云计算 apache HIVE 的使用

一、 Hive 概述

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 HQL，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 没有专门的数据格式。 Hive 可以很好的工作在 Thrift 之上，控制分隔符，也允许用户指定数据格式。

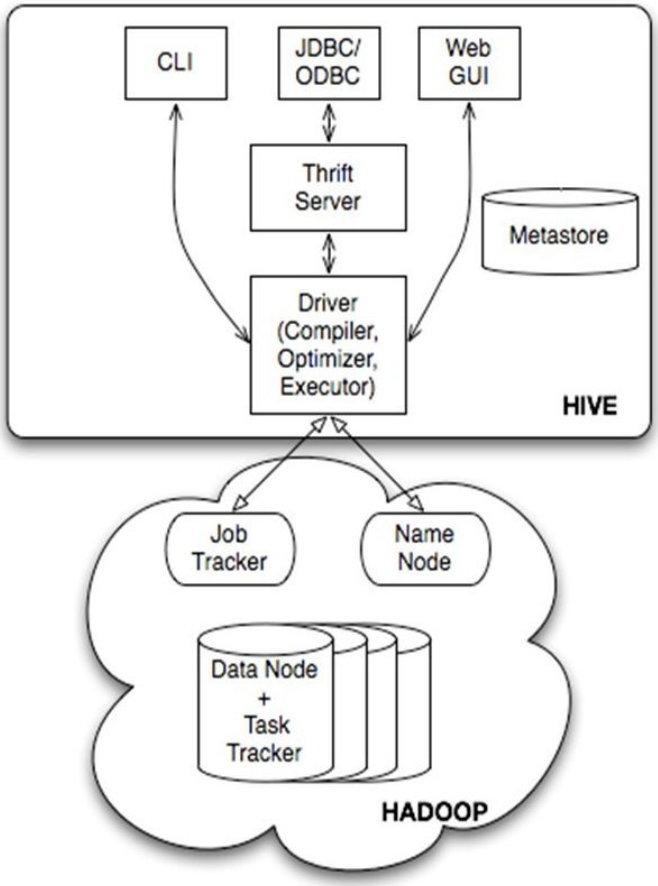
由于 Hive 采用了 SQL 的查询语言 HQL，因此很容易将 Hive 理解为数据库。其实从结构上来看，Hive 和数据库除了拥有类似的查询语言，再无类似之处。数据库可以用在 Online 的应用中，但是 Hive 是为数据仓库而设计的，清楚这一点，有助于从应用角度理解 Hive 的特性。Hive 和数据库比较大致有以下几方面的不同：

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 的体系结构



Hive 的体系结构如上图所示，Hive 主要分为以下几个部分：

- 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。

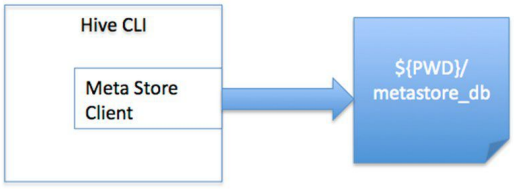
1. 用户接口主要有三个：CLI，Client 和 WUI。其中最常用的是 CLI，Cli 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在节点，并且在该节点启动 Hive Server。 WUI 是通过浏览器访问 Hive。
2. Hive 将元数据存储在数据库中，如 MySQL、derby。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
3. 解释器、编译器、优化器完成 HQL 查询语句从词法分析、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 的元数据存储

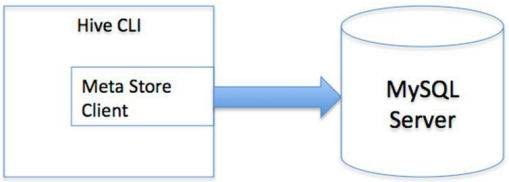
Hive 将元数据存储在 RDBMS 中，有三种模式可以连接到数据库：

- Single User Mode： 此模式连接到一个 In-memory 的数据库 Derby，一般用于 Unit Test。



Parameter	Description	Example
javax.jdo.option.ConnectionURL	JDBC connection URL along with database name containing metadata	jdbc:derby:;databaseName=metastore_db;create=true
javax.jdo.option.ConnectionDriverName	JDBC driver name. Embedded Derby for Single user mode.	org.apache.derby.jdbc.EmbeddedDriver
javax.jdo.option.ConnectionUserName	User name for Derby database	APP
javax.jdo.option.ConnectionPassword	Password	mine

- Multi User Mode：通过网络连接到一个数据库中，是最经常使用到的模式。



Parameter	Description	Example
javax.jdo.option.ConnectionURL	JDBC connection URL along with database name containing metadata	jdbc:mysql://<host name>/<database name>?createDatabaseIfNotExist=true
javax.jdo.option.ConnectionDriverName	Any JDO supported JDBC driver.	com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName	User name	
javax.jdo.option.ConnectionPassword	Password	

- Remote Server Mode：用于非 Java 客户端访问元数据库，在服务器端启动一个 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 访问元数据库。

- Server Configuration same as multi user mode client config (prev slide). To run server

```
$JAVA_HOME/bin/java -Xmx1024m -Dlog4j.configuration=file://$HIVE_HOME/conf/hms-log4j.properties
    -Djava.library.path=$HADOOP_HOME/lib/native/Linux-amd64-64/ -cp $CLASSPATH
    org.apache.hadoop.hive.metastore.HiveMetaStore
```

- Client Configuration

Parameter	Description	Example
hive.metastore.uris	Location of the metastore server	thrift://<host_name>:9083
hive.metastore.local		false

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 pvs，它在 HDFS 中的路径为：/wh/pvs，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：pvs 表中包含 ds 和 city 两个 Partition，则对应于 ds = 20090801, ctry = US 的 HDFS 子目录为：/wh/pvs/ds=20090801/ctry=US；对应于 ds = 20090801, ctry = CA 的 HDFS 子目录为；/wh/pvs/ds=20090801/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：/wh/pvs/ds=20090801/ctry=US/part-00000；hash 值为 20 的 HDFS 目录为：/wh/pvs/ds=20090801/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。

二、 基本的 SQL 语句

Hive 常用的 SQL 命令操作

创建表

```
hive> CREATE TABLE pokes (foo INT, bar STRING);
```

创建表并创建索引字段 ds

```
hive> CREATE TABLE invites (foo INT, bar STRING) PARTITIONED BY (ds STRING);
```

显示所有表

```
hive> SHOW TABLES;
```

按正条件（正则表达式）显示表，

```
hive> SHOW TABLES '.*s';
```

表添加一列

hive> ALTER TABLE pokes ADD COLUMNS (new_col INT);

添加一列并增加列字段注释

hive> ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment');

更改表名

hive> ALTER TABLE events RENAME TO 3koobecaf;

删除列

hive> DROP TABLE pokes;

元数据存储

将文件中的数据加载到表中

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

加载本地数据，同时给定分区信息

hive> LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');

加载 DFS 数据 ，同时给定分区信息

hive> LOAD DATA INPATH '/user/myname/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');

The above command will load data from an HDFS file/directory to the table. Note that loading data from HDFS will result in moving the file/directory. As a result, the operation is almost instantaneous.

SQL 操作

按先件查询

hive> SELECT a.foo FROM invites a WHERE a.ds='<DATE>';

将查询数据输出至目录

hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='<DATE>';

将查询结果输出至本地目录

hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;

选择所有列到本地目录

hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a;

hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a WHERE a.key < 100;

hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/reg_3' SELECT a.* FROM events a;

hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_4' select a.invites, a.pokes FROM profiles a;

hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT COUNT(1) FROM invites a WHERE a.ds='<DATE>';

hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT a.foo, a.bar FROM invites a;

hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/sum' SELECT SUM(a.pc) FROM pc1 a;

将一个表的统计结果插入另一个表中

hive> FROM invites a INSERT OVERWRITE TABLE events SELECT a.bar, count(1) WHERE a.foo > 0 GROUP BY a.bar;

hive> INSERT OVERWRITE TABLE events SELECT a.bar, count(1) FROM invites a WHERE a.foo > 0 GROUP BY a.bar;

JOIN

hive> FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar) INSERT OVERWRITE TABLE events SELECT t1.bar, t1.foo, t2.foo;

将多表数据插入到同一表中

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;

将文件流直接插入文件

hive> 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';

This streams the data in the map phase through the script /bin/cat (like hadoop streaming). Similarly - streaming can be used on the reduce side (please see the Hive Tutorial or examples)

实际示例

创建一个表

```
CREATE TABLE u_data (

userid INT,

movieid INT,

rating INT,

unixtime STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

STORED AS TEXTFILE;
```

下载示例数据文件，并解压缩

```
wget http://www.grouplens.org/system/files/ml-data.tar__0.gz

tar xvzf ml-data.tar__0.gz
```

加载数据到表中

```
LOAD DATA LOCAL INPATH 'ml-data/u.data'

OVERWRITE INTO TABLE u_data;
```

统计数据总量

```
SELECT COUNT(1) FROM u_data;
```

现在做一些复杂的数据分析

创建一个 weekday_mapper.py: 文件，作为数据按周进行分割

```
import sys

import datetime
```

for line in sys.stdin:

line = line.strip()

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

生成数据的周信息

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

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

使用映射脚本

//创建表，按分割符分割行中的字段值

CREATE TABLE u_data_new (

userid INT,

movieid INT,

rating INT,

weekday INT)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t';

//将 python 文件加载到系统

add FILE weekday_mapper.py;

将数据按周进行分割

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;

SELECT  weekday, COUNT(1)

FROM u_data_new

GROUP BY weekday;
```

Hive 的官方文档中对查询语言有了很详细的描述，请参考：http://wiki.apache.org/hadoop/Hive/LanguageManual ，本文的内容大部分翻译自该页面，期间加入了一些在使用过程中需要注意到的事项。

Create Table

```
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]

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。

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。

Drop Table

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。

Alter Table

Alter table 语句允许用户改变现有表的结构。用户可以增加列/分区，改变 serde，增加表和 serde 熟悉，表本身重命名。

Add Partitions

```
ALTER TABLE table_name ADD
  partition_spec [ LOCATION 'location1' ]
  partition_spec [ LOCATION 'location2' ] ...
```

partition_spec:

: PARTITION (partition_col = partition_col_value,

partition_col = partiton_col_value, ...)

用户可以用 ALTER TABLE ADD PARTITION 来向一个表中增加分区。当分区名是字符串时加引号。

```
ALTER TABLE page_view ADD
  PARTITION (dt='2008-08-08', country='us')
    location '/path/to/us/part080808'
  PARTITION (dt='2008-08-09', country='us')
    location '/path/to/us/part080809';
```

DROP PARTITION

```
ALTER TABLE table_name DROP
  partition_spec, partition_spec,...
```

用户可以用 ALTER TABLE DROP PARTITION 来删除分区。分区的元数据和数据将被一并删除。

```
ALTER TABLE page_view
    DROP PARTITION (dt='2008-08-08', country='us');
```

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 Name/Type/Position/Comment

```
ALTER TABLE table_name CHANGE [COLUMN]
  col_old_name col_new_name column_type
    [COMMENT col_comment]
    [FIRST|AFTER column_name]
```

这个命令可以允许用户修改一个列的名称、数据类型、注释或者位置。

比如：

```
CREATE TABLE test_change (a int, b int, c int);
```

ALTER TABLE test_change CHANGE a a1 INT; 将 a 列的名字改为 a1.

ALTER TABLE test_change CHANGE a a1 STRING AFTER b; 将 a 列的名字改为 a1，a 列的数据类型改为 string ,并将它放置在列 b 之后。新的表结构为：b int, a1 string, c int.

ALTER TABLE test_change CHANGE b b1 INT FIRST; 会将 b 列的名字修改为 b1, 并将它放在第一列。新表的结构为： b1 int, a string, c int.

注意：对列的改变只会修改 Hive 的元数据，而不会改变实际数据。用户应该确定保证元数据定义和实际数据结构的一致性。

Add/Replace Columns

ALTER TABLE table_name ADD|REPLACE

COLUMNS (col_name data_type [COMMENT col_comment], ...)

ADD COLUMNS 允许用户在当前列的末尾增加新的列，但是在分区列之前。

REPLACE COLUMNS 删除以后的列，加入新的列。只有在使用 native 的 SerDE（DynamicSerDe or MetadataTypeColumnsetSerDe）的时候才可以这么做。

Alter Table Properties

ALTER TABLE table_name SET TBLPROPERTIES table_properties

table_properties:

: (property_name = property_value, property_name = property_value, ...)

用户可以用这个命令向表中增加 metadata ，目前 last_modified_user ，last_modified_time 属性都是由 Hive 自动管理的。用户可以向列表中增加自己的属性。可以使用 DESCRIBE EXTENDED TABLE 来获得这些信息。

Add Serde Properties

ALTER TABLE table_name

SET SERDE serde_class_name

[WITH SERDEPROPERTIES serde_properties]

ALTER TABLE table_name

SET SERDEPROPERTIES serde_properties

serde_properties:

: (property_name = property_value,

property_name = property_value, ...)

这个命令允许用户向 SerDe 对象增加用户定义的元数据。Hive 为了序列化和反序列化数据，将会初始化 SerDe 属性，并将属性传给表的 SerDe。如此，用户可以为自定义的 SerDe 存储属性。

Alter Table File Format and Organization

```
ALTER TABLE table_name SET FILEFORMAT file_format
ALTER TABLE table_name CLUSTERED BY (col_name, col_name, ...)
  [SORTED BY (col_name, ...)] INTO num_buckets BUCKETS
```

这个命令修改了表的物理存储属性。

Loading files into table

当数据被加载至表中时，不会对数据进行任何转换。Load 操作只是将数据复制/移动至 Hive 表对应的位置。

Syntax:

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

Synopsis:

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。

SELECT

Syntax

```
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]
```

一个 SELECT 语句可以是一个 union 查询或一个子查询的一部分。

table_reference 是查询的输入，可以是一个普通表、一个视图、一个 join 或一个子查询

简单查询。例如，下面这一语句从 t1 表中查询所有列的信息。

```
SELECT * FROM t1
```

WHERE Clause

where condition 是一个布尔表达式。例如，下面的查询语句只返回销售记录大于 10，且归属地属于美国的销售代表。Hive 不支持在 WHERE 子句中的 IN，EXIST 或子查询。

```
SELECT * FROM sales WHERE amount > 10 AND region = "US"
```

ALL and DISTINCT Clauses

使用 ALL 和 DISTINCT 选项区分对重复记录的处理。默认是 ALL，表示查询所有记录。DISTINCT 表示去掉重复的记录。

```
hive> SELECT col1, col2 FROM t1
```

1 3

1 4

2 5

hive> SELECT DISTINCT col1, col2 FROM t1

1 3

1 4

2 5

hive> SELECT DISTINCT col1 FROM t1

1

2

基于 Partition 的查询

一般 SELECT 查询会扫描整个表（除非是为了抽样查询）。但是如果一个表使用 PARTITIONED BY 子句建表，查询就可以利用分区剪枝（input pruning）的特性，只扫描一个表中它关心的那一部分。Hive 当前的实现是，只有分区断言出现在离 FROM 子句最近的那个 WHERE 子句中，才会启用分区剪枝。例如，如果 page_views 表使用 date 列分区，以下语句只会读取分区为 '2008-03-01' 的数据。

```
SELECT page_views.*
    FROM page_views
    WHERE page_views.date >= '2008-03-01'
```

AND page_views.date <= '2008-03-31';

HAVING Clause

Hive 现在不支持 HAVING 子句。可以将 HAVING 子句转化为一个字查询，例如：

SELECT col1 FROM t1 GROUP BY col1 HAVING SUM(col2) > 10

可以用以下查询来表达：

SELECT col1 FROM (SELECT col1, SUM(col2) AS col2sum

FROM t1 GROUP BY col1) t2

WHERE t2.col2sum > 10

LIMIT Clause

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 查询。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

Join

Syntax

```
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 semi 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。

2. 可以 join 多于 2 个表，例如

```
SELECT a.val, b.val, c.val FROM a JOIN b
  ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
```

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

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

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

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1)
  JOIN c ON (c.key = b.key2)
```

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

join 时，每次 map/reduce 任务的逻辑是这样的：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。

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。

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)
```

三、 配置 apache HIVE 元数据 DB 为 PostgreSQL

HIVE 的元数据默认使用 derby 作为存储 DB，derby 作为轻量级的 DB，在开发、测试过程中使用比较方便，但是在实际的生产环境中，还需要考虑易用性、容灾、稳定性以及各种监控、运维工具等，这些都是 derby 缺乏的。MySQL 和 PostgreSQL 是两个比较常用的开源数据库系统，在生产环境中比较多的用来替换 derby。配置 MySQL 在网上的文章比较多，这里不再赘述，本文主要描述配置 HIVE 元数据 DB 为 PostgreSQL 的方法。

HIVE 版本：HIVE 0.7-snapshot，HIVE 0.8-snapshot

步骤 1：在 PG 中为元数据增加用户的 DB

首先在 PostgreSQL 中为 HIVE 的元数据建立帐号和 DB。

--以管理员身份登入 PG：

```
psql postgres -U postgres
```

--创建用户 hive_user:

```
Create user hive_user;
```

--创建 DB metastore_db，owner 为 hive_user:

```
Create database metastore_db with owner=hive_user;
```

--设置 hive_user 的密码：

```
/password hive_user
```

完成以上步骤以后，还要确保 PostgreSQL 的 pg_hba.conf 中的配置允许 HIVE 所在的机器 ip 可以访问 PG。

步骤 2：下载 PG 的 JDBC 驱动

在 HIVE_HOME 目录下创建 auxlib 目录：

```
mkdir auxlib
```

此时 HIVE_HOME 目录中应该有 bin，lib，auxlib，conf 等目录。

下载 PG 的 JDBC 驱动

```
Wget http://jdbc.postgresql.org/download/postgresql-9.0-801.jdbc4.jar
```

将下载到的 postgresql-9.0-801.jdbc4.jar 放到 auxlib 中。

步骤 3：修改 HIVE 配置文件

在 HIVE_HOME 中新建 hive-site.xml 文件，内容如下，蓝色字体按照 PG server 的相关信息进行修改。

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>

<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:postgresql://pg_server_ip:pg_server_port/metastore_db?</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>org.postgresql.Driver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hive_user</value>
  <description>username to use against metastore database</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>hive_user_pass</value>
  <description>password to use against metastore database</description>
</property>

</configuration>
```

步骤 4：初始化元数据表

元数据库 metastore 中默认没有表，当 HIVE 第一次使用某个表的时候，如果发现该表不存在就会自动创建。对 derby 和 mysql，这个过程没有问题，因此 derby 和 mysql 作为元数据库不需要这一步。

PostgreSQL 在初始化的时候，会遇到一些问题，导致 PG 数据库死锁。例如执行以下 HIVE 语句：

```
>Create table kv (key,int,value string) partitioned by (ds string);
OK
>Alter table kv add partition (ds = '20110101');
```

执行这一句的时候，HIVE 会一直停在这。

查看 PG 数据库，发现有两个连接在进行事务操作，其中一个是：

<IDLE> in transaction

此时处于事务中空闲，另外一个是：

ALTER TABLE "PARTITIONS" ADD CONSTRAINT "PARTITIONS_FK1" FOREIGN KEY ("SD_ID") REFERENCES "SDS" ("SD_ID") INITIALLY DEFERRED

处于等待状态。

进一步查看日志，发现大致的过程是这样的：

HIVE 发起 Alter table kv add partition (ds = '20110101')语句，此时 DataNucleus 接口发起第一个 isolation 为 SERIALIZABLE 的事务，锁定了 TBLS 等元数据表。在这个的事务进行过程中，DataNucleu 发现 PARTITIONS 等表没有，则要自动创建。于是又发起了另外一个 isolation 为 SERIALIZABLE 的事务，第一个事务变为<IDLE> in transaction。第二个事务创建了 PARTITIONS 的表后，还要给它增加约束条件，这时，它需要获得它引用的表 SDS 的排他锁，但这个锁已经被第一个事务拿到了，因此需要等待第一个事务结束。而第一个事务也在等待第二个事务结束。这样就造成了死锁。

类似的情况出现在：

```
>create test(key int);
OK
>drop table test;
```

当 drop table 时会去 drop 它的 index，而此时没有 index 元数据表，它去键，然后产生死锁。

有三种方法可以解决这个死锁问题：

第一种方法：

使用 PG 的 pg_terminate_backend()将第一个事务结束掉，这样可以保证第二个事务完成下去，将元数据表键成功。

第二种方法：

使 HIVE 将创建元数据表的过程和向元数据表中添加数据的过程分离：

```
>Create table kv (key,int,value string) partitioned by (ds string);
OK
>show partitions kv;
OK
>Alter table kv add partition (ds = '20110101');
OK
```

执行以上语句时就不会发生死锁，因为在执行 show partitions kv 语句时，它是只读语句，不会加锁。当这个语句发现 PARTITIONS 等表不在时，创建这些表不会发生死锁。

同样对于 index 表，使用

```
>Show index on kv;
```

可以将 IDXS 表建好。

第三种方法：

使用 DataNucleu 提供的 SchemaTool，将 HIVE 的 metastore/src/model/package.jdo 文件作为输入，这个工具可以自动创建元数据中的表。具体的使用方法见：

http://www.datanucleus.org/products/accessplatform_2_0/rdbms/schematool.html

小结

本文给出了使用 PostgreSQL 作为 HIVE 元数据 DB 的配置方法，以及遇到的死锁问题的解决办法，希望对使用 HIVE 和 PostgreSQL 的朋友有帮助。

四、 Hive 源码解析—之—Hive 的入口

Hive 源码解析—之—hive 的入口：

初衷：hi，大家好，我叫红门，在 hive 方面是个菜鸟，现在读 hive 源码希望能够更了解底层，尤其是 hive 与 Hadoop 切换这块。但在读 hive 源码时发现比 Hadoop 源码难读一些，虽然 Hadoop 源码量比较大，但是整体很规范，命名规范，关键地方注释的比较明确。
去年在读和修改 Hadoop 源码时都感觉比较清晰，可读性比较好一些，往往可以望文生义，可能也有自己对 hive 不熟的原因在里面吧！

想必别人应该也有人在关注 hive 底层，所以决定拿出来与大家分享，共同学习，如有理解不到位的地方，欢迎拍砖，更欢迎交流。

一直在思索应该从哪里写，不能一大堆代码粘上来，一通狂砍，大家会不知所云，有点装大。后来给自己定了个原则：

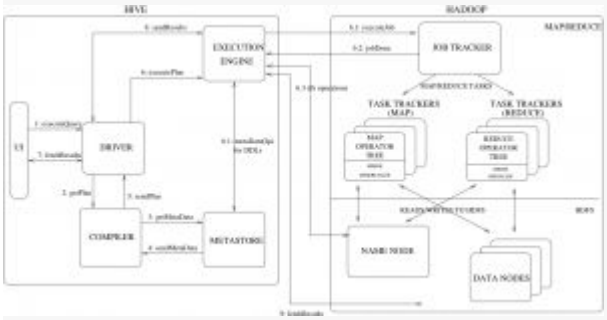
1。hive 执行过程为主线，尽量把关键的一些部分提出来，每次定一个主题。

2．怎么描述才能让别人更容易理解，更容易理清思路，尽量图形化描述。（我用的是 mindManager 画的图，不知道这种图形化的是不是可以让你看的更舒服。）

废话说得差不多了，现在开始！！

我们先从 hive 入口聊起，一路按着 hive 的执行过程主线走下来，

这次的主题就叫做： hive 的入口！！ (该图借用网上的，不知出处)：



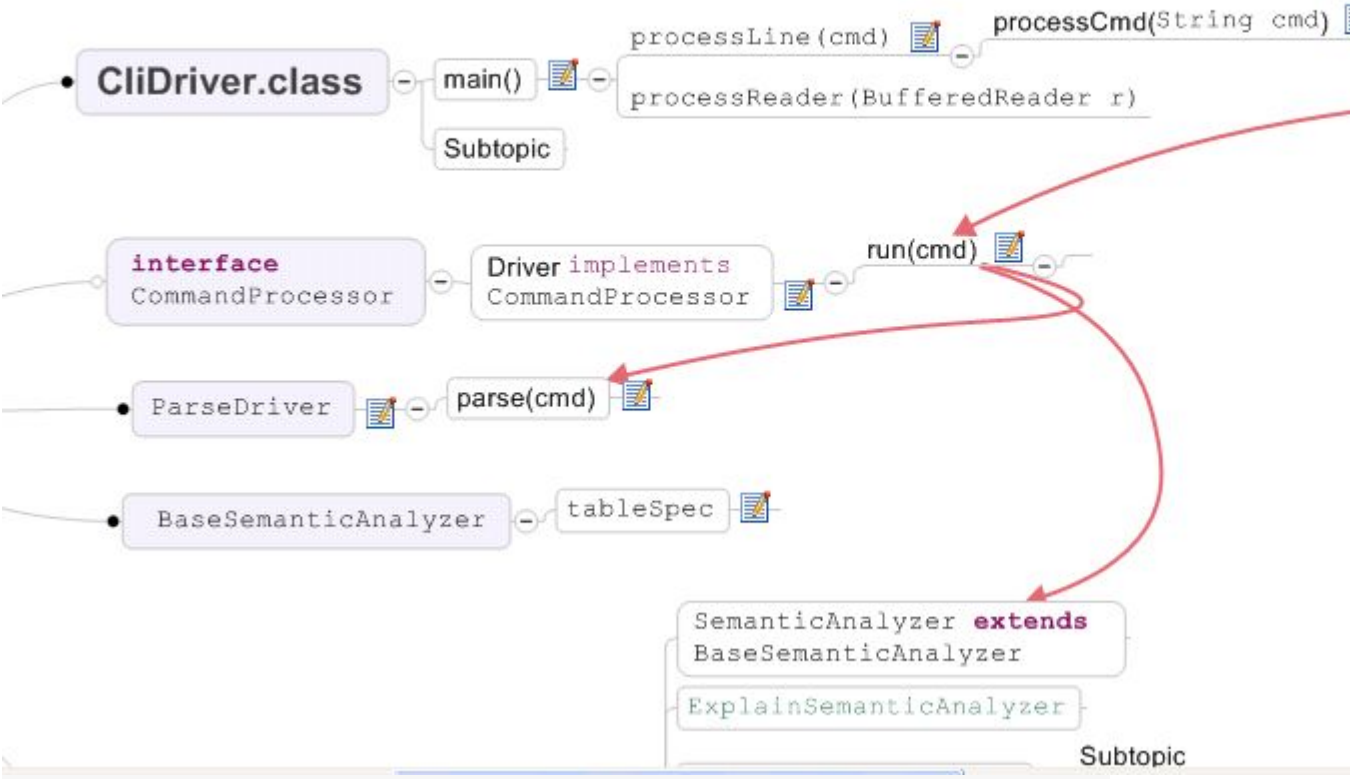
CliDriver 可以说是 hive 的入口，对应上图中的 UI 部分。大家看它的结构就可以明白了，main（）函数！对！你猜的没错就是从 main（）开始。

下图是类结构，总共有五个关键的函数。

这个类可以说是用户和 hive 交互的平台，你可以把它认为是 hive 客户端。总共有 4 个 key

函数：

下图是这个 CliDriver 类在整个 Hive 执行过程中的作用的地位。



如图，hive 执行流程_按正常步骤走：

1．—CliDriver.classz 中 main（）开始，初始化 Hive 环境变量，获取客户端提供的 string 或者 file。

2 —将其代码送入 processLine（cmd），这步主要是读入 cmd：';'之前的所有字符串都读入(不做任何检查)，之后的会忽略。读完后，传入 processCmd()处理

3 —调用 processCmd（cmd），分情况处理

//– 读入 cmd，并分情况处理，总共分为以下五种情况，根据命令的开头字符串来确定用什么方法处理。

// 1.set.. 设置 operator 参数，hive 环境参数

// 2.quit or exit — 退出 Hive 环境

// 3.! 开头

// 4.dfs 开头 交给 FsShell 处理

// 5.hivesql 正常 hivesql 执行语句，我们最关心的是这里。语句交给了、、Hive 真正的核

心引擎◊ Driver。返回 ret = Driver.run(cmd);

4．—不同情况不同处理方法。我们关心的第五种情况：正常的 HiveSQL 如何处理？其实是进入 driver.class 里面 run()，

//读入 hivesql ,词法分析，语法分析，直到执行结束

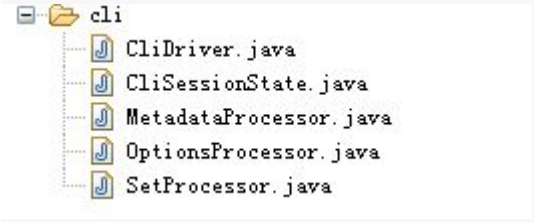
//1.ParseDriver 返回 词法树 CommonTree

//2.BaseSemanticAnalyzer sem.analyze(tree, ctx);//语义解释，生成执行计划

5．—。。。etc

今天的主题是 hive 的入口，我们只聊前三步。

现在我们细化主要函数，看 hive 实际是怎么处理的。（如果你只想了解 hive 工作流程或原理，不想拘泥于细节，可以跳过下面的细节，如果你想修改源码，做优化，可以继续往下看）



下面是 hive 入口 涉及的一些关键类和关键函数。

————————————-类 CliDriver —

由于这个类，可以说贯彻 Hive 的整个流程架构，所以我聊的比较细。

————————————————————main()

```
 public static void main(String[] args) throws IOException
{

OptionsProcessor oproc = new OptionsProcessor();
if(! oproc.process_stage1(args)) {
System.exit(1);
}

// NOTE: It is critical to do this here so that log4j is
reinitialized before
```

```
// any of the other core hive classes are loaded
SessionState.initHiveLog4j();
//建立客户端 sesssion
CliSessionState ss = new CliSessionState (new
HiveConf(SessionState.class));
ss.in = System.in;//标准输入
try {
ss.out = new PrintStream(System.out, true, "UTF-8");//??
ss.err = new PrintStream(System.err, true, "UTF-8");//??
} catch (UnsupportedEncodingException e) {
System.exit(3);
}

SessionState.start(ss);// -- start session 通过复制当前
CliSessionState 新建立 SessionState

if(! oproc.process_stage2(ss)) {
System.exit(2);
}

// set all properties specified via command line
HiveConf conf = ss.getConf();//设置所有配置属性
for(Map.Entry item: ss.cmdProperties.entrySet()) {
conf.set((String) item.getKey(), (String) item.getValue());
}

sp = new SetProcessor();//?? what is proccessor
qp = new Driver(); // 正常 hiveSql 的处理引擎
dfs = new FsShell(ss.getConf());//dfs 接口，用于 dfs 命令处理
```

```
if(ss.execString != null) {// 输入的是命令行，按命令执行
System.exit(processLine(ss.execString));
}

try {
if(ss.fileName != null) {// 输入的是文件名，读文件执行
System.exit(processReader(new BufferedReader(new
FileReader(ss.fileName))));
}
} catch (FileNotFoundException e) {//没有找到该文件
System.err.println("Could not open input file for reading.
("+e.getMessage()+")");
System.exit(3);
}

Character mask = null;
String trigger = null;

ConsoleReader reader = new ConsoleReader();//hive Console
控制台命令读取器
reader.setBellEnabled(false);
//reader.setDebug(new PrintWriter(new
FileWriter("writer.debug", true)));

List completors = new LinkedList();
completors.add(new SimpleCompletor(new String[] { "set",
"from",
"create", "load",
"describe", "quit", "exit" }));
reader.addCompletor(new ArgumentCompletor(completors));

String line;
```

```
PrintWriter out = new PrintWriter(System.out);
    final String HISTORYFILE = ".hivehistory";//建立历史文件，记录所有的命令行
    String historyFile = System.getProperty("user.home") + File.separator  + HISTORYFILE;
    reader.setHistory(new History(new File(historyFile)));
    in  et = 0
```

——————————— processLine(Cmd)

// 读入 cmd：‘;’之前的所有字符串都读入(不做任何检查)，之后的都会忽略。读完后，传入 processCmd 处理.

```
 public static int processLine(String line) {
int ret = 0;
for(String oneCmd: line.split(";")) {
oneCmd = oneCmd.trim();
if(oneCmd.equals(""))
continue;

ret = processCmd(oneCmd);//--执行命令
if(ret != 0) {
// ignore anything after the first failed command
return ret;
}
}
return 0;
}
```

——————————— processCmd（）

//– 读入 cmd，并分情况处理，总共分为以下五种情况，根据命令的开头字符串来确定用什么方法处理。

// 1.set.. 设置 operator 参数，hive 环境参数

```
// 2.quit or exit — 退出 Hive 环境
// 3.! 开头
// 4.dfs 开头 交给 FsShell 处理
// 5.hivesql 正常 hivesql 执行语句，我们最关心的是这里。语句交给了、、Hive 真正的核心引

public static int processCmd(String cmd) {
String[] tokens = cmd.split("\\s+");
String cmd_1 = cmd.substring(tokens[0].length());
int ret = 0;

if(tokens[0].equals("set")) { //1
ret = sp.run(cmd_1);// 调用这句就可以更改 hadoop 配置
} else if (cmd.equals("quit") || cmd.equals("exit")) {//2
//退出 Hive 环境
System.exit(0);
} else if (cmd.startsWith("!")) {//3 : ! 开头的命令
SessionState ss = SessionState.get();
String shell_cmd = cmd.substring(1);
if (shell_cmd.endsWith(";")) {
shell_cmd = shell_cmd.substring(0, shell_cmd.length()-1);
}//--除掉';'??
//shell_cmd = "/bin/bash -c \'" + shell_cmd + "\'";

try {
Process executor =
Runtime.getRuntime().exec(shell_cmd);//!!??这句得好好 跟踪
StreamPrinter outPrinter = new
StreamPrinter(executor.getInputStream(), null, ss.out);
StreamPrinter errPrinter = new
StreamPrinter(executor.getErrorStream(), null, ss.err);
```

```
outPrinter.start();
errPrinter.start();

int exitVal = executor.waitFor();//?? look executor
if (exitVal != 0) {
ss.err.write((new String("Command failed with exit code = "
+ exitVal)).getBytes());
}
}
catch (Exception e) {
e.printStackTrace();
}
} else if (cmd.startsWith("dfs")) {//4 "dfs" 开头解析方法 --
cmd.
// Hadoop DFS 操作接口 处理!
SessionState ss = SessionState.get();
if(dfs == null)
dfs = new FsShell(ss.getConf());
String hadoopCmd = cmd.replaceFirst("dfs\\s+", "");
hadoopCmd = hadoopCmd.trim();
if (hadoopCmd.endsWith(";")) {
hadoopCmd = hadoopCmd.substring(0, hadoopCmd.length()-1);
}
String[] args = hadoopCmd.split("\\s+");//
try {
PrintStream oldOut = System.out;
System.setOut(ss.out);
int val = dfs.run(args);//??
System.setOut(oldOut);
if (val != 0) {
```

```
ss.err.write((new String("Command failed with exit code = "
+ val)).getBytes());
}
} catch (Exception e) {
ss.err.println("Exception raised from DFSShell.run " +
e.getLocalizedMessage());
}
} else {//5 hivesql 正常运行，重点在这里

ret = qp.run(cmd);//正常执行 hive 命令,如: select .. ;

addfile ..;
Vector res = new Vector();
while (qp.getResults(res)) {//获得执行结果 result

for (String r:res) {
SessionState ss = SessionState.get();
PrintStream out = ss.out;
out.println(r);
}
res.clear();
}

int cret = qp.close();
if (ret == 0) {
ret = cret;
}
}
return ret;
}
```

————————类 CliSessionState

CliSessionState 除了做了个初始化，基本都是上继承了 SessionState 的实现方法，可能是作者为了低耦合。

public class CliSessionState extends SessionState

所以我们直接看 SessionState。

SessionState 可以说你是你自己当前的 Hive 环境，建立、初始化你 Hive 的 session，一方面它来自 conf 的初始化设置，一方面来自你手动 set。可以通过命令行形式，也可以通过 file，这都取决于你的选择。

它会连接 Hive 元数据数据库，得到现有的元数据信息。

此类主要关键功能：

1 主要是生成 session，并赋予一个唯一 id(设置规则：用户名_年月日分秒 即 user_id + "_"+ yyyymmddHHmm)的 session，

生成 session 有两种方式：1.直接新建 session ， 2. 通过拷贝方式复制一个 session。

第一种我们最常用，但第二种很有用，我们可以确保我们两个环境是完全一致的，而且避免琐碎设置工作。

2 给每个 cmd 给予一个 queryID，可以通过 queryID 得到命令行，也可以反过来得到 id

3 每个 sessionState 都会有一个 logHelper，用于日志记录

其中 clude ： hiveconf，连接 db 元数据数据库

下图是 SessionState 的类结构：

关键函数：

String makeSessionId() ——— //生成 sessionID ： user_id+"_" + yyyyMMDDhhmm

setCmd(String cmdString)—— //给命令 cmd 设置 query Id

protected final static HiveConf.ConfVars [] metaVars —-//获取元数据系统，路径等

public String getCmd() —— // –通过 queryID 获取命令代码 cmd

————————————类 CommandProcessor

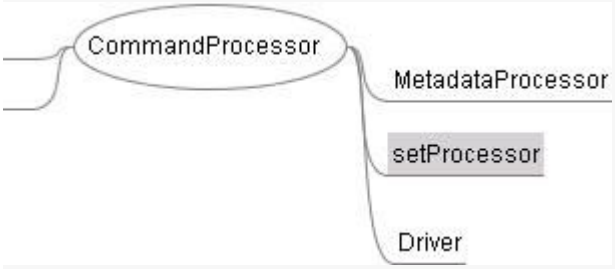
CommandProcessor 类的很简单，是个接口类。

public interface CommandProcessor {

public int run(String command);

}

你会奇怪为什么先聊这个接口类，因为有三个类实现了这个接口（如下图），其中 setProcessor, MetadataProcessor 是我们 Hive 入口的关键类。



————类 MetadataProcessor

Hive 部分元信息提取与处理

// run()中 得到表的元信息，如果出错返回 1，如：找不到表名等情况

```
public int run(String command) {
SessionState ss = SessionState.get();
String table_name = command.trim();
if(table_name.equals("")) {
return 0;
}

try {
MetaStoreClient msc = new MetaStoreClient(ss.getConf());

if(!msc.tableExists(table_name)) {//表不存在
ss.err.println("table does not exist: " + table_name);
return 1;
} else {
List fields = msc.get_fields(table_name);//获得表信息

for(FieldSchema f: fields) {
ss.out.println(f.getName() + ": " + f.getType());
}
}
```

```
} catch (MetaException err) {
ss.err.println("Got meta exception: " + err.getMessage());
return 1;
} catch (Exception err) {
ss.err.println("Got exception: " + err.getMessage());
return 1;
}
return 0;
}
```

————————- 类 SetProcessor

主要是设置 Hive 环境，总共分为两大类：

1. set session 为安全模式，

如：set silent = true;

2.set 该 session 的 conf 配置，即调用 hadoop 时的配置参数，以及改变执行时的具体实现。

如：set hive.exec.compress.output='false';

// 我们可以调用这里 run(String command)更改 hadoop 配置 ， hive 执行参数等，

```
public int run(String command) {
SessionState ss = SessionState.get();//建一个 SessionState
对象
String nwcmd = command.trim();//去空格
if(nwcmd.equals("")) {
dumpOptions(ss.getConf().getChangedProperties());
return 0;
}
if(nwcmd.equals("-v")) {
dumpOptions(ss.getConf().getAllProperties());
return 0;
}
String[] part = new String [2];
int eqIndex = nwcmd.indexOf('=');
if(eqIndex == -1) {
```

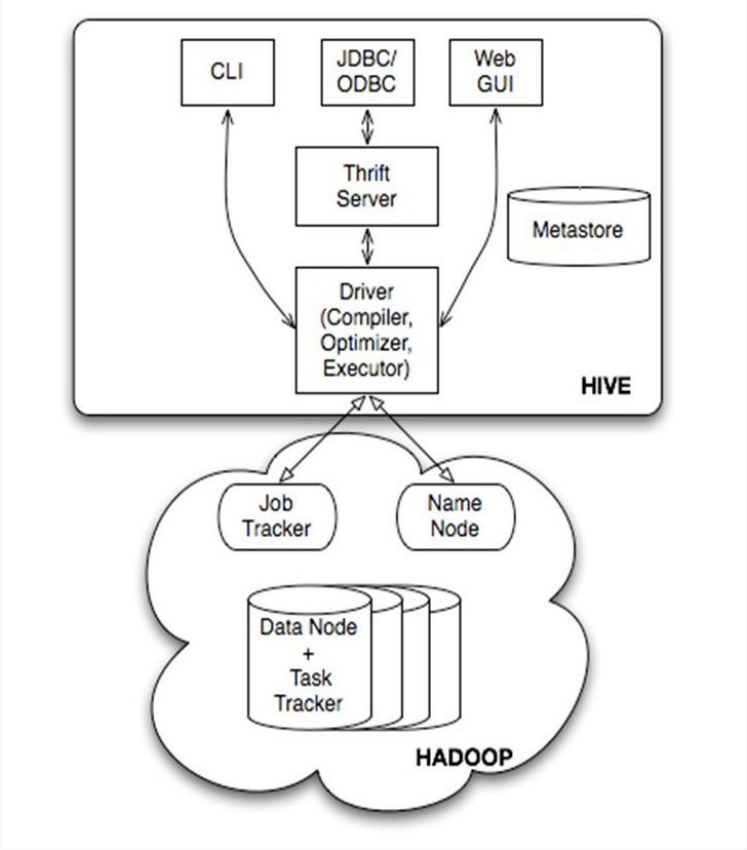
```
// no equality sign - print the property out
dumpOption(ss.getConf().getAllProperties(), nwcmd);
return (0);
} else if (eqIndex == nwcmd.length()-1) {
part[0] = nwcmd.substring(0, nwcmd.length()-1);
part[1] = "";
} else {
part[0] = nwcmd.substring(0, eqIndex);// 中间=号隔开的 set cmd
part[1] = nwcmd.substring(eqIndex+1);
}

try {//
if (part[0].equals("silent")) {// 设置 silent 模式
boolean val = getBoolean(part[1]);//
ss.setIsSilent(val);//
} else {
ss.getConf().set(part[0], part[1]);// 设置 key - value
（如: .gmmt = ture）修改该 session 的 conf 里配置
}
```

Hive 的入口先到这里，以后会陆续更新，欢迎交流。希望能这个 Hive 源码分析系列能够在你探索 Hive 的路上，节省您的时间。谢谢~

五、 Hive 随谈（二）– Hive 结构

Hive 体系结构

Hive 的结构如图所示，



主要分为以下几个部分：

- 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。

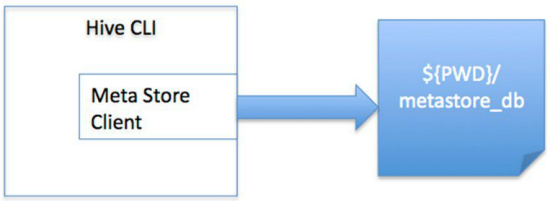
1. 用户接口主要有三个：CLI，Client 和 WUI。其中最常用的是 CLI，Cli 启动的时候，会同时启动一个 Hive 副本。Client 是 Hive 的客户端，用户连接至 Hive Server。在启动 Client 模式的时候，需要指出 Hive Server 所在节点，并且在该节点启动 Hive Server。 WUI 是通过浏览器访问 Hive。
2. Hive 将元数据存储在数据库中，如 mysql、derby。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。
3. 解释器、编译器、优化器完成 HQL 查询语句从词法分析、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 元数据存储

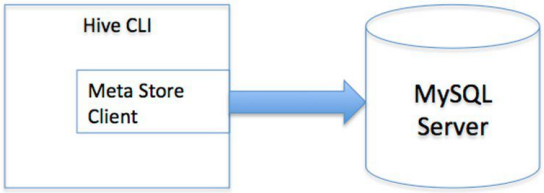
Hive 将元数据存储在 RDBMS 中，有三种模式可以连接到数据库：

- Single User Mode： 此模式连接到一个 In-memory 的数据库 Derby，一般用于 Unit Test。



Parameter	Description	Example
javax.jdo.option.ConnectionURL	JDBC connection URL along with database name containing metadata	jdbc:derby:;databaseName=metastore_db;create=true
javax.jdo.option.ConnectionDriverName	JDBC driver name. Embedded Derby for Single user mode.	org.apache.derby.jdbc.EmbeddedDriver
javax.jdo.option.ConnectionUserName	User name for Derby database	APP
javax.jdo.option.ConnectionPassword	Password	mine

- Multi User Mode：通过网络连接到一个数据库中，是最经常使用到的模式。



Parameter	Description	Example
javax.jdo.option.ConnectionURL	JDBC connection URL along with database name containing metadata	jdbc:mysql://<host name>/<database name>?createDatabaseIfNotExist=true
javax.jdo.option.ConnectionDriverName	Any JDO supported JDBC driver.	com.mysql.jdbc.Driver
javax.jdo.option.ConnectionUserName	User name	
javax.jdo.option.ConnectionPassword	Password	

- Remote Server Mode：用于非 Java 客户端访问元数据库，在服务器端启动一个 MetaStoreServer，客户端利用 Thrift 协议通过 MetaStoreServer 访问元数据库。

- Server Configuration same as multi user mode client config (prev slide). To run server

```
$JAVA_HOME/bin/java -Xmx1024m -Dlog4j.configuration=file://$HIVE_HOME/conf/hms-log4j.properties
    -Djava.library.path=$HADOOP_HOME/lib/native/Linux-amd64-64/ -cp $CLASSPATH
    org.apache.hadoop.hive.metastore.HiveMetaStore
```

- Client Configuration

Parameter	Description	Example
hive.metastore.uris	Location of the metastore server	thrift://<host_name>:9083
hive.metastore.local		false

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 pvs，它在 HDFS 中的路径为：/wh/pvs，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：pvs 表中包含 ds 和 city 两个 Partition，则对应于 ds = 20090801, ctry = US 的 HDFS 子目录为：/wh/pvs/ds=20090801/ctry=US；对应于 ds = 20090801, ctry = CA 的 HDFS 子目录为；/wh/pvs/ds=20090801/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：/wh/pvs/ds=20090801/ctry=US/part-00000；hash 值为 20 的 HDFS 目录为：/wh/pvs/ds=20090801/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
-
-

六、 Hive 随谈（三）– Hive 和数据库的异同

摘要：由于 Hive 采用了 SQL 的查询语言 HQL，因此很容易将 Hive 理解为数据库。其实
从结构上来看，Hive 和数据库除了拥有类似的查询语言，再无类似之处。本文将
从多个方面来阐述 Hive 和数据库的差异。数据库可以用在 Online 的应用中，但是
Hive 是为数据仓库而设计的，清楚这一点，有助于从应用角度理解 Hive 的特性。

Hive 和数据库的比较	HQL	SQL
查询语言		
数据存储位置	HDFS	Raw Device 或者 Local FS
数据格式	用户定义	系统决定
数据更新	支持	不支持
索引	无	有
执行	MapRedcue	Executor
执行延迟	高	低
可扩展性	高	低
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。

七、 Hive 随谈（四）– Hive QL

Hive 的官方文档中对查询语言有了很详细的描述，请参考：http://wiki.apache.org/hadoop/Hive/LanguageManual ，本文的内容大部分翻译自该页面，期间加入了一些在使用过程中需要注意到的事项。

Create Table

```
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]
```

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。

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。

Drop Table

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。

Alter Table

Alter table 语句允许用户改变现有表的结构。用户可以增加列/分区，改变 serde，增加表和 serde 熟悉，表本身重命名。

Add Partitions

```
ALTER TABLE table_name ADD
partition_spec [ LOCATION 'location1' ]
partition_spec [ LOCATION 'location2' ] ...
```

```
partition_spec:
: PARTITION (partition_col = partition_col_value,
partition_col = partiton_col_value, ...)
```

用户可以用 ALTER TABLE ADD PARTITION 来向一个表中增加分区。当分区名是字符串时加引号。

```
 ALTER TABLE page_view ADD
PARTITION (dt='2008-08-08', country='us')
location '/path/to/us/part080808'
PARTITION (dt='2008-08-09', country='us')
location '/path/to/us/part080809';
```

DROP PARTITION

```
ALTER TABLE table_name DROP
partition_spec, partition_spec,...
```

用户可以用 ALTER TABLE DROP PARTITION 来删除分区。分区的元数据和数据将被一并删除。

```
ALTER TABLE page_view
DROP PARTITION (dt='2008-08-08', country='us');
```

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 Name/Type/Position/Comment

```
ALTER TABLE table_name CHANGE [COLUMN]
col_old_name col_new_name column_type
[COMMENT col_comment]
[FIRST|AFTER column_name]
```

这个命令可以允许用户修改一个列的名称、数据类型、注释或者位置。

比如：

```
CREATE TABLE test_change (a int, b int, c int);
```

```
ALTER TABLE test_change CHANGE a a1 INT;
```
将 a 列的名字改为 a1.

```
ALTER TABLE test_change CHANGE a a1 STRING AFTER b;
```
将 a 列的名字改为 a1，a 列的数据类型改为 string，并将它放置在列 b 之后。新的表结构为： b int, a1 string, c int.

```
ALTER TABLE test_change CHANGE b b1 INT FIRST;
```
会将 b 列的名字修改为 b1，并将它放在第一列。新表的结构为： b1 int, a string, c int.

注意：对列的改变只会修改 Hive 的元数据，而不会改变实际数据。用户应该确定保证元数据定义和实际数据结构的一致性。

Add/Replace Columns

```
ALTER TABLE table_name ADD|REPLACE
COLUMNS (col_name data_type [COMMENT col_comment], ...)
```

ADD COLUMNS 允许用户在当前列的末尾增加新的列，但是在分区列之前。

REPLACE COLUMNS 删除以后的列，加入新的列。只有在使用 native 的 SerDE（DynamicSerDe or MetadataTypeColumnsetSerDe）的时候才可以这么做。

Alter Table Properties

```
ALTER TABLE table_name SET TBLPROPERTIES table_properties
table_properties:
: (property_name = property_value, property_name =
property_value, ... )
```

用户可以用这个命令向表中增加 metadata，目前 last_modified_user，last_modified_time 属性都是由 Hive 自动管理的。用户可以向列表中增加自己的属性。可以使用 DESCRIBE EXTENDED TABLE 来获得这些信息。

Add Serde Properties

```
ALTER TABLE table_name
SET SERDE serde_class_name
[WITH SERDEPROPERTIES serde_properties]

ALTER TABLE table_name
SET SERDEPROPERTIES serde_properties

serde_properties:
: (property_name = property_value,
property_name = property_value, ... )
```

这个命令允许用户向 SerDe 对象增加用户定义的元数据。Hive 为了序列化和反序列化数据，将会初始化 SerDe 属性，并将属性传给表的 SerDe。如此，用户可以为自定义的 SerDe 存储属性。

Alter Table File Format and Organization

```
ALTER TABLE table_name SET FILEFORMAT file_format
ALTER TABLE table_name CLUSTERED BY (col_name, col_name, ...)
[SORTED BY (col_name, ...)] INTO num_buckets BUCKETS
```

这个命令修改了表的物理存储属性。

Loading files into table

当数据被加载至表中时，不会对数据进行任何转换。Load 操作只是将数据复制/移动至 Hive 表对应的位置。

Syntax:

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

Synopsis:

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。

SELECT

Syntax

```
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]
```

- 一个 SELECT 语句可以是一个 union 查询或一个子查询的一部分。
- table_reference 是查询的输入，可以是一个普通表、一个视图、一个 join 或一个子查询
- 简单查询。例如，下面这一语句从 t1 表中查询所有列的信息。

```
SELECT * FROM t1
```

WHERE Clause

where condition 是一个布尔表达式。例如，下面的查询语句只返回销售记录大于 10，且归属地属于美国的销售代表。Hive 不支持在 WHERE 子句中的 IN，EXIST 或子查询。

```
SELECT * FROM sales WHERE amount > 10 AND region = "US"
```

ALL and DISTINCT Clauses

使用 ALL 和 DISTINCT 选项区分对重复记录的处理。默认是 ALL，表示查询所有记录。DISTINCT 表示去掉重复的记录。

```
hive> SELECT col1, col2 FROM t1
1 3
```

```
1 3
1 4
2 5
hive> SELECT DISTINCT col1, col2 FROM t1
1 3
1 4
2 5
hive> SELECT DISTINCT col1 FROM t1
1
2
```

基于 Partition 的查询

一般 SELECT 查询会扫描整个表（除非是为了抽样查询）。但是如果一个表使用 PARTITIONED BY 子句建表，查询就可以利用分区剪枝（input pruning）的特性，只扫描一个表中它关心的那一部分。Hive 当前的实现是，只有分区断言出现在离 FROM 子句最近的那个 WHERE 子句中，才会启用分区剪枝。例如，如果 page_views 表使用 date 列分区，以下语句只会读取分区为‘2008-03-01’的数据。

```
 SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01'
     AND page_views.date <= '2008-03-31';
```

HAVING Clause

Hive 现在不支持 HAVING 子句。可以将 HAVING 子句转化为一个字查询，例如：

```
SELECT col1 FROM t1 GROUP BY col1 HAVING SUM(col2) > 10
```

可以用以下查询来表达：

```
SELECT col1 FROM (SELECT col1, SUM(col2) AS col2sum
FROM t1 GROUP BY col1) t2
WHERE t2.col2sum > 10
```

LIMIT Clause

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 查询。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
```

Join

Syntax

```
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 semi 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。

2. 可以 join 多于 2 个表，例如

```
 SELECT a.val, b.val, c.val FROM a JOIN b
ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
```

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

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

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

```
SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1)
JOIN c ON (c.key = b.key2)
```

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

join 时，每次 map/reduce 任务的逻辑是这样的：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。

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。

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)
```

八、 Hive 随谈（五）– Hive 优化

Hive 针对不同的查询进行了优化，优化可以通过配置进行控制，本文将介绍部分优化的策略以及优化控制选项。

列裁剪（Column Pruning）

在读数据的时候，只读取查询中需要用到的列，而忽略其他列。例如，对于查询：

```
SELECT a,b FROM T WHERE e < 10;
```

其中，T 包含 5 个列 (a,b,c,d,e)，列 c，d 将会被忽略，只会读取 a, b, e 列

这个选项默认为真： **hive.optimize.cp = true**

分区裁剪（Partition Pruning）

在查询的过程中减少不必要的分区。例如，对于下列查询：

```
SELECT * FROM (SELECT c1, COUNT(1)
FROM T GROUP BY c1) subq
WHERE subq.prtn = 100;

SELECT * FROM T1 JOIN
(SELECT * FROM T2) subq ON (T1.c1=subq.c2)
WHERE subq.prtn = 100;
```

会在子查询中就考虑 subq.prtn = 100 条件，从而减少读入的分区数目。

此选项默认为真：**hive.optimize.pruner=true**

Join

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：

Hive QL – Map Join

page_view

pageid	userid	time
1	111	9:08:01
2	111	9:08:13
1	222	9:08:14

Hash table

key	value
111	<1,2>
222	<2>

pv_users

Pageid	age
1	25
2	25
1	32

user

userid	age	gender
111	25	female
222	32	male

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**

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。

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

九、 Hive 随谈（六）– Hive 的扩展特性

Hive 是一个很开放的系统，很多内容都支持用户定制，包括：

- 文件格式：Text File，Sequence File
- 内存中的数据格式： Java Integer/String, Hadoop IntWritable/Text
- 用户提供的 map/reduce 脚本：不管什么语言，利用 stdin/stdout 传输数据
- 用户自定义函数: Substr, Trim, 1 – 1
- 用户自定义聚合函数: Sum, Average…… n – 1

File Format

	TextFile	SequenceFIle	RCFFile
Data type	Text Only	Text/Binary	Text/Binary
Internal Storage Order	Row-based	Row-based	Column-based
Compression	File Based	Block Based	Block Based
Splitable	YES	YES	YES
Splitable After Compression	No	YES	YES

```
 CREATE TABLE mylog ( user_id BIGINT, page_url STRING,
unix_time INT)
STORED AS TEXTFILE;
```

当用户的数据文件格式不能被当前 Hive 所识别的时候，可以自定义文件格式。可以参考 contrib/src/java/org/apache/hadoop/hive/contrib/fileformat/base64 中的例子。写完自定义的格式后，在创建表的时候指定相应的文件格式就可以：

```
 CREATE TABLE base64_test(col1 STRING, col2 STRING)
STORED AS
INPUTFORMAT 'org.apache.hadoop.hive.contrib.
fileformat.base64.Base64TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.contrib.
fileformat.base64.Base64TextOutputFormat';
```

SerDe

SerDe 是 Serialize/Deserilize 的简称，目的是用于序列化和反序列化。序列化的格式包括：

- 分隔符（tab、逗号、CTRL-A）
- Thrift 协议

反序列化（内存内）：

- Java Integer/String/ArrayList/HashMap
- Hadoop Writable 类
- 用户自定义类

目前存在的 Serde 见下图：

	LazySimpleSerDe	LazyBinarySerDe (HIVE-640)	BinarySortable SerDe
serialized format	delimited	proprietary binary	proprietary binary sortable*
deserialized format	LazyObjects*	LazyBinaryObjects*	Writable
	ThriftSerDe (HIVE-706)	**RegexSerDe**	**ColumnarSerDe**
serialized format	Depends on the Thrift Protocol	Regex formatted	proprietary column-based
deserialized format	User-defined Classes, Java Primitive Objects	ArrayList<String>	LazyObjects*

其中，LazyObject 只有在访问到列的时候才进行反序列化。 BinarySortable：保留了排序的二进制格式。

当存在以下情况时，可以考虑增加新的 SerDe：

- 用户的数据有特殊的序列化格式，当前的 Hive 不支持，而用户又不想在将数据加载至 Hive 前转换数据格式。
- 用户有更有效的序列化磁盘数据的方法。

用户如果想为 Text 数据增加自定义 Serde ，可以参照 contrib/src/java/org/apache/hadoop/hive/contrib/serde2/RegexSerDe.java 中的例子。RegexSerDe 利用用户提供的正则表倒是来反序列化数据，例如：

```
 CREATE TABLE apache_log(
host STRING,
identity STRING,
user STRING,
time STRING,
request STRING,
status STRING,
size STRING,
```

```
referer STRING,
agent STRING)
ROW FORMAT
SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES
( "input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\])
([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^
\"]*|\"[^\"]*\")
([^ \"]*|\"[^\"]*\"))?",
"output.format.string" =
"%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s";)
STORED AS TEXTFILE;
```

用户如果想为 Binary 数据增加自定义的 SerDE，可以参考例子：serde/src/java/org/apache/hadoop/hive/serde2/binarysortable，例如：

```
 CREATE TABLE mythrift_table
ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.thrift.ThriftSerDe
'
WITH SERDEPROPERTIES (
"serialization.class" =
"com.facebook.serde.tprofiles.full",
"serialization.format" =
"com.facebook.thrift.protocol.TBinaryProtocol";);
```

Map/Reduce 脚本（Transform）

用户可以自定义 Hive 使用的 Map/Reduce 脚本，比如：

```
 FROM (
SELECT TRANSFORM(user_id, page_url, unix_time)
USING 'page_url_to_id.py'
AS (user_id, page_id, unix_time)
FROM mylog
```

```
DISTRIBUTE BY user_id
SORT BY user_id, unix_time)
mylog2
SELECT TRANSFORM(user_id, page_id, unix_time)
USING 'my_python_session_cutter.py' AS (user_id,
session_info);
```

Map/Reduce 脚本通过 stdin/stdout 进行数据的读写，调试信息输出到 stderr。

UDF（User-Defined-Function）

用户可以自定义函数对数据进行处理，例如：

```
 add jar build/ql/test/test-udfs.jar;
CREATE TEMPORARY FUNCTION testlength
AS 'org.apache.hadoop.hive.ql.udf.UDFTestLength';

SELECT testlength(src.value) FROM src;

DROP TEMPORARY FUNCTION testlength;
```

UDFTestLength.java 为：

```
 package org.apache.hadoop.hive.ql.udf;

public class UDFTestLength extends UDF {
public Integer evaluate(String s) {
if (s == null) {
return null;
}
return s.length();
}
}
```

自定义函数可以重载：

```
 add jar build/contrib/hive_contrib.jar;
CREATE TEMPORARY FUNCTION example_add
AS
'org.apache.hadoop.hive.contrib.udf.example.UDFExampleAdd
';

SELECT example_add(1, 2) FROM src;
SELECT example_add(1.1, 2.2) FROM src;
```

UDFExampleAdd.java:

```
 public class UDFExampleAdd extends UDF {
public Integer evaluate(Integer a, Integer b) {
if (a = null || b = null)
return null;
return a + b;
}

public Double evaluate(Double a, Double b)       = null || b
= null)
return null;
return a + b;
}
}
```

%%

在使用 UDF 的时候，会自动进行类型转换，这个 java 或者 C 中的类型转换有些类似，比如:

```
 SELECT example_add(1, 2.1) FROM src;
```

的结果是 3.1，这是因为 UDF 将类型为 Int 的参数“1”转换为 double。

类型的隐式转换是通过 UDFResolver 来进行控制的，并且可以根据不同的 UDF 进行不同的控制。

UDF 还可以支持变长的参数，例如 UDFExampleAdd.java：

```
 public class UDFExampleAdd extends UDF {
public Integer evaluate(Integer... a) {
int total = 0;
for (int i=0; i<a.length; i++)
if (a[i] != null) total += a[i];

return total;
} // the same for Double public Double evaluate(Double... a) }
```

使用例子为：

```
 SELECT example_add(1, 2) FROM src;
  SELECT example_add(1, 2, 3) FROM src;
  SELECT example_add(1, 2, 3, 4.1) FROM src;
```

综上，UDF 具有以下特性：

- 用 java 写 UDF 很容易。
- Hadoop 的 Writables/Text 具有较高性能。
- UDF 可以被重载。
- Hive 支持隐式类型转换。
- UDF 支持变长的参数。
- genericUDF 提供了较好的性能（避免了反射）。

UDAF（User-Defined Aggregation Funcation）

例子：

```
 SELECT page_url, count(1), count(DISTINCT user_id) FROM
mylog;
```

UDAFCount.java:

```
 public class UDAFCount extends UDAF {
```

```
public static class Evaluator implements UDAFEvaluator {
private int mCount;

public void init() {
mcount = 0;
}

public boolean iterate(Object o) {
if (o!=null)
mCount++;

return true;
}

public Integer terminatePartial() {
return mCount;
}

public boolean merge(Integer o) {
mCount += o;
return true;
}

public Integer terminate() {
return mCount;
}
}
```

UDAF 总结：

- 编写 UDAF 和 UDF 类似
- UDAF 可以重载
- UDAF 可以返回复杂类
- 在使用 UDAF 的时候可以禁止部分聚合功能

UDF，UDAF 和 MR 脚本的对比：

	UDF/UDAF	M/R scripts
language	Java	any language
data format	in-memory objects	serialized streams
1/1 input/output	supported via UDF	supported
n/1 input/output	supported via UDAF	supported
1/n input/output	not supported yet (UDTF)	supported
Speed	faster	Slower

十、 基于 Hive 的日志数据统计实战

一、Hive 简介 Hive 是一个基于 hadoop 的开源数据仓库工具，用于存储和处理海量结构化数据。 它把海量数据存储于 hadoop 文件系统，而不是数据库，但提供了一套类数据库的数据存储和处理机制

一、Hive 简介

Hive 是一个基于 hadoop 的开源数据仓库工具，用于存储和处理海量结构化数据。 它把海量数据存储于 hadoop 文件系统，而不是数据库，但提供了一套类数据库的数据存储和处理机制，并采用 HQL （类 SQL ）语言对这些数据进行自动化管理和处理。我们可以把 hive 中海量结构化数据看成一个个的表，而实际上这些数据是分布式存储在 HDFS 中的。 Hive 经过对语句进行解析和转换，最终生成一系列基于 hadoop 的 map/reduce 任务，通过执行这些任务完成数据处理。

Hive 诞生于 facebook 的日志分析需求，面对海量的结构化数据， hive 以较低的成本完成了以往需要大规模数据库才能完成的任务，并且学习门槛相对较低，应用开发灵活而高效。

Hive 自 2009.4.29 发布第一个官方稳定版 0.3.0 至今，不过一年的时间，正在慢慢完善，网上能找到的相关资料相当少，尤其中文资料更少，本文结合业务对 hive 的应用做了一些探索，并把这些经验做一个总结，所谓前车之鉴，希望读者能少走一些弯路。

Hive 的官方 wiki 请参考这里 ：

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

官方主页在这里：

http://hadoop.apache.org/hive/

Hive-0.5.0 源码包和二进制发布包的下载地址

http://labs.renren.com/apache-mirror/hadoop/hive/hive-0.5.0/

二、部署

由于 Hive 是基于 hadoop 的工具，所以 hive 的部署需要一个正常运行的 hadoop 环境。以下介绍 hive 的简单部署和应用。

部署环境：

操作系统： Red Hat Enterprise Linux AS release 4 (Nahant Update 7)

Hadoop ： hadoop-0.20.2 ，正常运行

部署步骤如下：

1、下载最新版本发布包 hive-0.5.0-dev.tar.gz ，传到 hadoop 的 namenode 节点上，解压得到 hive 目录。假设路径为： /opt/hadoop/hive-0.5.0-bin

2、设置环境变量 HIVE_HOME ，指向 hive 根目录 /opt/hadoop/hive-0.5.0-bin 。由于 hadoop 已运行，检查环境变量 JAVA_HOME 和 HADOOP_HOME 是否正确有效。

3、切换到 $HIVE_HOME 目录， hive 配置默认即可，运行 bin/hive 即可启动 hive ，如果正常启动，将会出现“ hive> ”提示符。

4、在命令提示符中输入“ show tables; ”，如果正常运行，说明已部署成功，可供使用。

常见问题：

1、执行“ show tables; ”命令提示“ FAILED: Error in metadata: java.lang.IllegalArgumentException: URI: does not have a scheme ”，这是由于 hive 找不到存放元数据库的数据库而导致的，修改 conf/ hive-default.xml 配置文件中的 hive.metastore.local 为 true 即可。由于 hive 把结构化数据的元数据信息放在第三方数据

库，此处设置为 true ， hive 将在本地创建 derby 数据库用于存放元数据。当然如果有需要也可以采用 mysql 等第三方数据库存放元数据，不过这时 hive.metastore.local 的配置值应为 false 。

2、如果你已有一套 nutch1.0 系统正在跑，而你不想单独再去部署一套 hadoop 环境，你可以直接使用 nutch1.0 自带的 hadoop 环境，但这样的部署会导致 hive 不能正常运行，提示找不到某些方法。这是由于 nutch1.0 使用了 commons-lang-2.1.jar 这个包，而 hive 需要的是 commons-lang-2.4.jar ，下载一个 2.4 版本的包替换掉 2.1 即可， nutch 和 hive 都能正常运行。

三、应用场景

本文主要讲述使用 hive 的实践，业务不是关键，简要介绍业务场景，本次的任务是对搜索日志数据进行统计分析。

集团搜索刚上线不久，日志量并不大 。这些日志分布在 5 台前端机，按小时保存，并以小时为周期定时将上一小时产生的数据同步到日志分析机，统计数据要求按小时更新。这些统计项，包括关键词搜索量 pv ，类别访问量，每秒访问量 tps 等等。

基于 hive ，我们将这些数据按天为单位建表，每天一个表，后台脚本根据时间戳将每小时同步过来的 5 台前端机的日志数据合并成一个日志文件，导入 hive 系统，每小时同步的日志数据被追加到当天数据表中，导入完成后，当天各项统计项将被重新计算并输出统计结果。

以上需求若直接基于 hadoop 开发，需要自行管理数据，针对多个统计需求开发不同的 map/reduce 运算任务，对合并、排序等多项操作进行定制，并检测任务运行状态，工作量并不小。但使用 hive ，从导入到分析、排序、去重、结果输出，这些操作都可以运用 hql 语句来解决，一条语句经过处理被解析成几个任务来运行，即使是关键词访问量增量这种需要同时访问多天数据的较为复杂的需求也能通过表关联这样的语句自动完成，节省了大量工作量。

四、Hive 实战

初次使用 hive ，应该说上手还是挺快的。 Hive 提供的类 SQL 语句与 mysql 语句极为相似，语法上有大量相同的地方，这给我们上手带来了很大的方便，但是要得心应手地写好这些语句，还需要对 hive 有较好的了解，才能结合 hive 特色写出精妙的语句。

关于 hive 语言的详细语法可参考官方 wiki 的语言手册：
http://wiki.apache.org/hadoop/Hive/LanguageManual

虽然语法风格为我们提供了便利，但初次使用遇到的问题还是不少的，下面针对业务场景谈谈我们遇到的问题，和对 hive 功能的定制。

1、 分隔符问题

首先遇到的是日志数据的分隔符问题，我们的日志数据的大致格式如下：

2010-05-24
00:00:02@$_$@QQ2010@$_$@all@$_$@NOKIA_1681C@$_$@1@$_$@10@$_$@@$_$@-1@$_$@10@$_$@application@$_$@1

从格式可见其分隔符是“ @$_$@ ”，这是为了尽可能防止日志正文出现与分隔符相同的字符而导致数据混淆。本来 hive 支持在建表的时候指定自定义分隔符的，但经过多次测试发现只支持单个字符的自定义分隔符，像“ @$_$@ ”这样的分隔符是不能被支持的，但是我们可以通过对分隔符的定制解决这个问题， hive 的内部分隔符是“ \001 ”，只要把分隔符替换成“\001 ”即可。

经过探索我们发现有两条途径解决这个问题。

a)自定义 outputformat 和 inputformat 。

Hive 的 outputformat/inputformat 与 hadoop 的 outputformat/inputformat 相当类似，inputformat 负责把输入数据进行格式化，然后提供给 hive ， outputformat 负责把 hive 输出的数据重新格式化成目标格式再输出到文件，这种对格式进行定制的方式较为底层，对其进行定制也相对简单，重写 InputFormat 中 RecordReader 类中的 next 方法即可，示例代码如下：

```
public boolean next(LongWritable key, BytesWritable value)

throws IOException {

while ( reader .next(key, text ) ) {

String strReplace = text .toString().toLowerCase().replace( "@$_$@" , "\001" );

Text txtReplace = new Text();

txtReplace.set(strReplace );

value.set(txtReplace.getBytes(), 0, txtReplace.getLength());
```

```
return true ;
}
return false ;
}
```

重写 HiveIgnoreKeyTextOutputFormat 中 RecordWriter 中的 write 方法，示例代码如下：

```
public void write (Writable w) throws IOException {
String strReplace = ((Text)w).toString().replace( "\001" , "@$_$@" );
Text txtReplace = new Text();
txtReplace.set(strReplace);
byte [] output = txtReplace.getBytes();
bytesWritable .set(output, 0, output. length );
writer .write( bytesWritable );
}
```

自定义 outputformat/inputformat 后，在建表时需要指定 outputformat/inputformat ，如下示例：

stored as INPUTFORMAT 'com.aspire.search.loganalysis.hive.SearchLogInputFormat' OUTPUTFORMAT 'com.aspire.search.loganalysis.hive.SearchLogOutputFormat'

b) 通过 SerDe(serialize/deserialize) ，在数据序列化和反序列化时格式化数据。

这种方式稍微复杂一点，对数据的控制能力也要弱一些，它使用正则表达式来匹配和处理数据，性能也会有所影响。但它的优点是可以自定义表属性信息 SERDEPROPERTIES ，在 SerDe 中通过这些属性信息可以有更多的定制行为。

2、 数据导入导出

a) 多版本日志格式的兼容

由于 hive 的应用场景主要是处理冷数据（只读不写），因此它只支持批量导入和导出数据，并不支持单条数据的写入或更新，所以如果要导入的数据存在某些不太规范的行，则需要我们定制一些扩展功能对其进行处理。

我们需要处理的日志数据存在多个版本，各个版本每个字段的数据内容存在一些差异，可能版本 A 日志数据的第二个列是搜索关键字，但版本 B 的第二列却是搜索的终端类型，如果这两个版本的日志直接导入 hive 中，很明显数据将会混乱，统计结果也不会正确。我们的任务是要使多个版本的日志数据能在 hive 数据仓库中共存，且表的 input/output 操作能够最终映射到正确的日志版本的正确字段。

这里我们不关心这部分繁琐的工作，只关心技术实现的关键点，这个功能该在哪里实现才能让 hive 认得这些不同格式的数据呢？经过多方尝试，在中间任何环节做这个版本适配都将导致复杂化，最终这个工作还是在 inputformat/outputformat 中完成最为优雅，毕竟 inputformat 是源头， outputformat 是最终归宿。具体来说，是在前面提到的 inputformat 的 next 方法中和在 outputformat 的 write 方法中完成这个适配工作。

b) Hive 操作本地数据

一开始，总是把本地数据先传到 HDFS ，再由 hive 操作 hdfs 上的数据，然后再把数据从 HDFS 上传回本地数据。后来发现大可不必如此， hive 语句都提供了“ local ”关键字，支持直接从本地导入数据到 hive ，也能从 hive 直接导出数据到本地，不过其内部计算时当然是用 HDFS 上的数据，只是自动为我们完成导入导出而已。

3、 数据处理

日志数据的统计处理在这里反倒没有什么特别之处，就是一些 SQL 语句而已，也没有什么高深的技巧，不过还是列举一些语句示例，以示 hive 处理数据的方便之处，并展示 hive 的一些用法。

a) 为 hive 添加用户定制功能，自定义功能都位于 hive_contrib.jar 包中

```
add jar /opt/hadoop/hive-0.5.0-bin/lib/hive_contrib.jar;
```

b) 统计每个关键词的搜索量，并按搜索量降序排列，然后把结果存入表 keyword_20100603 中

```
create table keyword_20100603 as select keyword,count(keyword) as count from searchlog_20100603 group by keyword order by count desc;
```

c) 统计每类用户终端的搜索量，并按搜索量降序排列，然后把结果存入表 device_20100603 中

create table device_20100603 as select device,count(device) as count from searchlog_20100603 group by device order by count desc;

d) 创建表 time_20100603 ，使用自定义的 INPUTFORMAT 和 OUTPUTFORMAT ，并指定表数据的真实存放位置在 '/LogAnalysis/results/time_20100603' （ HDFS 路径），而不是放在 hive 自己的数据目录中

create external table if not exists time_20100603(time string, count int) stored as INPUTFORMAT 'com.aspire.search.loganalysis.hive.XmlResultInputFormat' OUTPUTFORMAT 'com.aspire.search.loganalysis.hive.XmlResultOutputFormat' LOCATION '/LogAnalysis/results/time_20100603';

e) 统计每秒访问量 TPS ，按访问量降序排列，并把结果输出到表 time_20100603 中，这个表我们在上面刚刚定义过，其真实位置在 '/LogAnalysis/results/time_20100603' ，并且由于 XmlResultOutputFormat 的格式化，文件内容是 XML 格式。

insert overwrite table time_20100603 select time,count(time) as count from searchlog_20100603 group by time order by count desc;

f) 计算每个搜索请求响应时间的最大值，最小值和平均值

insert overwrite table response_20100603 select max(responsetime) as max,min(responsetime) as min,avg(responsetime) as avg from searchlog_20100603;

g)创建一个表用于存放今天与昨天的关键词搜索量和增量及其增量比率，表数据位于 '/LogAnalysis/results/keyword_20100604_20100603' ，内容将是 XML 格式。

create external table if not exists keyword_20100604_20100603(keyword string, count int, increment int, incrementrate double) stored as INPUTFORMAT 'com.aspire.search.loganalysis.hive.XmlResultInputFormat' OUTPUTFORMAT 'com.aspire.search.loganalysis.hive.XmlResultOutputFormat' LOCATION '/LogAnalysis/results/keyword_20100604_20100603';

h)设置表的属性，以便 XmlResultInputFormat 和 XmlResultOutputFormat 能根据 output.resulttype 的不同内容输出不同格式的 XML 文件。

alter table keyword_20100604_20100603 set tblproperties ('output.resulttype'='keyword');

i) 关联今天关键词统计结果表（ keyword_20100604 ）与昨天关键词统计结果表（ keyword_20100603 ），统计今天与昨天同时出现的关键词的搜索次数，今天相对昨天

的增量和增量比率，并按增量比率降序排列，结果输出到刚刚定义的 keyword_20100604_20100603 表中，其数据文件内容将为 XML 格式。

insert overwrite table keyword_20100604_20100603 select cur.keyword, cur.count, cur.count-yes.count as increment, (cur.count-yes.count)/yes.count as incrementrate from keyword_20100604 cur join keyword_20100603 yes on (cur.keyword = yes.keyword) order by incrementrate desc;

4、用户自定义函数 UDF

部分统计结果需要以 CSV 的格式输出，对于这类文件体全是有效内容的文件，不需要像 XML 一样包含 version ， encoding 等信息的文件头，最适合用 UDF(user define function)了。

UDF 函数可直接应用于 select 语句，对查询结构做格式化处理之后，再输出内容。自定义 UDF 需要继承 org.apache.hadoop.hive.ql.exec.UDF ，并实现 evaluate 函数，Evaluate 函数支持重载，还支持可变参数。我们实现了一个支持可变字符串参数的 UDF ，支持把 select 得出的任意个数的不同类型数据转换为字符串后，按 CSV 格式输出，由于代码较简单，这里给出源码示例：

```
public String evaluate(String... strs) {

StringBuilder sb = new StringBuilder();

for ( int i = 0; i < strs. length ; i++) {

sb.append(ConvertCSVField(strs[i])).append( ',' );

}

sb.deleteCharAt(sb.length()-1);

return sb.toString();

}
```

需要注意的是，要使用 UDF 功能，除了实现自定义 UDF 外，还需要加入包含 UDF 的包，示例：

add jar /opt/hadoop/hive-0.5.0-bin/lib/hive_contrib.jar;

然后创建临时方法，示例：

CREATE TEMPORARY FUNCTION Result2CSv AS ‘com.aspire.search.loganalysis.hive.Result2CSv';

使用完毕还要 drop 方法，示例：

DROP TEMPORARY FUNCTION Result2CSv;

5、输出 XML 格式的统计结果

前面看到部分日志统计结果输出到一个表中，借助 XmlResultInputFormat 和 XmlResultOutputFormat 格式化成 XML 文件，考虑到创建这个表只是为了得到 XML 格式的输出数据，我们只需实现 XmlResultOutputFormat 即可，如果还要支持 select 查询，则我们还需要实现 XmlResultInputFormat ，这里我们只介绍 XmlResultOutputFormat 。

前面介绍过，定制 XmlResultOutputFormat 我们只需重写 write 即可，这个方法将会把 hive 的以 ’\001’ 分隔的多字段数据格式化为我们需要的 XML 格式，被简化的示例代码如下：

```
public void write(Writable w) throws IOException {

String[] strFields = ((Text) w).toString().split( "\001" );

StringBuffer sbXml = new StringBuffer();

if ( strResultType .equals( "keyword" )) {

sbXml.append( "<record><keyword>" ).append(strFields[0]).append(

"</keyword><count>" ).append(strFields[1]).append( "</count><increment>" ).append(strFields[2]).append(

"</increment><rate>" ).append(strFields[3]).append(

"</rate></result>" );

}

Text txtXml = new Text();
```

```
byte [] strBytes = sbXml.toString().getBytes( "utf-8" );

txtXml.set(strBytes, 0, strBytes. length );

byte [] output = txtXml.getBytes();

bytesWritable .set(output, 0, output. length );

writer .write( bytesWritable );

}
```

其中的 strResultType .equals("keyword") 指定关键词统计结果，这个属性来自以下语句对结果类型的指定，通过这个属性我们还可以用同一个 outputformat 输出多种类型的结果。

alter table keyword_20100604_20100603 set tblproperties ('output.resulttype'='keyword');

仔细看看 write 函数的实现便可发现，其实这里只输出了 XML 文件的正文，而 XML 的文件头和结束标签在哪里输出呢？所幸我们采用的是基于 outputformat 的实现，我们可以在构造函数输出 version ， encoding 等文件头信息，在 close() 方法中输出结束标签。

这也是我们为什么不使用 UDF 来输出结果的原因，自定义 UDF 函数不能输出文件头和文件尾，对于 XML 格式的数据无法输出完整格式，只能输出 CSV 这类所有行都是有效数据的文件。

五、总结

Hive 是一个可扩展性极强的数据仓库工具，借助于 hadoop 分布式存储计算平台和 hive 对 SQL 语句的理解能力，我们所要做的大部分工作就是输入和输出数据的适配，恰恰这两部分 IO 格式是千变万化的，我们只需要定制我们自己的输入输出适配器， hive 将为我们透明化存储和处理这些数据，大大简化我们的工作。本文的重心也正在于此，这部分工作相信每一个做数据分析的朋友都会面对的，希望对您有益。

本文介绍了一次相当简单的基于 hive 的日志统计实战，对 hive 的运用还处于一个相对较浅的层面，目前尚能满足需求。对于一些较复杂的数据分析任务，以上所介绍的经验很可能是不够用的，甚至是 hive 做不到的， hive 还有很多进阶功能，限于篇幅本文未能涉及，待日后结合具体任务再详细阐述。

十一、 hadoop 中用 java 访问 hive

```
export HADOOP_HOME=/opt/hadoop/hadoop-1.0.3
export JAVA_HOME=/opt/hadoop/jdk1.7.0_06
export HIVE_HOME=/opt/hadoop/hive-0.8.1
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin:$HIVE_HOME/bin
export CLASSPATH=
expor
tCLASSPATH=$CLASSPATH:$HIVE_HOME/lib/mysql-connector-java-5.1.18-bin.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/hive-exec-0.8.1.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/hive-jdbc-0.8.1.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/hive-metastore-0.8.1.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/hive-service-0.8.1.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/libfb303.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/log4j-1.2.16.jar
export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/slf4j-log4j12-1.6.1.jar

export CLASSPATH=$CLASSPATH:$HIVE_HOME/lib/slf4j-api-1.6.1.jar

export
CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-configuration-1.6.jar

export CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-logging-1.1.1.jar
export CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-io-2.1.jar

export CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.0.3.jar



import java.sql.SQLException;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.sql.DriverManager;


public class HiveJdbcClient {
```

```
    private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";



    /**
   * @param args
   * @throws SQLException
    */
    public static void main(String[] args) throws SQLException {
        try {
        Class.forName(driverName);
      } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        System.exit(1);
      }
      Connection con =
DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");
      Statement stmt = con.createStatement();
      String tableName = "ghh";
      //stmt.executeQuery("drop table " + tableName);
      //ResultSet res = stmt.executeQuery("create table " + tableName + " (key int, value
string)");
      ResultSet res;
      // show tables
      String sql = "show tables '" + tableName + "'";
      System.out.println("Running: " + sql);
      res = stmt.executeQuery(sql);
      if (res.next()) {
        System.out.println(res.getString(1));
      }
      // describe table
      sql = "describe " + tableName;
      System.out.println("Running: " + sql);
      res = stmt.executeQuery(sql);
      while (res.next()) {
```

```
      System.out.println(res.getString(1) + "\t" + res.getString(2));
    }


    // load data into table
    // NOTE: filepath has to be local to the hive server
    // NOTE: /tmp/a.txt is a ctrl-A separated file with two fields per line
    //String filepath = "/tmp/a.txt";
    //sql = "load data local inpath '" + filepath + "' into table " + tableName;
    //System.out.println("Running: " + sql);
    //res = stmt.executeQuery(sql);


    // select * query
    sql = "select * from " + tableName;
    System.out.println("Running: " + sql);
    res = stmt.executeQuery(sql);
    while (res.next()) {
      System.out.println(String.valueOf(res.getInt(1)) + "\t" + res.getString(2));
    }


    // regular hive query
    sql = "select count(1) from " + tableName;
    System.out.println("Running: " + sql);
    res = stmt.executeQuery(sql);
    while (res.next()) {
      System.out.println(res.getString(1));
    }
  }
}
```

十二、 Hive— 基于 Hadoop 的 PB 级数据仓库

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao,Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu

and Raghotham Murthy

Facebook 数据基础设施团队

摘要——用于商业智能所收集和分析的数据集正在迅速增长，使得传统的数据仓库解决方案变得极其昂贵。Hadoop[1]作为一个流行的开源 Map/Reduce 实现被如 Yahoo，Facebook 等公司所使用，用以在普通硬件上存储和处理极其巨大的数据集。

然而，Map/Reduce 是非常底层的编程模式，需要开发人员编写难以维护和重用的程序。在本文中，我们将介绍 Hive，一个开源的基于 Hadoop 的数据仓库解决方案。Hive 支持类 SQL 的声明式查询语言 —HiveQL，它被编译成的 MapReduce 工作并由 Hadoop 执行工作。此外，HiveQL 允许用户使用自定义 Map/Reduce 脚本查询。该语言包含一个类型系统，此系统包含原始类型，如数组和地图的集合，
相同的嵌套组成。底层的 IO 库可以被扩展用于使用自定义格式查询。Hive 还包括系统目录 - Metastore - 包含模式(schema)和有用的数据探测相关统计数据，查询优化和查询编译。在 Facebook，Hive 数据仓库包含数万张表(table)和超过 700TB 的数据，Hive 每月被超过 200 用户的报告和即时查询所使用。

I.导言

在 Facebook，大型数据集上的伸缩性分析一直是一部分团队的工作的核心功能— 无论工程的和非工程的。除了商业情报分析师使用的即席分析和商业智能应用，Facebook 的产品也是基于分析的。这些产品的范围从简单的报告像 Facebook 的广告网络的 Insight，到更先进的，如 Facebook 的词典的产品[2]。因此，一个灵活的基础设施的需求，迎合这些不同的应用程序和用户，同时能以一种有效的方式统计衡量 Facebook 日益增长的数据，是至关重要的。Hive 和 Hadoop 正式我们在 Facebook 中用来满足以上要求所需要的使用的技术。

2008 年之前，Facebook 整个数据处理基础设施都是围绕商用 RDBMS 所构建的，我们产生的数据增长速度非常快 – 举例说明，我们 2007 年拥有 15TB 数据量，如今我们拥有 700TB 数据。那时的数据处理基础设施现在看来是如此的不合时宜，以至于某些日常的数据处理工作需要花费超过一天的时间来处理，并且这种情况正在变得越来越糟糕。我们对于基础设施有一个迫切需要，可以扩展我们数据的伸缩性。因此，我们开始探索的 Hadoop 作为一种解决我们的伸缩性需求的技术。事实上，Hadoop 已经是一个开源项目，它使用普通硬件处理 PB 级规模的数据并提供的可伸缩性，这对我们来说是非常引人注目。同样的工作超过一天才能完成，现在使用 hadoop 只需要几个小时即可。

然而，对最终用户来说，使用 Hadoop 并不容易，特别是对那些不熟悉与 Map/Reduce 的用户。最终用户不得不为简单的任务，如计算行数或平均数，编写 Map/Reduce 程序。Hadoop 缺乏想 SQL 这样的流行查询语言的表达式，因此最终用户必须为了很简单的分析编写几个小时的程序。为了真正的加强公司的数据分析能力和效率，我们非常清楚必须提升 Hadoop 的能力。为了让数据更贴近用户，我们在 2007 年 1 月开始构建 Hive。我们的愿景是将类似于表(Table),列(Column)，分区(partition)等概念引入并且使得其成为 SQL 子集，从而来解耦合 Hadoop，同时保留 Hadoop 的扩展性和灵活性。Hive 在 2008 年 8 月开源，从那时起，为了数据处理的需要，很多 Hadoop 用户开始使用和探索 Hive。

从一开始，Hive 在 Facebook 就非常的流行。如今，为了各式各样的应用，Hive 定期在 Hadoop/Hive 集群上运行数千个工作，这些应用不仅有简单的统计摘要，也有复杂的商业智能，机器学习及支持 Facebook 的产品特性。

在随后的部分中，我们将对 Hive 的架构和能力提供更多的详细内容。第二部分描述了数据模型，类型系统和 HiveQL。第三部分，详细描述数据是如何存储在底层的分布式文件系统中，即— HDFS(Hadoop file system)。第四部分描述了系统架构及不同的 Hive 组件。第五部分我们着重描述 Hive 在 Facebook 的使用统计数据并在第六部分我们提供相关的工作。第七部分是我们对未来工作的展望。

II. 数据模型，类型系统和查询语言

Hive 将数据描述成易于理解的概念，如表，列，行和分区。它支持所有的主要原始数据类型— Integers,Floats,Doubles 和 Strings —同样地，也支持复杂的数据类型，如 Maps,List,和 Structs。后者可以随意地内嵌，从而构造更为复杂的类型。另外，Hive 允许我们使用自

己的类型和功能进行扩展。Hive 的查询语言与 SQL 非常类似，因此任何一个熟悉 SQL 的用户都可以轻松地理解。但是在数据模型，类型系统和 HiveQL 上，它与传统的数据库有细微差别并且在 Facebook 我们已经有所体会。在此部分，我们将会着重强调这些方面和一些其他的细节。

A. 数据模型和类型系统

Integers — bigint(8 字节)，int(4 字节)，smallint(2 字节)，tinyint(1 字节)。所有的 Integer 类型都是有符号的。

Floating point numbers — float(单精度)，double(双精度)。

String 。

Hive 同样支持本地复杂类型：

Associativearrays—map<key-type, value-type>

Lists—list<element-type>

Structs—struct<file-name:field-type, ... >

这些复杂类型是模板并可以用于生成任意复杂类型。比如，list<map<string, struct<p1:int, p2:int>>表示映射 String 到 Struct 的相关数组列表，该数组按包含两个 Interger 字段，一个名为 p1,另一个名为 p2。这些能够在创建表声明中放在一起，从而创建期望的模式。比如说，以下声明创建了一个拥有复杂模式的表 t1。

CREATETABLE t1(st string, fl float, li list<map<string,

struct<p1:int,p2:int>>);

查询表达式能够通过是同'.'运算符来访问 Struct 中的字段。相关数组和列表的中的值可以使用'[]'运算符来访问。在前列中，t1.li[0]给出了列表的第一个与元素，t1.li[0]['key']给出了列表中与'key'相关的 struct。最后，Struct 中的 p2 字段可以通过 t1.li[0]['key'].p2 访问。这样，Hive 就能够支持任意复杂的结构。

上述中的表通过期待的方式创建，并被 Hive 中现成的默认序列化器(Serializer)和反序列化器(Deserializer)序列化和反序列化。然而，有些时候数据是由其他程序或者一流数据所提供。Hive 提供一种灵活性，包括，使表中包含并未转换过的诗句，这样能够为大型数据集处理节省大量的时间。像稍后描述的一样，我们可以通过向 Hive 提供实现 SerDe 接口的 jar 来达成目的。在这样的情况下，类型信息也可以通过提供 ObjectInspector 接口来实现，同时，我们可以在 SerDe 接口实现中使用 getObjectInspector 方法来提供接口。关于这个接口的详细资料可以参见 Hivewiki[3]，但是，此处的最基本方式是通过提供包含 SerDe 和 ObjectInspector 接口实现的 jar 来将任意的数据格式和类型进行编码。因而，一旦在 Tables 和 jar 之间形成了恰当的关联，那么查询层面将会把他们与本地的类型和格式视为同一水平。比如说，下面的声明将包含 SerDe 和 ObjectInspector 接口的 jar 添加到分布式缓存中，从而让 Hadoop 通过过在 custom serde 上创建表来处理。

addjar/jars/myformat.jar；

CREATE TABLE t2 ROW FORMAT SERDE 'com.myformat.MySerDe';

注意，如果可以，这些表模式可以由组成复杂和原始的类型的方式提供。

B．查询语言

Hive 查询语言(HiveQL)有 SQL 和一些在我们的环境中对我们有用的扩展组成。HiveQL 所具有的 SQL 特性如子查询，varioustypes of joins – inner，left outer,right outer 和外连接，笛卡尔积，group by 和 aggregation , union all, createtable as select 和很多对原始和复杂类型有用的功能是得它很像 SQL。事实上，像很多概念所提到的那样，他确实很像 SQL。这使得一个熟悉 SQL 的人来说，都能够快速的启动一个 Hive 命令行并且开始查询系统。浏览元数据如 Tables 和描述同样是现成的并且 explain plan capabilities to inspect query plans(虽然 the plan 与传统的 RDBMS 中的 explainplain 非常不同)同样现成可用。但是有一些限制，如只有 equalitypredicates 在 join predicate 中是支持的并且这中 joins 必须使用 ANSI 语法，如

SELECT t1.a1 as c1, t2.b1 as c2

FROM t1, t2

WHERE t1.a2 = t2.b2

另外的限制是 inserts 如何被完成。Hive 目前并不支持插入现存的 table 或者数据分区中并且所有的 inserts 都会覆盖现有数据。因此，我们这样清楚地描述语法：

INSERT OVERWRITE TABLE t1

SELECT * FROM t2;

在实际情况下，这些限制实际上都不是问题。我们几乎没见过表意不清楚的 equi-join 查询并且因为大部分数据都被我们实时地每天地加载进数据仓库，我们仅仅是将这些数据加载进当天或当时的新的表分区。然而，我们意识到多次地平凡地加载，这些分区将会变得非常的大并且可能会需要我们实现 INSERT INTO 语义。Hive 对 INSERT INTO , UPDATE 和 DELETE 的缺乏在另外一方面使得我们能够使用非常简单的机制来处理并发读写而不需要实现复杂的锁协议。

除了这些限制之外，HiveQL 扩展能够支持特定的分析如用户实现的 Map-Reduce 程序和用户自选的编程语言实现。这使得高级用户按照 Map-Reduce 程序来表达复杂逻辑，这些 MapReduce 程序都是无缝地挂如 HiveQL 查询的。有些时候，某些库是由 Python, php 或者其他用户所需要用来做数据转换的语言所编写的，这成了唯一合理的方法。以在一张表中字母计数为例，通过以下方法是用 MapReduce：

FROM(

MAP doctext USING ‘python wc_mapper.py’ AS (word, cnt)

FROM docs

CLUSTER BY word

)a

REDUCE word, cnt USING ‘python wc_reduce.py’;

正如上例中所展示的一样，MAP 子句指明了如何输入列(此例中使用的是 doctext)如何被用户的程序(此例中使用的是’python wc_mapper.py‘)转换到输出列(word 和 cnt)。CLUSTERBY 子句表明了输出列进行了散列处理，将这些数据分布到 Reducers 中，最后

REDUCE 子句指定了在输出列上调用用户程序(此例中使用的是 python wc_reduce.py)。有些时候，Mapper 和 Reducer 之剑的分布标准需要向 Reducer 提供数据，因此它能在和之前被用于分布的列集不同的列集上进行分类，如一个会话中的所有动作都需要被时间所排序。Hive 提供 DISTRIBUTE BY 和 SORT BY 子句来完成这些功能，举例如下：

```
FROM (
  FROM session_table
  SELECT sessionid, tstamp,data
  DISTRIBUTE BY sessionidSORT BY tstamp
)a
REDUCE sessionid, tstamp, data USING 'session_reducer.sh'
```

注意，在上例中没有指明输入列的 Map 子句，同样地，当 Reduce 部分不需要转换数据是同样也是可以不需要 REDUCE 子句的， FROM 子句出现在 SELECT 子句之前也是不符合 SQL 语法的。Hive 是允许用户将 SELECT/MAP/REDUCE 子句通过子查询的方式进行位置互换的。当处理多插入时这是非常有用和直观的。HiveQL 支持将不同的转换结果插入不同的表，分区，HDFS 或者本地目录作为查询的一部分。这种能力使得在 Reducing number ofscans done on the input data，举例如下：

```
FROM t1
 INSERT OVERWRITE TABLE t2
 SELECT t3.c2, count(1)
 FROM t3
 WHERE t3.c1 <= 20
 GROUP BY t3.c2

  INSERT OVERWRITEDIRECTORY '/output_dir'
```

```
SELECT t3.c2, avg(t3.c1)

FROM t3

WHERE t3.c1 > 20 ANDt3.c1 <= 30

GROUP BY t3.c2


INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'

SELECT t3.c2, sum(t3.c1)

FROM t3

WHERE t3.c1 > 30

GROUP BY t3.c2;
```

在这个例子中，t1 的不同部分被聚合，并且用于生成 t2,一个 HDFS 目(/output_dir)和一个本地目录(/home/dir)。

III．数据存储， SerDE 和 文件格式

A． 数据存储

由于表是 Hive 这中的逻辑数据单元，表的元数据将表中的数据与 HDFS 目录相关联。这些主要数据单元和他们在 HDFS 名字空间中的映射如下：

，Tables — 存储在 HDFS 目录中的表。

.Partitions — 存储在子目录下的表分区，并且该子目录存在与表的目录中。

Buckets — 存储在分区或表目录下的 bucket,他依赖与该表是否被分区。

举例来说，在 HDFS 中，表 test_table 被 map 到<warehouse_root_directory>/test_table 中。warehouse_root_directory 被 hive-site.xml 中的 hive.metastore.warehouse.dir 参数所指定。默认参数值设置在/user/hive/warehouse 中。

一张表可能是分区的或者非分区的。我们可以像下面的 CREATE TABLE 声明一样在声明中加入 PARTITIONED BY 子句来创建一张分区的表。

CREATE TABLE test_part(c1 string， c2 int)

PARTITIONEDBY(ds string, hi int);

如上例所示，表分区将被存储到/user/hive/warehouse/test_part 目录下的 HDFS 中，分个为不同的 DS 和 HR 值所存在的分区都由用户指定。注意，列分区并不是表数据的一部分并且分区列的值在分区目录下被编码(他们也存储于表的元数据中)。一个新的分区能够由 INSERT 或者 ALTER 声明通过像表中添加分区的方式创建。如以下两种声明方式：

INSERT OVERWRITE TABLE

Test_part PARTITION(DS='2009-01-01', HR=12)

SELECT * FROM t;

ALERT TABLE test_part

ADD PARTITION(ds='2009-02-02', hr=11);

以上都能向表 test_part 中添加新的分区。INSERT 声明同样能够在表 t 中为数据分配分区，就像创建空的分区一样来修改表。两种声明都已在 HDFS 目录项创建相应目录—/user/hive/warehouse/test_part/ds=2009-01-01/hr=12 和 /user/hive/warehouse/test_part/ds=2009-02-02/hr-11 结束。这种方法在创建带有如一下字符的分区值时会比较复杂，如'/ '或者'：',因为这些字符会在 HDFS 指定目录结构时使用，但是适当的规避这些字符确实可以更好的提高 HDFS 目录名称的兼容性。

Hive 编译器能够减少为了评估查询所需要浏览的目录。像一下的例子一样：

SELECT * FROM test_part WHERE ds='2009-01-01';

这将只会浏览在/user/hive/warehouse/test_part/ds=2009-01-01 目录下的数据，同时一下查询

SELECT * FROM test_part

WHERE ds='2009-02-02'AND hr=11;

将只会浏览在/user/hive/warehouse/test_part/ds=2009-01-01/hr=12 目录下的数据。减少数据量对查询时间的影响是具有非常重大意义的。在某种意义上说，这种分区模式就是被数据库厂商所提及的分区列表，但是这儿又有一些区别，分区的建是被存储在元数据而不是数据中的。

Hive 所使用的最终存储单元概念是所说的 Buckets。一个 Bucket 是一张表或者一个分区的叶子级别的目录。当表被创建时，用户可以指定所需要的 Bucket 的数目和哪一列来 bucket[存储]数据。在目前的实现中，这些信息被用于减少用户在简单数据上执行查询的数据量，比如说一张被 bucket 到 32 个 buckets 的表能够通过查看第一个 bucket 的数据从而快速的生成 1/32 的样品。类似于下面的声明：

SELECT * FOMR t TABLESAMPE(2 OUT OF 32);

它会浏览在第二个 bucket 中的数据。注意，需要确保的是这个 bucket 已经被正确的创建和命名，同时 HiveQL 的 DDL 声明目前是不会试图任何为了让数据与表属性相兼容的方式 bucket 这些数据的。因此，bucketing 的数据需要小心谨慎的使用。

虽然这些数据与表都被存储在 HDFS 中的<warehouse_root_directory>/test_table 下，Hive 也允许用户查询位于 HDFS 其他位置的数据。我们可以通过如下例所示的 EXTERNAL TABLE 来达到目的：

CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)

LOCATION '/user/mytables/mydata';

通过上例的声明，用户能够指定 test_extern 作为一张外部表，该表每行由两列—c1 和 c2 组成。另外，这些数据文件是存储在 HDFS 的/user/mytables/mydata 中的。注意，和传统的已经定义的 SerDe 不同，这些数据被认为是 Hive 的外部格式。一张外部表与普通表的不同只在于对外部表的 drop 命令只会删除它的元数据，而不会删除任何数据，而对于普通表的 drop 命令会也会删除与之上官的数据。

B.序列化/反序列化

像之前所提到，Hive 能够解释用户提供的 java SerDe 接口的实现并将只与表和分区想关联。这样，我们能够轻松的解释和查询传统的数据格式。这种 Hive 中默认的 SerDe 实现被称为 LazySerDe—它 lazily 地将列序列化到内部对象中，从而使得只有在行和列在查询表达式中需要时的会行序列化开销表的 incurred。在 LazzySerDe 存储数据的文件中，所有行由换行符分界(ascii 码：13)并且每一行的列被 ctrl-A(ascii 码 1)分界。SerDe 能够用于读取使用任何分隔符分隔的列的数据。

举例说明，以下声明：

```
CREATE TABLE test_delimited(c1 string, c2 int)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\002'
    LINES TERMINATED BY '\012';
```

以上声明指定了表 test_delimited 的数据使用 ctrl-B(ascii 码：2)作为列的分隔符并且使用 ctrl-L(ascii 码 12)作为行分隔符。另外，分隔符可以用于分隔 Map 的序列化键和值并且不同的分隔符能够用于分隔 List(集合)。如以下的示例：

```
CREATE TABLEtest_delimited2(c1 string, c2 list<map<string, int>>)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\002'
  COLLECTION ITEMS TERMINATED BY '\003'
  MAP KEYS TERMINATED BY '\004';
```

除 LazySerDe 之外，hive_contrib.jar 中还有一些有趣的 SerDe 他们通过分配提供。其中一个非常有用的是 RegexSerDe，它允许用户指定正则表达式从行中解析出各列。举例如下，分析 apache 日志：

```
CREATE TABLE apachelog(
```

```
    host string,
    identity string,
    user string.
    request string,
    status string,
    referer string,
    agent string)
ROW FORMAT SERDE
    'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES(
    'input.regex'='([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\")(-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^
\"]*|\"[^\"]*\"))?',
'output.format.string'='%1$s %2$s %3$s %4$s%5$s %6$s
%7$s %8$s %9$s');
```

Input.regex 属性是用用在每条记录上的正则表达式，output.format.string 指明了如何从与正则表达式匹配的组中构造列字段。这个例子展示了通过 WITH SERDEPROPERTIES 如何使任意键值对传递给 serde，这种能力在将任意参数传递给传统 SerDe 是非常有用的。

C．文件格式

Hadoop 文件可以用不同的格式存储。Hadoop 中的文件格式指定了数据是如何被存储在文件中的。例如，文本文件是由 TextInputFormat 格式存储的，二进制文件只由 SequenceFileInputFormat 格式存储的。用户可以自己实现文件格式。Hive 在存储数据的文件的类型的输入格式上强加限制。当创建表的时候可以指定格式。除了之前所提过的两种格

式以外，Hive 也提供一种面向列存储方式的格式—RCFileInputFormat。这样，用户可以提供重大的性能提升，特别是在某些不需要访问表的所有列的查询。用户可以添加自己的文件格式并且像以下声明那样将他们与表相关联：

```
CREATE TABLE dest1(key INT, value STRING)
    STORED AS
        INPUTFORMAT
            'org.apache.hadoop.mapred.SequenceFileInputFormat'
        OUTPUTFORMAT
            'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

STORED AS 指定决定位于表或者分区目录下的文件的输入，输出格式的类。这些类只需要实现 FileInputFormat 和 FileOutputFormat 接口。这些类生成对应的 jar 文件，并通过与添加传统 SerDe 类似的方式提供给 Hadoop。

IV. 系统架构和组件

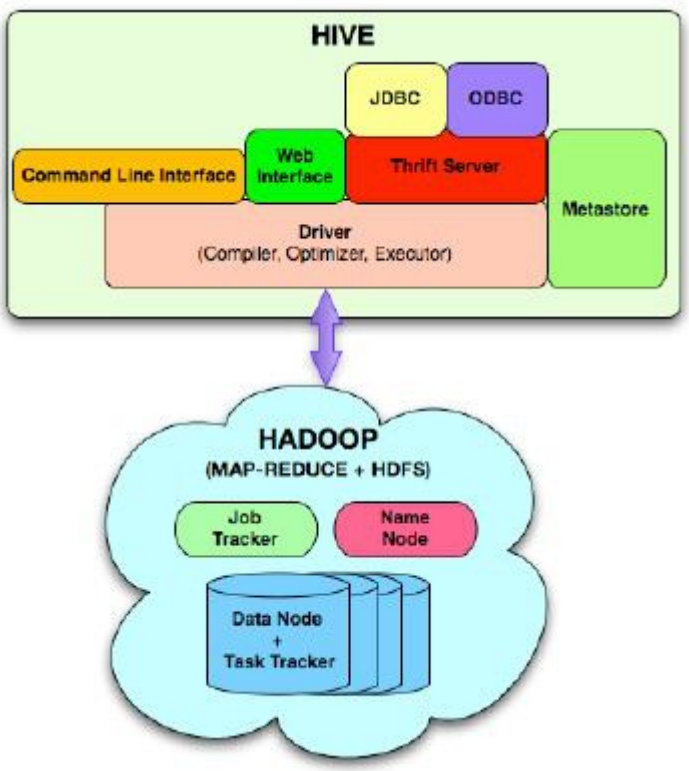


图 1.系统架构

以下组件是构成 Hive 的主要部分：

.Metastore — 该组件存储了与表，列，分区等等相关的系统目录和元数据。

.Driver — 该组件通知 HiveQL 声明的生命周期，因为它贯穿 Hive。该组件也维护着一个会话句柄和所有的会话统计。

Query Compiler — 该组件将 HiveQL 编译进定向的非周期性的 Map/Reduce 任务曲线图。

Excution Engine — 该组件在适当的依赖顺序下执行编译器所生成的任务。该组件与底层的 Hadoop 实力交互。

HiveServer — 该组件提供了一个 thrift 接口和一个 JDBC/ODBC 服务并且提供 Hive 与其他应用交互的方式。

Clients 组件就像命令行,(CLI) web UI 和 JDBC/ODBC 驱动一样。

Extensibility 接口包括 之前描述过的 SerDe 和 ObjectInspector 接口，也包括 UDF(用户定义功能)和 UDAF(用户定义集成功能)接口，它们允许用户定义自己的自定义功能。

HiveQL 声明通过命令行(CLI)，web UI 或者使用 thrift，ODBC 或者 JDBC 接口的外部提交。首先，Driver 将查询声明传递给编译器进行标准解析，类型检查和语义分析，使用存储在 Metastore 中的元数据。然后，编译器生成一个逻辑计划，该计划被 Optimizer 使用简单规则进行优化。最后，一个优化后的 DAG 形式的 Map/Reduce 任务和 HDFS 任务被生成。之后 execution engine 使用 hadoop 并按照依赖顺序执行这些任务。

在这部分我们会提供关于 Metastore，QueryCompiler 和 Execution Engine 更详细的介绍。

A. Metastore

Metastore 是 Hive 的系统目录。他存储了所有表，分区，模式，列和列的类型，表的位置等等的相关信息。这些信息可以使用 thrift 接口来查询和修改，因此，该信息可以被不同的编程语言实现的客户端所调用。编译器需要快速的相关信息服务，我们可以将这些数据存储在传统的 RDBMS 中。因此，Metastore 成为了一个运行在 RDBMS 上的应用，该应用使用一个称为 DataNucleus 的开源 ORM 层，为了将对象表示转换成关系模式，反之亦然。我们反对将这些信息存储在 HDFS 中，因为我们需要 Metastore 具有非常低的延迟。

DataNucleus 层允许我们挂入多种不用的 RDBMS 技术。我们在 Fackbook 的开发中，使用 mysql 来存储这些数据。

Metastore 对 Hive 来说是至关重要的。失去了系统目录，我们将无法使用 hadoop 文件中的结构。因此，定期备份存储在 Metastore 中的数据是如此的重要。理想状态下，应该部署冗余的服务器来满足我们的产品环境的需要。另外，确保对用户查询数目的伸缩性也是非常重要的。Hive 确保了所有的 Metastore 调用都不会由 Mapper 或者 Reducer 来完成以满足上一要求。所有 Mapper 或者 Reducer 所需要的元数据都是由编译器生成的 XML 计划文件来传递的，并且其中包含了所有运行时所需要的信息。

OMetastore 中的 ORM 逻辑能够被部署在客户端的库中，因此它是运行在客户端的，并且，issues 直接调用一个 RDBMS。当所有的客户端都以命令行或者 WEB UI 的方式与 Hive 交互，那么这种部署是非常理想的并易于开展。然而，非 java 程序语言，如 python，php 等，所编写的程序都需要以最快的速度对 Hive 的元数据进行操作和查询，因而，我们需要部署一个独立的 Metastore 服务器。

B．查询编译器

查询编译器使用存储在 Metastore 中的元数据来生成可执行计划。就像传统数据库的编译器一样，Hive 编译器以以下步骤处理 HiveQL 声明：

解析(Parse) — Hive 使用 Antlr 为查询生成抽象语法树(AST)。

类型检查和语义分析(Typechecking and Semantic Analysis) — 在这部分，编译器抓取 Metastore 中所有输入，输入表的信息并且使用这些数据构建逻辑计划。在此阶段，它会检查表达式的类型兼容性并且标记所有编译时错误。AST 到 DAG 的转换过程的操作是通过一个叫做查询块(Query Block)树的中间媒介表示完成的。编译器通过查询块树将内嵌查询转换为父子关系。同时，查询块树表示也帮助将 AST 树相关部分组织成一种形式，该形式更易于被转换成 DAG 的操作而不是 vanilla AST。

优化(Optimization)— 优化逻辑由一个转换链组成，如一个转换操作结果的 DAG 被作为输入传递给下一个转换操做。任何一个希望改变这样的编译器或者添加新的优化逻辑的用户都可以通过以下方式来完成,在实现该转换过程时，实现 Transform 接口扩展并且将其添加到 Optimizer 的转换链当中。

标准转换逻辑包含 walk on 这个 DAG 操作，以致确保处理动作被 DAG 操作着手，在相关条件或规则满足时。涉及转换相关的五种主要接口有 Node, GraphWalker, Dispatcher,

Rule 和 Processor。DAG 操作中的节点实现了 Node 接口。这样，我们可以使用以上提及的其他接口来管理 DAG 操作。标准的转换调用 walking 当前 DAG 并且访问每个 Node 实现，检查是否满足 Rule 实现，然后为已满足的 Rule 调用相应的 Processor。Dispatcher 维护这所有 Rule 到 Processor 的映射并且进行 Rule 的匹配。当一个 Node 实现被访问时，它被传递给 GraphWalker，以确保合适的 Processor 能够被分发。以下的序列图 Fig.2 展示了如何构造一个保准的转换。

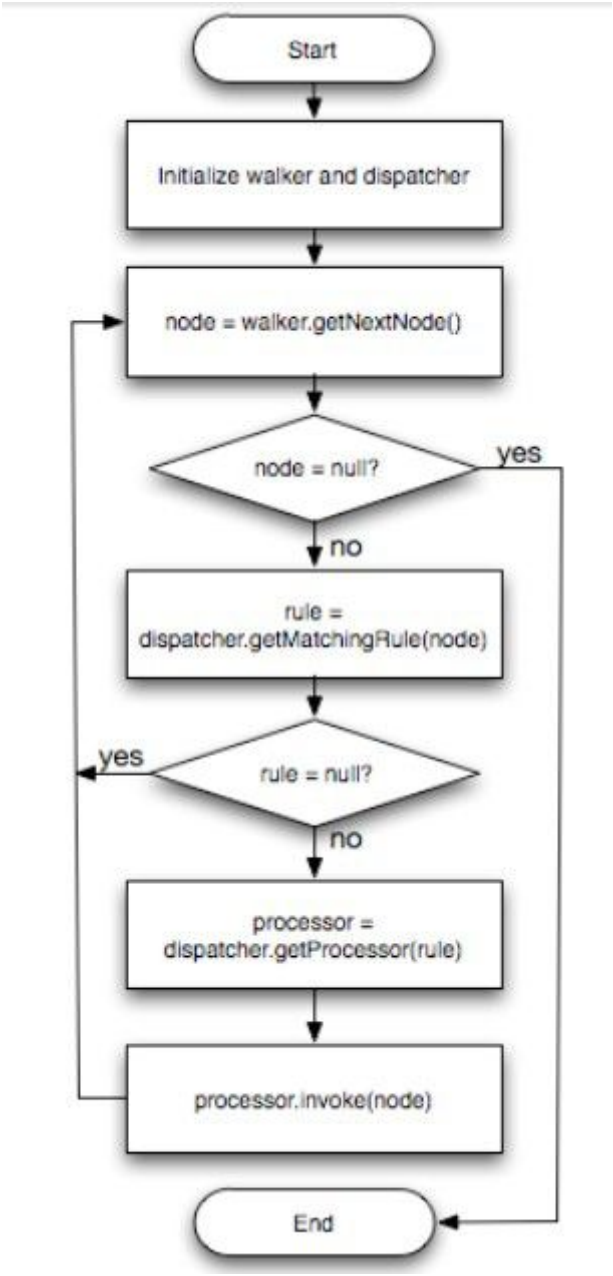


图 2.优化时的标准转换流程图

下面的转换步骤会作为优化的一部分在 Hive 中立即执行：

i. 列删减(Column pruning) — 该优化步骤确保只有查询真正需要的列才会被从行中投射出来。

ii. 谓词下挂(Predicate pushdown) — 如果允许，谓词将会被下挂扫描，因此处理时可以更早地过滤行。

iii. 分区删减(Partition pruning) — 位于被分区列上的谓词被用于删除那些不满足谓词的分区文件。

iv. 映射连接(Map side joins) — 某些连接中的表非常小，这些表在所有的 Mappper 中重复出现并且与其他表相连接。这种情况被以下的查询声明格式处罚：

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1

 FROM t1 JOIN t2 ON(t1.c2= t2.c2);
```

Mapper 持有重复表中内容的总内存数量被一些列的参数所控制。如 hive.mapjoin.size.key 和 hive.mapjoin.cache.numrows，它们控制着任何时候都存储在内存中的表的列数目并且也向系统提供连接键的大小。

v. 连接重排序(Join reordering) — 在 Reducer 的内存中，相对大的表被流化而不是被具体化，同时相对小的表被保存在内存中。这确保了在 Reducer 中，连接操作不会内存溢出。

此外，对于 MAPJOIN 提示来说，用户也可以提供提示或者集参数来完成以下功能：

i. 数据重分区，处理在 GROUP BY 中偏斜处理 — 很多现实世界中的数据集在使用 GROUP BY 进行普通查询是会在列上出现幂此定律。比如有这样的情况，普通的按照列将数据分配到组然后在 Reducer 中聚合，并不会正常工作，因为大部分数据都被发送到了极少数的 Reducer。那么，在这种情况下，我们需要有这样一个更好的计划，使用两次 Map/Reduce 步骤来处理聚合。第一步，数据被随机的分散(或者，如果出现 distinct 聚合，数据被散到 DISTINCT 列)到 Reducer 并且部分聚合被估算。这些部分聚合之后被分散到 GROUP BY 列，之后在第二次 Map/Reduce 中再次被分散到 Reducer。一旦这些部分聚合元组的数目比基础数据集的数目少，那么这种方式将会产生更好的效率。Hive 中，我们可以通过以下方式设置参数来触发以上行为：

```
Set hive.groupby.skewindata=true;

SELECT t1.c1, sum(t1.c2)
```

FROM t1

FROUP BY t1;

ii. Mapper 中基于散列的部分聚合 — 基于散列的部分聚合能够减少由 Mapper 发送至 Reducer 的数据量。这与减少在数据分类和合成的时间相一致。因此我们可以通过这样的方式来提高性能。Hive 允许用户控制 Mapper 所持有的用于优化的存储在散列表中的列的内存大小。Hiding 了以下参数 hive.map.aggr.hash.percentmemory 指定了 Mapper 持有的散列表的内存的大小，比如，0.5 当散列表的大小超过 Mapper 内存最大值的一半时，存储于该内存中的部分聚合数据将会尽快发送至 Reducer。以下参数 hive.map.aggr.hash.min.reduction 也被用于控制 Mapper 中内存总数。

..物理生成计划(Generation of the physical plan) —优化部分最后阶段生成的逻辑计划之后被分段到多个 Map/Reduce 和 HDFS 任务中。比如，在编写数据上进行分组的数据可以在最后的 HDFS 任务，该任务将结果移动到 HDFS 中正确的位置中，之后生成两个 Map/Reduce 任务。在该步骤的最后阶段，物理计划就像很多个 DAG 任务，每个任务封装了该计划的一部分。

下面，我们展示一个简单的多表插入查询和它在所有优化之后的相应物理计划：

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')

SELECT subq1.gender, COUNT(1)

GROUP BY subq1.gender

INSERT OVERWRITE TABLEschool_summary

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

SELECT subq1.school, COUNT(1)

GROUP BY subq1.school

该查询在两个不同的聚合之后进行单连接。通过编写多表连接查询，我们确保该连接只会执行一次。该查询的计划如下图 Fig3 所示。

计划中的节点是物理操作者[operators]，边缘代表了不同操作者之间的数据流。每个节点最后一行表示该操作的输出模式。对于空缺的地方，我们没有描述在不同操作节点中指定的参数。该计划有 3 个 Map/Reduce 工作。

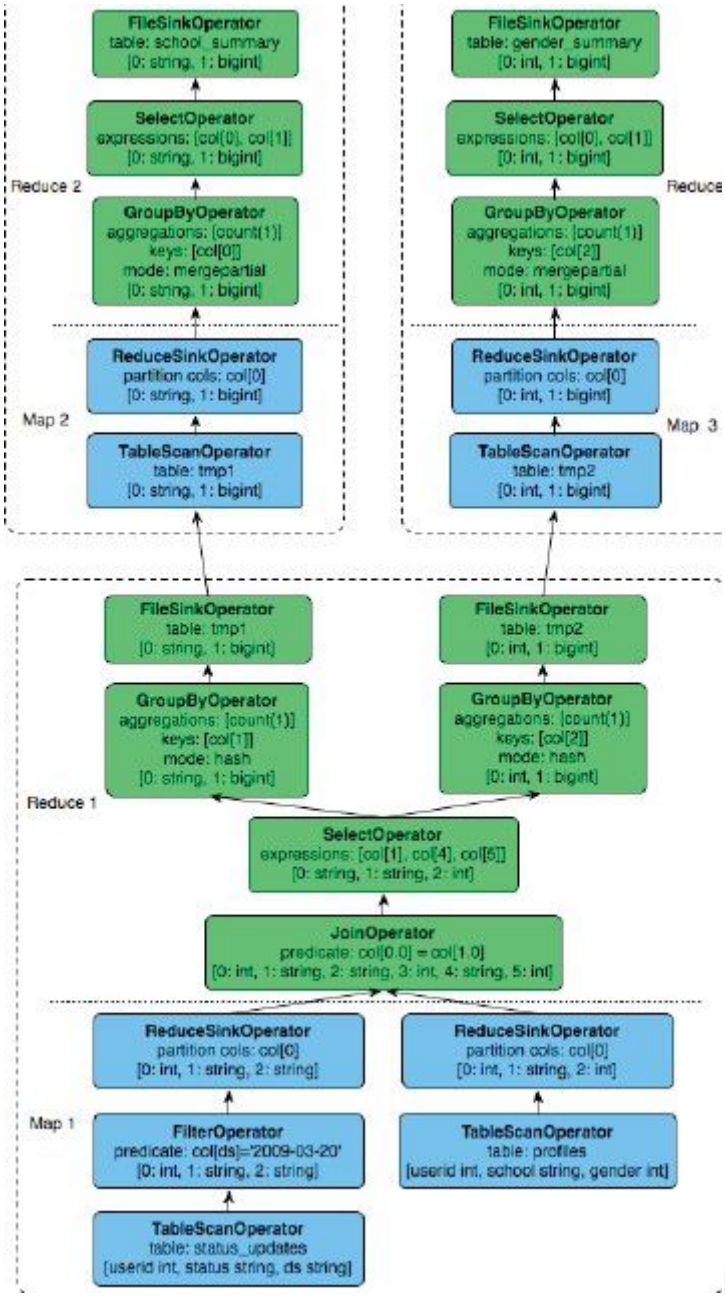


图 3.插入查询的 3 次 Map/Reduce 计划

在相同的 Map/Reduce 工作中重分区操作(ReduceSinkOperator)下的操作树部分是由 Mapper 所执行的并且 the portion above by reducer.自我重分区由 Execution Engine 执行。

注意，第一次 Map/Reduce 工作被写入两个临时 HDFS 文件，tmp1 和 tmp2,这两个文件将在第二次和第三次 Map/Reduce 工作中分别使用。因而，第二次和第三次 Map/Reduce 个工作都需要等待第一次工作的完成。

C．执行引擎(Execution Engine)

所有的任务将按照依赖顺序执行。每一个依赖任务都必须在起前提要求全部完成之后才能被执行。一个 Map/Reduce 任务首先将计划的一部分序列化进入 plan.xml 文件。这个文件之后将被添加到工作缓存中为 ExecMapper 所使用，并且 ExecReducer 会使用 Hadoop 生成。每个类序列化 plan.xml 文件并且执行相关的 DAG 操作部分。最终的结果将被存储在临时位置当中。在整个查询的最后部分，最终数据将被需要的位置，在 DMLs 例子下，数据最终被如上述一样的方式处理。

V．Hive 在 Facebook 中的使用

Hive 和 Hadoop 在 Facebook 各类数据处理中使用得非常广泛。目前，我们的数据仓库拥有 700TB 的数据(2.1PB 原始数据在 Hadoop 中,使用 3 种方式复制)。我们每天增加 5TB 的压缩数据(复制后占 15TB)，标准的压缩比是 1：7 并且有些时候更高。每天有超过 7500 个工作向集群提交并且超过 75TB 的压缩数据被处理。我们看到，在 Facebook 网络的持续扩大下，数据也在持续增加。与此同时，公司不仅要具有可伸缩性，集群也需要与增长的用户具有可伸缩性。

超过一半的工作量是即席查询，就像重置报告仪表板一样。在 Facebook，Hive 允许在 Hadoop 集群上处理这些类型的工作，因为简单的即席分析能够这样被完成。然而，由于不可预期的即席工作，因而，即席用户与报告用户共享相同资源变成了一个非常重大的业务困难。许多时候，这些工作并没有被恰当的调整并且消耗了宝贵的集群资源。这样导致了报告查询执行效率的降低，然后许多这些查询都对时间要求是非常苛刻的。资源规划对于 Hadoop 来说有那么一点无力并且目前唯一可行的解决方案是为即席查询和报告查询分别维护集群。

每天，也有许多多样化的工作在 Hive 中运行，其中有简单的词汇汇总和多位数据收集，也有更高级的机器学习算法。系统不仅有初级用户也有高级用虎，他们有间接使用的也有在几小时培训后就开始使用的。

种繁重的使用使得数据仓库中增加了大量的表，并且这种有序地大量地增加需要数据探索工具，尤其是对有新用户来说。总的来说，系统允许我们向引擎提供处理服务并且一部分分析师对传统的数据仓库基础构建有更高的消耗。为了具有可伸缩性能力，我们添加了上千的普通节点，这使得我们具有了可伸缩性基础构建的信心。

VI. 相关工作

目前有很多的工作都在 PB 级的开源或收费的数据处理系统上进行处理。Scope[14]是一个类 SQL 语言的由微软持有版权的 Map/Reduce 及分布式文件系统。Pig[13]允许用户编写声明式脚本来处理数据。Hive 与其他系统不同之处在于它提供了一个系统目录来是持久化与系统中与表相关的元数据。这是的 Hive 能够作为一个传统的数据仓库，并具有与标准报告工具，如 MicroStrategy[16]，相接口的的功能。HadoopDB[15]重用了大部分 Hive 系统，除了它使用传统的数据库实例替分布式文件系统代节点来存储数据。

VII. 总结及展望

Hive 正处于发展之中。它是一个开源项目，并且它由包括 Facebook 及其他外部组织提供支持和贡献。

HiveQL 当前只能接受 SQL 的子集来进行有效查询。我们目前致力于将 SQL 语法包含进 HiveQL 语法。Hive 目前有一个基于规则的本地优化器(Optimizer),这些规则数量并不大。我们计划构建一个基于开销(cost-based)的优化器，并且能与高级的优化技术想适应，从而使得计划更加有效率。我们正在探索柱状存储和更智能化的数据配置，以提高扫描性能。我们的运行基准是基于管理进度和与其他系统比较的。以我们的初步经验，我们能够提高 Hadoop 自身 20%的性能[9]。这种性能提升涉及使用更快的 Hadoop 数据结构来处理数据，比如，使用 Text 而不是 String。同样的查询在我们优化过的 Hive 实现中，HiveQL 具有 20%性能提升，比如，Hive 的性能与 Hadoop 的代码出处性能相同。我们同样执行行业标准 —TPC-H[11]。基于这些经验，我们找到了几个需要进行性能提升的区域并且开始着手。更多的详细资料请参见[10]和[12]。为了让 Hive 与商用 BI 工具集成，我们正在传统的关系型数据仓库下增强 JDBC 和 ODBC 驱动。我们也正在为单一 Map/Reduce 工作探索多查询优化技术和执行通用 N 路连接的方法