a

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

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

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

5．LEFT SEMI JOIN 是 IN/EXISTS 子查询的一种更高效的实现。Hive 当前没有实现 IN/EXISTS 子查询，所以你可以用 LEFT SEMI JOIN 重写你的子查询语句。LEFT SEMI JOIN 的限制是， JOIN 子句中右边的表只能在 ON 子句中设置过滤条件，在 WHERE 子句、SELECT 子句或其他地方过滤都不行。

SELECT a.key, a.value

FROM a

WHERE a.key in

(SELECT b.key

FROM B);

可以被重写为：

SELECT a.key, a.val

FROM a LEFT SEMI JOIN b on (a.key = b.key)

# HIVE参数设置

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，Hadoop的配置文件包括

* $HADOOP\_CONF\_DIR/hive-site.xml
* $HADOOP\_CONF\_DIR/hive-default.xml

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。

另外，SerDe参数必须写在DDL（建表）语句中。例如：

create table if not exists t\_dummy(

dummy string

)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'

WITH SERDEPROPERTIES (

'field.delim'='\t',

'escape.delim'='\\',

'serialization.null.format'=' '

) STORED AS TEXTFILE;

类似serialization.null.format这样的参数，必须和某个表或分区关联。在DDL外部声明将不起作用。

# HIVE UDF

## 基本函数

SHOW FUNCTIONS;

DESCRIBE FUNCTION <function\_name>;

### 关系操作符

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand types** | **Description** |
| A = B | All primitive types | TRUE if expression A is equal to expression B otherwise FALSE |
| A == B | None! | Fails because of invalid syntax. SQL uses =, not == |
| A <> B | All primitive types | NULL if A or B is NULL, TRUE if expression A is NOT equal to expression B otherwise FALSE |
| A < B | All primitive types | NULL if A or B is NULL, TRUE if expression A is less than expression B otherwise FALSE |
| A <= B | All primitive types | NULL if A or B is NULL, TRUE if expression A is less than or equal to expression B otherwise FALSE |
| A > B | All primitive types | NULL if A or B is NULL, TRUE if expression A is greater than expression B otherwise FALSE |
| A >= B | All primitive types | NULL if A or B is NULL, TRUE if expression A is greater than or equal to expression B otherwise FALSE |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE |
| A IS NOT NULL | All types | TRUE if expression A evaluates to NULL otherwise FALSE |
| A LIKE B | strings | NULL if A or B is NULL, TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The \_ character in B matches any character in A(similar to . in posix regular expressions) while the % character in B matches an arbitrary number of characters in A(similar to .\* in posix regular expressions) e.g. 'foobar' like 'foo' evaluates to FALSE where as 'foobar' like 'foo\_ \_ \_' evaluates to TRUE and so does 'foobar' like 'foo%' |
| A RLIKE B | strings | NULL if A or B is NULL, TRUE if string A matches the Java regular expression B(See Java regular expressions syntax), otherwise FALSE e.g. 'foobar' rlike 'foo' evaluates to FALSE where as 'foobar' rlike '^f.\*r$' evaluates to TRUE |
| A REGEXP B | strings | Same as RLIKE |

### 代数操作符

返回数字类型，如果任意一个操作符为NULL，则结果为NULL

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand types** | **Description** |
| A + B | All number types | Gives the result of adding A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. e.g. since every integer is a float, therefore float is a containing type of integer so the + operator on a float and an int will result in a float. |
| A - B | All number types | Gives the result of subtracting B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A \* B | All number types | Gives the result of multiplying A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. Note that if the multiplication causing overflow, you will have to cast one of the operators to a type higher in the type hierarchy. |
| A / B | All number types | Gives the result of dividing B from A. The result is a double type. |
| A % B | All number types | Gives the reminder resulting from dividing A by B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A & B | All number types | Gives the result of bitwise AND of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A | B | All number types | Gives the result of bitwise OR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A ^ B | All number types | Gives the result of bitwise XOR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| ~A | All number types | Gives the result of bitwise NOT of A. The type of the result is the same as the type of A. |

### 逻辑操作符

### 复杂类型操作符

|  |  |  |
| --- | --- | --- |
| Constructor Function | Operands | Description |
| Map | (key1, value1, key2, value2, ...) | Creates a map with the given key/value pairs |
| Struct | (val1, val2, val3, ...) | Creates a struct with the given field values. Struct field names will be col1, col2, ... |
| Array | (val1, val2, ...) | Creates an array with the given elements |

### 内建函数

### 数学函数

### 集合函数

### 类型转换

### 日期函数

|  |  |  |
| --- | --- | --- |
| 返回值类型 | 名称 | 描述 |
| string | from\_unixtime(int unixtime) | 将时间戳（unix epoch秒数）转换为日期时间字符串，例如from\_unixtime(0)="1970-01-01 00:00:00" |
| bigint | unix\_timestamp() | 获得当前时间戳 |
| bigint | unix\_timestamp(string date) | 获得date表示的时间戳 |
| bigint | to\_date(string timestamp) | 返回日期字符串，例如to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| string | year(string date) | 返回年，例如year("1970-01-01 00:00:00") = 1970，year("1970-01-01") = 1970 |
| int | month(string date) |  |
| int | day(string date) dayofmonth(date) |  |
| int | hour(string date) |  |
| int | minute(string date) |  |
| int | second(string date) |  |
| int | weekofyear(string date) |  |
| int | datediff(string enddate, string startdate) | 返回enddate和startdate的天数的差，例如datediff('2009-03-01', '2009-02-27') = 2 |
| int | date\_add(string startdate, int days) | 加days天数到startdate: date\_add('2008-12-31', 1) = '2009-01-01' |
| int | date\_sub(string startdate, int days) | 减days天数到startdate: date\_sub('2008-12-31', 1) = '2008-12-30' |

### 条件函数

|  |  |  |
| --- | --- | --- |
| 返回值类型 | 名称 | 描述 |
| - | if(boolean testCondition, T valueTrue, T valueFalseOrNull) | 当testCondition为真时返回valueTrue，testCondition为假或NULL时返回valueFalseOrNull |
| - | COALESCE(T v1, T v2, ...) | 返回列表中的第一个非空元素，如果列表元素都为空则返回NULL |
| - | CASE a WHEN b THEN c [WHEN d THEN e]\* [ELSE f] END | a = b，返回c；a = d，返回e；否则返回f |
| - | CASE WHEN a THEN b [WHEN c THEN d]\* [ELSE e] END | a 为真，返回b；c为真，返回d；否则e |

### 字符串函数

The following are built-in String functions are supported in hive:

|  |  |  |
| --- | --- | --- |
| 返回值类型 | 名称 | **描述** |
| Int | length(string A) | 返回字符串长度 |
| String | reverse(string A) | 反转字符串 |
| String | concat(string A, string B...) | 合并字符串，例如concat('foo', 'bar')='foobar'。注意这一函数可以接受任意个数的参数 |
| String | substr(string A, int start) substring(string A, int start) | 返回子串，例如substr('foobar', 4)='bar' |
| String | substr(string A, int start, int len) substring(string A, int start, int len) | 返回限定长度的子串，例如substr('foobar', 4, 1)='b' |
| String | upper(string A) ucase(string A) | 转换为大写 |
| String | lower(string A) lcase(string A) | 转换为小写 |
| String | trim(string A) |  |
| String | ltrim(string A) |  |
| String | rtrim(string A) |  |
| String | regexp\_replace(string A, string B, string C) | Returns the string resulting from replacing all substrings in B that match the Java regular expression syntax(See Java regular expressions syntax) with C e.g. regexp\_replace("foobar", "oo|ar", "") returns 'fb.' Note that some care is necessary in using predefined character classes: using '\s' as the second argument will match the letter s; '\\s' is necessary to match whitespace, etc. |
| String | regexp\_extract(string subject, string pattern, int intex) | 返回使用正则表达式提取的子字串。例如，regexp\_extract('foothebar', 'foo(.\*?)(bar)', 2)='bar'。注意使用特殊字符的规则：使用'\s'代表的是字符's'；空白字符需要使用'\\s'，以此类推。 |
| String | parse\_url(string urlString, string partToExtract) | 解析URL字符串，partToExtract的可选项有：HOST, PATH, QUERY, REF, PROTOCOL, FILE, AUTHORITY, USERINFO。 |
| 例如， |
| parse\_url('http://facebook.com/path/p1.php?query=1', 'HOST')='facebook.com' |
| parse\_url('http://facebook.com/path/p1.php?query=1', 'PATH')='/path/p1.php' |
| parse\_url('http://facebook.com/path/p1.php?query=1', 'QUERY')='query=1'，可以指定key来返回特定参数，key的格式是QUERY:<KEY\_NAME>，例如QUERY:k1 |
| parse\_url('http://facebook.com/path/p1.php?query=1&field=2','QUERY','query')='1'可以用来取出外部渲染参数key对应的value值 |
| parse\_url('http://facebook.com/path/p1.php?query=1&field=2','QUERY','field')='2' |
| parse\_url('http://facebook.com/path/p1.php?query=1#Ref', 'REF')='Ref' |
| parse\_url('http://facebook.com/path/p1.php?query=1#Ref', 'PROTOCOL')='http' |
| String | get\_json\_object(string json\_string, string path) | 解析json字符串。若源json字符串非法则返回NULL。path参数支持JSONPath的一个子集，包括以下标记： |
| $: Root object |
| []: Subscript operator for array |
| &: Wildcard for [] |
| .: Child operator |
| String | space(int n) | 返回一个包含n个空格的字符串 |
| String | repeat(string str, int n) | 重复str字符串n遍 |
| String | ascii(string str) | 返回str中第一个字符的ascii码 |
| String | lpad(string str, int len, string pad) | 左端补齐str到长度为len。补齐的字符串由pad指定。 |
| String | rpad(string str, int len, string pad) | 右端补齐str到长度为len。补齐的字符串由pad指定。 |
| Array | split(string str, string pat) | 返回使用pat作为正则表达式分割str字符串的列表。例如，split('foobar', 'o')[2] = 'bar'。？不是很明白这个结果 |
| Int | find\_in\_set(string str, string strList) | Returns the first occurance of str in strList where strList is a comma-delimited string. Returns null if either argument is null. Returns 0 if the first argument contains any commas. e.g. find\_in\_set('ab', 'abc,b,ab,c,def') returns 3 |

## UDTF

UDTF即Built-in Table-Generating Functions

使用这些UDTF函数有一些限制:

1、SELECT里面不能有其它字段

如：SELECT pageid, explode(adid\_list) AS myCol...

2、不能嵌套

如：SELECT explode(explode(adid\_list)) AS myCol...不支持

3、不支持GROUP BY / CLUSTER BY / DISTRIBUTE BY / SORT BY

如：SELECT explode(adid\_list) AS myCol ... GROUP BY myCol

### Explode

将数组进行转置

例如：

1、create table test2(mycol array<int>);

2、insert OVERWRITE table test2 select \* from (select array(1,2,3) from a union all select array(7,8,9) from d)c;

3、hive> select \* from test2;

OK

[1,2,3]

[7,8,9]

1. hive> SELECT explode(myCol) AS myNewCol FROM test2;

OK

1

2

3

7

8

9

# HIVE 的MAP/REDUCE

## JOIN

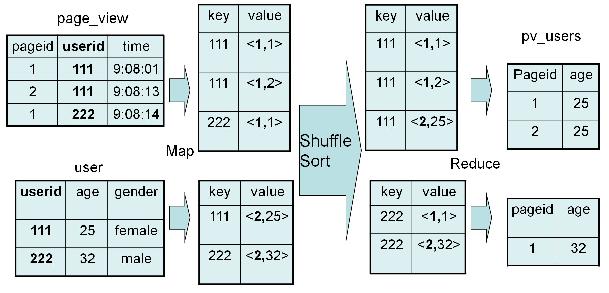
对于 JOIN 操作：

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT pv.pageid, u.age FROM page\_view pv JOIN user u ON (pv.userid = u.userid); |

实现过程为：

* Map：
  + 以 JOIN ON 条件中的列作为 Key，如果有多个列，则 Key 是这些列的组合
  + 以 JOIN 之后所关心的列作为 Value，当有多个列时，Value 是这些列的组合。在 Value 中还会包含表的 Tag 信息，用于标明此 Value 对应于哪个表。
  + 按照 Key 进行排序。
* Shuffle：
  + 根据 Key 的值进行 Hash，并将 Key/Value 对按照 Hash 值推至不同对 Reduce 中。
* Reduce：
  + Reducer 根据 Key 值进行 Join 操作，并且通过 Tag 来识别不同的表中的数据。

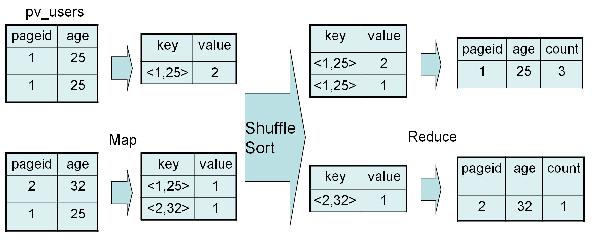
具体实现过程如图：

[](http://baike.corp.taobao.com/index.php/File:Hql-join-mr.JPG)

## GROUP BY

|  |
| --- |
| SELECT pageid, age, count(1) FROM pv\_users GROUP BY pageid, age; |

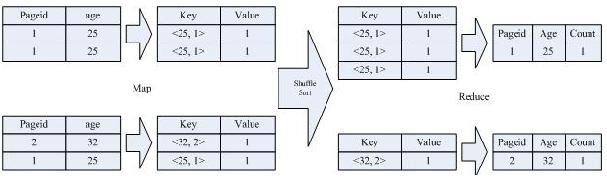
具体实现过程如图：

[](http://baike.corp.taobao.com/index.php/File:Hql-groupby-mr.JPG)

## DISTINCT

|  |
| --- |
| SELECT age, count(distinct pageid) FROM pv\_users GROUP BY age; |

实现过程如图：

[](http://baike.corp.taobao.com/index.php/File:Hql-distinct-mr.JPG)

# 使用HIVE注意点

## 字符集

Hadoop和Hive都是用UTF-8编码的，所以, 所有中文必须是UTF-8编码, 才能正常使用

备注：中文数据load到表里面, 如果字符集不同，很有可能全是乱码需要做转码的, 但是hive本身没有函数来做这个

## 压缩

hive.exec.compress.output 这个参数, 默认是 false，但是很多时候貌似要单独显式设置一遍

否则会对结果做压缩的，如果你的这个文件后面还要在hadoop下直接操作, 那么就不能压缩了

## count(distinct)

当前的 Hive 不支持在一条查询语句中有多 Distinct。如果要在 Hive 查询语句中实现多Distinct，需要使用至少 n+1 条查询语句（n为distinct的数目），前 n 条查询分 别对 n 个列去重，最后一条查询语句对 n 个去重之后的列做 Join 操作，得到最终结果。

## JOIN

只支持等值连接

## DML操作

只支持INSERT/LOAD操作，无UPDATE和DELTE

## HAVING

不支持HAVING操作。如果需要这个功能要嵌套一个子查询用where限制

## 子查询

Hive不支持where子句中的子查询

## Join中处理null值的语义区别

SQL标准中，任何对null的操作（数值比较，字符串操作等）结果都为null。Hive对null值处理的逻辑和标准基本一致，除了Join时的特殊逻辑。

这里的特殊逻辑指的是，Hive的Join中，作为Join key的字段比较，null=null是有意义的，且返回值为true。检查以下查询：

select u.uid, count(u.uid)

from t\_weblog l join t\_user u on (l.uid = u.uid) group by u.uid;

查询中，t\_weblog表中uid为空的记录将和t\_user表中uid为空的记录做连接，即l.uid = u.uid=null成立。

如果需要与标准一致的语义，我们需要改写查询手动过滤null值的情况：

select u.uid, count(u.uid)

from t\_weblog l join t\_user u

on (l.uid = u.uid and l.uid is not null and u.uid is not null)

group by u.uid;

实践中，这一语义区别也是经常导致数据倾斜的原因之一。

## 分号字符

分号是SQL语句结束标记，在HiveQL中也是，但是在HiveQL中，对分号的识别没有那么智慧，例如：

select concat(cookie\_id,concat(';',’zoo’)) from c02\_clickstat\_fatdt1 limit 2;

FAILED: Parse Error: line 0:-1 cannot recognize input '<EOF>' in function specification

可以推断，Hive解析语句的时候，只要遇到分号就认为语句结束，而无论是否用引号包含起来。

解决的办法是，使用分号的八进制的ASCII码进行转义，那么上述语句应写成：

select concat(cookie\_id,concat('\073','zoo')) from c02\_clickstat\_fatdt1 limit 2;

为什么是八进制ASCII码？

我尝试用十六进制的ASCII码，但Hive会将其视为字符串处理并未转义，好像仅支持八进制，原因不详。这个规则也适用于其他非SELECT语句，如CREATE TABLE中需要定义分隔符，那么对不可见字符做分隔符就需要用八进制的ASCII码来转义。

## Insert

### 新增数据

根据语法Insert必须加“OVERWRITE”关键字，也就是说每一次插入都是一次重写。那如何实现表中新增数据呢？

假设Hive中有表xiaojun1,

hive> DESCRIBE xiaojun1;

OK

id int

value int

hive> SELECT \* FROM xiaojun1;

OK

3 4

1 2

2 3

现增加一条记录:

hive> INSERT OVERWRITE TABLE xiaojun1

SELECT id, value FROM (

SELECT id, value FROM xiaojun1

UNION ALL

SELECT 4 AS id, 5 AS value FROM xiaojun1 limit 1

) u;

结果是:

hive>SELECT \* FROM p1;

OK

3 4

4 5

2 3

1 2

其中的关键在于, 关键字UNION ALL的应用, 即将原有数据集和新增数据集进行结合, 然后重写表.

### 插入次序

INSERT OVERWRITE TABLE在插入数据时,是按照后面的SELECT语句中的字段顺序插入的. 也就说, 当id 和value 的位置互换, 那么value将被写入id, 同id被写入value.

### 初始值

INSERT OVERWRITE TABLE在插入数据时, 后面的字段的初始值应注意与表定义中的一致性. 例如, 当为一个STRING类型字段初始为NULL时:

NULL AS field\_name // 这可能会被提示定义类型为STRING, 但这里是void

CAST(NULL AS STRING) AS field\_name // 这样是正确的

又如, 为一个BIGINT类型的字段初始为0时:

CAST(0 AS BIGINT) AS field\_name

# 优化

## HADOOP计算框架特性

* 数据量大不是问题，数据倾斜是个问题。
* jobs数比较多的作业运行效率相对比较低，比如即使有几百行的表，如果多次关联多次汇总，产生十几个jobs，耗时很长。原因是map reduce作业初始化的时间是比较长的。
* sum,count,max,min等UDAF，不怕数据倾斜问题,hadoop在map端的汇总合并优化，使数据倾斜不成问题。
* count(distinct ),在数据量大的情况下，效率较低，如果是多count(distinct )效率更低，因为count(distinct)是按group by 字段分组，按distinct字段排序，一般这种分布方式是很倾斜的，比如男uv,女uv，淘宝一天30亿的pv，如果按性别分组，分配2个reduce,每个reduce处理15亿数据。

## 优化的常用手段

* 好的模型设计事半功倍。
* 解决数据倾斜问题。
* 减少job数。
* 设置合理的map reduce的task数，能有效提升性能。(比如，10w+级别的计算，用160个reduce，那是相当的浪费，1个足够)。
* 了解数据分布，自己动手解决数据倾斜问题是个不错的选择。set hive.groupby.skewindata=true;这是通用的算法优化，但算法优化有时不能适应特定业务背景，开发人员了解业务，了解数据，可以通过业务逻辑精确有效的解决数据倾斜问题。
* 数据量较大的情况下，慎用count(distinct)，count(distinct)容易产生倾斜问题。
* 对小文件进行合并，是行至有效的提高调度效率的方法，假如所有的作业设置合理的文件数，对云梯的整体调度效率也会产生积极的正向影响。
* 优化时把握整体，单个作业最优不如整体最优。

## 全排序

Hive的排序关键字是SORT BY，它有意区别于传统数据库的ORDER BY也是为了强调两者的区别–SORT BY只能在单机范围内排序。

### 例1

set mapred.reduce.tasks=2;

原值

select cookie\_id,page\_id,id from c02\_clickstat\_fatdt1

where cookie\_id IN('1.193.131.218.1288611279693.0','1.193.148.164.1288609861509.2')

1.193.148.164.1288609861509.2 113181412886099008861288609901078194082403 684000005

1.193.148.164.1288609861509.2 127001128860563972141288609859828580660473 684000015

1.193.148.164.1288609861509.2 113181412886099165721288609915890452725326 684000018

1.193.131.218.1288611279693.0 01c183da6e4bc50712881288611540109914561053 684000114

1.193.131.218.1288611279693.0 01c183da6e4bc22412881288611414343558274174 684000118

1.193.131.218.1288611279693.0 01c183da6e4bc50712881288611511781996667988 684000121

1.193.131.218.1288611279693.0 01c183da6e4bc22412881288611523640691739999 684000126

1.193.131.218.1288611279693.0 01c183da6e4bc50712881288611540109914561053 684000128

hive> select cookie\_id,page\_id,id from c02\_clickstat\_fatdt1 where

cookie\_id IN('1.193.131.218.1288611279693.0','1.193.148.164.1288609861509.2')

SORT BY COOKIE\_ID,PAGE\_ID;

SORT排序后的值

1.193.131.218.1288611279693.0 684000118 01c183da6e4bc22412881288611414343558274174 684000118

1.193.131.218.1288611279693.0 684000114 01c183da6e4bc50712881288611540109914561053 684000114

1.193.131.218.1288611279693.0 684000128 01c183da6e4bc50712881288611540109914561053 684000128

1.193.148.164.1288609861509.2 684000005 113181412886099008861288609901078194082403 684000005

1.193.148.164.1288609861509.2 684000018 113181412886099165721288609915890452725326 684000018

1.193.131.218.1288611279693.0 684000126 01c183da6e4bc22412881288611523640691739999 684000126

1.193.131.218.1288611279693.0 684000121 01c183da6e4bc50712881288611511781996667988 684000121

1.193.148.164.1288609861509.2 684000015 127001128860563972141288609859828580660473 684000015

select cookie\_id,page\_id,id from c02\_clickstat\_fatdt1

where cookie\_id IN('1.193.131.218.1288611279693.0','1.193.148.164.1288609861509.2')

ORDER BY PAGE\_ID,COOKIE\_ID;

1.193.131.218.1288611279693.0 684000118 01c183da6e4bc22412881288611414343558274174 684000118

1.193.131.218.1288611279693.0 684000126 01c183da6e4bc22412881288611523640691739999 684000126

1.193.131.218.1288611279693.0 684000121 01c183da6e4bc50712881288611511781996667988 684000121

1.193.131.218.1288611279693.0 684000114 01c183da6e4bc50712881288611540109914561053 684000114

1.193.131.218.1288611279693.0 684000128 01c183da6e4bc50712881288611540109914561053 684000128

1.193.148.164.1288609861509.2 684000005 113181412886099008861288609901078194082403 684000005

1.193.148.164.1288609861509.2 684000018 113181412886099165721288609915890452725326 684000018

1.193.148.164.1288609861509.2 684000015 127001128860563972141288609859828580660473 684000015

可以看到SORT和ORDER排序出来的值不一样。一开始我指定了2个reduce进行数据分发(各自进行排序)。结果不一样的主要原因是上述查询没有reduce key，hive会生成随机数作为reduce key。这样的话输入记录也随机地被分发到不同reducer机器上去了。为了保证reducer之间没有重复的cookie\_id记录，可以使用DISTRIBUTE BY关键字指定分发key为cookie\_id。

select cookie\_id,country,id,page\_id,id from c02\_clickstat\_fatdt1 where cookie\_id IN('1.193.131.218.1288611279693.0','1.193.148.164.1288609861509.2') distribute by cookie\_id SORT BY COOKIE\_ID,page\_id;

1.193.131.218.1288611279693.0 684000118 01c183da6e4bc22412881288611414343558274174 684000118

1.193.131.218.1288611279693.0 684000126 01c183da6e4bc22412881288611523640691739999 684000126

1.193.131.218.1288611279693.0 684000121 01c183da6e4bc50712881288611511781996667988 684000121

1.193.131.218.1288611279693.0 684000114 01c183da6e4bc50712881288611540109914561053 684000114

1.193.131.218.1288611279693.0 684000128 01c183da6e4bc50712881288611540109914561053 684000128

1.193.148.164.1288609861509.2 684000005 113181412886099008861288609901078194082403 684000005

1.193.148.164.1288609861509.2 684000018 113181412886099165721288609915890452725326 684000018

1.193.148.164.1288609861509.2 684000015 127001128860563972141288609859828580660473 684000015

### 例2

CREATE TABLE if not exists t\_order(

id int, -- 订单编号

sale\_id int, -- 销售ID

customer\_id int, -- 客户ID

product \_id int, -- 产品ID

amount int -- 数量

) PARTITIONED BY (ds STRING);

在表中查询所有销售记录，并按照销售ID和数量排序：

set mapred.reduce.tasks=2;

Select sale\_id, amount from t\_order

Sort by sale\_id, amount;

这一查询可能得到非期望的排序。指定的2个reducer分发到的数据可能是（各自排序）：

Reducer1：

Sale\_id | amount

0 | 100

1 | 30

1 | 50

2 | 20

Reducer2：

Sale\_id | amount

0 | 110

0 | 120

3 | 50

4 | 20

使用DISTRIBUTE BY关键字指定分发key为sale\_id。改造后的HQL如下：

set mapred.reduce.tasks=2;

Select sale\_id, amount from t\_order

Distribute by sale\_id

Sort by sale\_id, amount;

这样能够保证查询的销售记录集合中，销售ID对应的数量是正确排序的，但是销售ID不能正确排序，原因是hive使用hadoop默认的HashPartitioner分发数据。

这就涉及到一个全排序的问题。解决的办法无外乎两种：

1.) 不分发数据，使用单个reducer：

set mapred.reduce.tasks=1;

这一方法的缺陷在于reduce端成为了性能瓶颈，而且在数据量大的情况下一般都无法得到结果。但是实践中这仍然是最常用的方法，原因是通常排序的查询是为了得到排名靠前的若干结果，因此可以用limit子句大大减少数据量。使用limit n后，传输到reduce端（单机）的数据记录数就减少到n\* （map个数）。

2.) 修改Partitioner，这种方法可以做到全排序。这里可以使用Hadoop自带的TotalOrderPartitioner（来自于Yahoo!的TeraSort项目），这是一个为了支持跨reducer分发有序数据开发的Partitioner，它需要一个SequenceFile格式的文件指定分发的数据区间。如果我们已经生成了这一文件（存储在/tmp/range\_key\_list，分成100个reducer），可以将上述查询改写为

set mapred.reduce.tasks=100;

set hive.mapred.partitioner=org.apache.hadoop.mapred.lib.TotalOrderPartitioner;

set total.order.partitioner.path=/tmp/ range\_key\_list;

Select sale\_id, amount from t\_order

Cluster by sale\_id

Sort by amount;

有很多种方法生成这一区间文件（例如hadoop自带的o.a.h.mapreduce.lib.partition.InputSampler工具）。这里介绍用Hive生成的方法，例如有一个按id有序的t\_sale表：

CREATE TABLE if not exists t\_sale (

id int,

name string,

loc string

);

则生成按sale\_id分发的区间文件的方法是：

create external table range\_keys(sale\_id int)

row format serde

'org.apache.hadoop.hive.serde2.binarysortable.BinarySortableSerDe'

stored as

inputformat

'org.apache.hadoop.mapred.TextInputFormat'

outputformat

'org.apache.hadoop.hive.ql.io.HiveNullValueSequenceFileOutputFormat'

location '/tmp/range\_key\_list';

insert overwrite table range\_keys

select distinct sale\_id

from source t\_sale sampletable(BUCKET 100 OUT OF 100 ON rand()) s

sort by sale\_id;

生成的文件（/tmp/range\_key\_list目录下）可以让TotalOrderPartitioner按sale\_id有序地分发reduce处理的数据。区间文件需要考虑的主要问题是数据分发的均衡性，这有赖于对数据深入的理解。

## 怎样做笛卡尔积

当Hive设定为严格模式（hive.mapred.mode=strict）时，不允许在HQL语句中出现笛卡尔积，这实际说明了Hive对笛卡尔积支持较弱。因为找不到Join key，Hive只能使用1个reducer来完成笛卡尔积。

当然也可以用上面说的limit的办法来减少某个表参与join的数据量，但对于需要笛卡尔积语义的需求来说，经常是一个大表和一个小表的Join操作，结果仍然很大（以至于无法用单机处理），这时MapJoin才是最好的解决办法。

MapJoin，顾名思义，会在Map端完成Join操作。这需要将Join操作的一个或多个表完全读入内存。

MapJoin的用法是在查询/子查询的SELECT关键字后面添加/\*+ MAPJOIN(tablelist) \*/提示优化器转化为MapJoin（目前Hive的优化器不能自动优化MapJoin）。其中tablelist可以是一个表，或以逗号连接的表的列表。tablelist中的表将会读入内存，应该将小表写在这里。

PS：有用户说MapJoin在子查询中可能出现未知BUG。在大表和小表做笛卡尔积时，规避笛卡尔积的方法是，给Join添加一个Join key，原理很简单：将小表扩充一列join key，并将小表的条目复制数倍，join key各不相同；将大表扩充一列join key为随机数。

## 怎样写exist/in子句

Hive不支持where子句中的子查询，SQL常用的exist in子句需要改写。这一改写相对简单。考虑以下SQL查询语句：

SELECT a.key, a.value

FROM a

WHERE a.key in

(SELECT b.key

FROM B);

可以改写为

SELECT a.key, a.value

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

WHERE b.key <> NULL;

一个更高效的实现是利用left semi join改写为：

SELECT a.key, a.val

FROM a LEFT SEMI JOIN b on (a.key = b.key);

left semi join是0.5.0以上版本的特性。

## 怎样决定reducer个数

Hadoop MapReduce程序中，reducer个数的设定极大影响执行效率，这使得Hive怎样决定reducer个数成为一个关键问题。遗憾的是Hive的估计机制很弱，不指定reducer个数的情况下，Hive会猜测确定一个reducer个数，基于以下两个设定：

1. hive.exec.reducers.bytes.per.reducer（默认为1000^3）

2. hive.exec.reducers.max（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)

通常情况下，有必要手动指定reducer个数。考虑到map阶段的输出数据量通常会比输入有大幅减少，因此即使不设定reducer个数，重设参数2还是必要的。依据Hadoop的经验，可以将参数2设定为0.95\*(集群中TaskTracker个数)。

## 合并MapReduce操作

**Multi-group by**

Multi-group by是Hive的一个非常好的特性，它使得Hive中利用中间结果变得非常方便。例如，

FROM (SELECT a.status, b.school, b.gender

FROM status\_updates a JOIN profiles b

ON (a.userid = b.userid and

a.ds='2009-03-20' )

) subq1

INSERT OVERWRITE TABLE gender\_summary

PARTITION(ds='2009-03-20')

SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender

INSERT OVERWRITE TABLE school\_summary

PARTITION(ds='2009-03-20')

SELECT subq1.school, COUNT(1) GROUP BY subq1.school

上述查询语句使用了Multi-group by特性连续group by了2次数据，使用不同的group by key。这一特性可以减少一次MapReduce操作。

**Multi-distinct**

Multi-distinct是淘宝开发的另一个multi-xxx特性，使用Multi-distinct可以在同一查询/子查询中使用多个distinct，这同样减少了多次MapReduce操作

## *Bucket 与 sampling*

Bucket是指将数据以指定列的值为key进行hash，hash到指定数目的桶中。这样就可以支持高效采样了。

如下例就是以userid这一列为bucket的依据，共设置32个buckets

CREATE TABLE page\_view(viewTime INT, userid BIGINT,

page\_url STRING, referrer\_url STRING,

ip STRING COMMENT 'IP Address of the User')

COMMENT 'This is the page view table'

PARTITIONED BY(dt STRING, country STRING)

CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '1'

COLLECTION ITEMS TERMINATED BY '2'

MAP KEYS TERMINATED BY '3'

STORED AS SEQUENCEFILE;

Sampling可以在全体数据上进行采样，这样效率自然就低，它还是要去访问所有数据。而如果一个表已经对某一列制作了bucket，就可以采样所有桶中指定序号的某个桶，这就减少了访问量。

如下例所示就是采样了page\_view中32个桶中的第三个桶。

SELECT \* FROM page\_view TABLESAMPLE(BUCKET 3 OUT OF 32);

## Partition

Partition就是分区。分区通过在创建表时启用partition by实现，用来partition的维度并不是实际数据的某一列，具体分区的标志是由插入内容时给定的。当要查询某一分区的内容时可以采用where语句，形似where tablename.partition\_key > a来实现。

创建含分区的表

CREATE TABLE page\_view(viewTime INT, userid BIGINT,

page\_url STRING, referrer\_url STRING,

ip STRING COMMENT 'IP Address of the User')

PARTITIONED BY(date STRING, country STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '1'

STORED AS TEXTFILE;

载入内容，并指定分区标志

LOAD DATA LOCAL INPATH `/tmp/pv\_2008-06-08\_us.txt` INTO TABLE page\_view PARTITION(date='2008-06-08', country='US');

查询指定标志的分区内容

SELECT page\_views.\*

FROM page\_views

WHERE page\_views.date >= '2008-03-01' AND page\_views.date <= '2008-03-31' AND page\_views.referrer\_url like '%xyz.com';

## JOIN

### JOIN原则

在使用写有 Join 操作的查询语句时有一条原则：应该将条目少的表/子查询放在 Join 操作符的左边。原因是在 Join 操作的 Reduce 阶段，位于 Join 操作符左边的表的内容会被加载进内存，将条目少的表放在左边，可以有效减少发生 OOM 错误的几率。对于一条语句中有多个 Join 的情况，如果 Join 的条件相同，比如查询：

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT pv.pageid, u.age FROM page\_view p  JOIN user u ON (pv.userid = u.userid)  JOIN newuser x ON (u.userid = x.userid); |

* 如果 Join 的 key 相同，不管有多少个表，都会则会合并为一个 Map-Reduce
* 一个 Map-Reduce 任务，而不是 ‘n’ 个
* 在做 OUTER JOIN 的时候也是一样

如果 Join 的条件不相同，比如：

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT pv.pageid, u.age FROM page\_view p  JOIN user u ON (pv.userid = u.userid)  JOIN newuser x on (u.age = x.age); |

Map-Reduce 的任务数目和 Join 操作的数目是对应的，上述查询和以下查询是等价的：

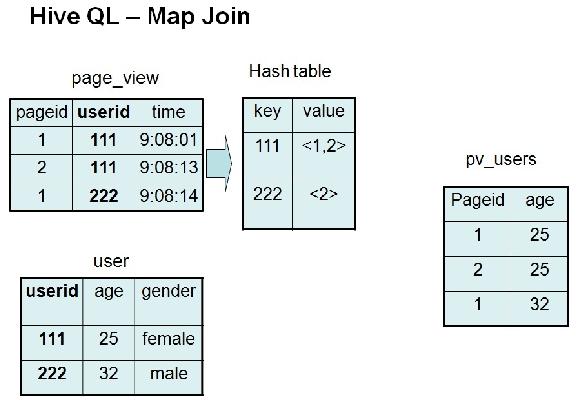
|  |
| --- |
| INSERT OVERWRITE TABLE tmptable  SELECT \* FROM page\_view p JOIN user u  ON (pv.userid = u.userid);  INSERT OVERWRITE TABLE pv\_users  SELECT x.pageid, x.age FROM tmptable x  JOIN newuser y ON (x.age = y.age); |

### Map Join

Join 操作在 Map 阶段完成，不再需要Reduce，前提条件是需要的数据在 Map 的过程中可以访问到。比如查询：

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT /\*+ MAPJOIN(pv) \*/ pv.pageid, u.age  FROM page\_view pv  JOIN user u ON (pv.userid = u.userid); |

可以在 Map 阶段完成 Join，如图所示：

[](http://baike.corp.taobao.com/index.php/File:Hive-ql-mapjoin.JPG)

相关的参数为：

* **hive.join.emit.interval = 1000** How many rows in the right-most join operand Hive should buffer before emitting the join result.
* **hive.mapjoin.size.key = 10000**
* **hive.mapjoin.cache.numrows = 10000**

## 数据倾斜

### 空值数据倾斜

**场景：**如日志中，常会有信息丢失的问题，比如全网日志中的user\_id，如果取其中的user\_id和bmw\_users关联，会碰到数据倾斜的问题。

**解决方法1：** user\_id为空的不参与关联

Select \* From log a

Join bmw\_users b

On a.user\_id is not null

And a.user\_id = b.user\_id

Union all

Select \* from log a

where a.user\_id is null;

**解决方法2 ：**赋与空值分新的key值

Select \*

from log a

left outer join bmw\_users b

**on case when a.user\_id is null then concat(‘dp\_hive’,rand() ) else a.user\_id end = b.user\_id;**

**结论：**方法2比方法效率更好，不但io少了，而且作业数也少了。方法1 log读取两次，jobs是2。方法2 job数是1 。这个优化适合无效id(比如-99,’’,null等)产生的倾斜问题。把空值的key变成一个字符串加上随机数，就能把倾斜的数据分到不同的reduce上 ,解决数据倾斜问题。附上hadoop通用关联的实现方法（关联通过二次排序实现的，关联的列为parition key,关联的列c1和表的tag组成排序的group key,根据parition key分配reduce。同一reduce内根据group key排序）

### 不同数据类型关联产生数据倾斜

**场景：**一张表s8的日志，每个商品一条记录，要和商品表关联。但关联却碰到倾斜的问题。s8的日志中有字符串商品id,也有数字的商品id,类型是string的，但商品中的数字id是bigint的。猜测问题的原因是把s8的商品id转成数字id做hash来分配reduce，所以字符串id的s8日志，都到一个reduce上了，解决的方法验证了这个猜测。

**解决方法：**把数字类型转换成字符串类型

Select \* from s8\_log a

Left outer join r\_auction\_auctions b

On a.auction\_id = cast(b.auction\_id as string);

### 大表Join的数据偏斜

MapReduce编程模型下开发代码需要考虑数据偏斜的问题，Hive代码也是一样。数据偏斜的原因包括以下两点：

1. Map输出key数量极少，导致reduce端退化为单机作业。

2. Map输出key分布不均，少量key对应大量value，导致reduce端单机瓶颈。

Hive中我们使用MapJoin解决数据偏斜的问题，即将其中的某个表（全量）分发到所有Map端进行Join，从而避免了reduce。这要求分发的表可以被全量载入内存。

极限情况下，Join两边的表都是大表，就无法使用MapJoin。

这种问题最为棘手，目前已知的解决思路有两种：

1. 如果是上述情况1，考虑先对Join中的一个表去重，以此结果过滤无用信息。这样一般会将其中一个大表转化为小表，再使用MapJoin 。

一个实例是广告投放效果分析，例如将广告投放者信息表i中的信息填充到广告曝光日志表w中，使用投放者id关联。因为实际广告投放者数量很少（但是投放者信息表i很大），因此可以考虑先在w表中去重查询所有实际广告投放者id列表，以此Join过滤表i，这一结果必然是一个小表，就可以使用MapJoin。

2. 如果是上述情况2，考虑切分Join中的一个表为多片，以便将切片全部载入内存，然后采用多次MapJoin得到结果。

一个实例是商品浏览日志分析，例如将商品信息表i中的信息填充到商品浏览日志表w中，使用商品id关联。但是某些热卖商品浏览量很大，造成数据偏斜。例如，以下语句实现了一个inner join逻辑，将商品信息表拆分成2个表：

select \* from

(

select w.id, w.time, w.amount, i1.name, i1.loc, i1.cat

from w left outer join i sampletable(1 out of 2 on id) i1

)

union all

(

select w.id, w.time, w.amount, i2.name, i2.loc, i2.cat

from w left outer join i sampletable(1 out of 2 on id) i2

)

);

以下语句实现了left outer join逻辑：

select t1.id, t1.time, t1.amount,

coalease(t1.name, t2.name),

coalease(t1.loc, t2.loc),

coalease(t1.cat, t2.cat)

from (

select w.id, w.time, w.amount, i1.name, i1.loc, i1.cat

from w left outer join i sampletable(1 out of 2 on id) i1

) t1 left outer join i sampletable(2 out of 2 on id) t2;

上述语句使用Hive的sample table特性对表做切分。

## 合并小文件

文件数目过多，会给 HDFS 带来压力，并且会影响处理效率，可以通过合并 Map 和 Reduce 的结果文件来消除这样的影响：

hive.merge.mapfiles = true 是否和并 Map 输出文件，默认为 True

hive.merge.mapredfiles = false 是否合并 Reduce 输出文件，默认为 False

hive.merge.size.per.task = 256\*1000\*1000 合并文件的大小

## Group By

* Map 端部分聚合：

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。

基于 Hash

参数包括：

* + - hive.map.aggr = true 是否在 Map 端进行聚合，默认为 True
    - hive.groupby.mapaggr.checkinterval = 100000 在 Map 端进行聚合操作的条目数目
* 有数据倾斜的时候进行负载均衡

hive.groupby.skewindata = false

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 的输出结果集合会随机分布到 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。

# HIVE FAQ

1、 [admin@hadoop1 ~]$ hive

Cannot find hadoop installation: $HADOOP\_HOME must be set or hadoop must be in the path

原因：HADOOP路径没有在环境变量中定义

解决方法：admin@hadoop1 ~]$ export HADOOP\_HOME=$HOME/hadoop-0.19.2

2、FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask

解决方案：Hive的元数据库derby服务没有启动

进入到hive的安装目录

/home/admin/caona/hive/build/dist/db-derby-10.4.1.3-bin/bin

运行startNetworkServer -h 0.0.0.0

3、[admin@hadoop1 conf]$ hive

Unable to create log directory ${build.dir}/tmp

原因：存放日志文件的目录被人删除了。

解决方法：进行到${build.dir}下面，创建一个tmp目录。

如：[admin@hadoop1 build]$ pwd

/home/admin/caona/hive/build

[admin@hadoop1 build]$ mkdir tmp

# 常用参考资料路径

Hive地址

<http://wiki.apache.org/hadoop/Hive>

http://hive.apache.org/

Velocity地址

<http://velocity.apache.org/engine/releases/velocity-1.7/user-guide.html>

Hadoop地址

<http://hadoop.apache.org/>

<http://www.cloudera.com/>

Hadoop中文文档地址

<http://hadoop.apache.org/common/docs/r0.18.2/cn/commands_manual.html>