Introduction

Hive is a framework designed for data warehousing that runs on top of Hadoop. It enables users to run queries on the huge volumes of data. Its basic function is to convert SQL queries into MapReduce jobs.

Installation

Prerequisites:

- Java 6
- Cygwin (for Windows only).
- Hadoop version 0.20.x.

1. Download stable release at http://hive.apache.org/downloads.html
2. Unpack a tarball at suitable place:

```
% tar xzf hive-z.y.z-bin.tar.gz
```

3. Set environment:

```
cd hive-z.y.z-bin/bin

export HIVE_HOME=$PWD

export PATH=$HIVE_HOME/bin:$PATH
```

The Hive Shell

The shell is the primary way to interact with Hive by issuing commands in HiveQL which is a dialect of SQL. To list TABLES (in order to see if HIVE works):

```
hive> SHOW TABLES;
OK
Time taken: 10.425 seconds
```

For fresh install, the command takes a few seconds to run since it is lazily creating the metastore database on your machine.

In order to run Hive shell in non-interactive mode use –f switch and provide name of file.

```
% hive -f script.q
```

For short scripts you can use -e option (the final semicolon is not required):

```
% hive -e 'SELECT * FROM dummy'
```

In both interactive and non-interactive modes, Hive will print information to standard error. You can suppress these messages using the -S option.

RUNNING HIVE

Hive uses Hadoop so you must have hadoop in your path or run the following:

```
export HADOOP_HOME=<hadoop-install-dir>
```

In addition, you must create */tmp and /user/hive/warehouse* (aka hive.metastore.warehouse.dir) and set them *chmod g+w* in HDFS before you can create a table in Hive.

```
$ <hadoop-directory>/bin/hadoop fs -mkdir       /tmp
$ <hadoop-directory>/bin/hadoop fs -mkdir       /user/hive/warehouse
$ <hadoop-directory>/bin/hadoop fs -chmod g+w   /tmp
$ <hadoop-directory>/hadoop fs -chmod g+w   /user/hive/warehouse
```

You can also set HIVE_HOME

```
$ export HIVE_HOME=<hive-install-dir>
```

RUNNING HCATALOG

```
$HIVE_HOME /hcatalog/hcatalog/sbin/hcat_server.sh$
$HIVE_HOME/hcatalog/sbin/hcat_server.sh
```

RUNNING WEBHCat (Templeton)

To run the WebHCat server from the shell in Hive release 0.11.0 and later:

```
$HIVE_HOME/hcatalog/sbin/webhcat_server.sh
```

CONFIGURATION

Hive by default gets its configuration from <install-dir>/conf/hive-default.xml.The location of the Hive configuration directory can be changed by setting environment variable:

```
HIVE_CONF_DIR
```

They can be changed by (re)defining them in

```
<install-dir>/conf/hive-site.xml.
```

Log4j configuration is stored in

```
<install-dir>/conf/hive-log4j.properties
```

RUNTIME CONFIGURATION

Hive queries are executed using map-reduce queries, therefore the behavior of such queries can be controlled by the Hadoop configuration variables.
The CLI command 'SET' can be used to set any Hadoop (or Hive) configuration variable

```
hive> SET
mapred.job.tracker=myhost.mycompany.
com:50030;
hive> SET -v;
```

The second command shows all the current settings. Without the -v option only the variables that differ from the base Hadoop configuration are displayed.

HIVE, MAP-REDUCE AND LOCAL-MODE

Hive compiler generates map-reduce jobs for most queries. These jobs are then submitted to the Map-Reduce cluster indicated by the variable

```
mapred.job.tracker
```

This points to a map-reduce cluster with multiple nodes, Hadoop also offers an option to run map-reduce jobs locally on the user's workstation. To enable local mode of execution, the user can enable the following option:

```
hive> SET mapred.job.tracker=local
```

Starting with release 0.7 Hive supports local mode execution. To enable this, the user can enable the followig option

```
hive> SET mapred.job.tracker = local
```

ERROR LOGS

Hive uses log4j for logging. By default logs are not emitted to the consol by the CLI. The default logging level is WARN for Hive releases prior to 0.13.0. Starting with Hive 0.13.0 , the default logging level is INFO. The logs are stored in the folder

```
/tmp/<user.name>/hive.log
```

HIVE SERVICES:

The Hive shell is only one of several services that you can run using the hive command. You can specify the service to run using the --service option:

```
% hive --service [name]
```

- *cli*
  The command line interface to Hive (the shell). This is the default service

- *hiveserver*
  Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages. Applications using the Thrift , JDBC and ODBC connectors need to run a Hive server to communicate with Hive. Set the HIVE_PORT environment variable to specify the port the server will listen on (defaults to 10,000)
  hwi
- *Hive Web Interface* - alternative to the shell. Use the following commands:

```
% export ANT_LIB=/path/to/ant/lib
% hive --service hwi
```

- *Jar*
  The Hive equivalent to hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.
- *metastore*
  By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process. Set the METASTORE_PORT environment variable to specify the port the server will listen on.

HIVE CLIENTS

If you run Hive as a server , then there are number of different mechanisms for connecting to it from applications:

- *Thrift Client*
  Makes it easy to run Hive commands from a wide range of programming language. Thrift bindings for Hive are available for C++, Java , PHP, Python and Ruby.
- *JDBC Driver*
  Hive provides a Type 4(pure Java) JDBC driver, defined in the class

```
org.apache.hadoop.hive.jdbc.HiveDriver
```

- *ODBC Driver*
  The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive. It is still in development so you should refer to the latest instructions on the hive.

PARTITIONS

Hive organizes tables into partitions - a way of dividing a table into coarse-grained parts based on the value of a partition column, such as date. Tables or partitions may be further subdivided into buckets, to give extra structure to the data thay may be used for more efficient queries. Partitions are defined at table creation time using the PARTITIONED BY clause, which takes a list of column definitions. For the hypothetical log files example, we might define a table with records comprising a timestamp and the log line itself.

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARITIONED BY (dt STRING , country STRING) ;
```

When we load data into a partitioned table, the partition values are specified explicitly:

```
LOAD DATA LOCAL INPATH
'input/hive/partitions/file1' INTO TABLE logs
PARTITION (dt='2001-01-01', country='GB');
```

At the file system level, partitions are simply nested subdirectories of the table directory. After loading few more files into the logs table, the directory structure might look like this:

```
/user/hive/warehouse/logs/dt=2010-01-01/country=GB/file1
/file2
/country=US/file3
 /dt=2010-01-02/country=GB/file4
/country=US/file5
/file6
```

We can ask Hive for the paritions in a table using:

```
hive> SHOW PARITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-2/country=G
dt=2001-01-02/country=US
```

The columns definitions in the PARTITIONED BY clause are full-fledged table columns, called partition columns; however, the data files do not contain values for these columns

since they are derived from the directory names. You can use partitions' columns in SELECT statements in the usual way. Hive performs input pruning to scan only the relevant partitions. For example:

```
SELECT ts, dt, line
FROM logs
WHERE country='GB';
```

will only scan file1, file2 and file4. Notice, too, that the query returns the values of the dt partition column, which Hive reads from the directory names since they are not in the data files.

BUCKETS

There are two reasons why to organize tables (or partitions) into buckets.

- to enable more efficient queries
- to make sampling more efficient

We use the CLUSTERED BY clause to specify the columns to bucket on and the number of buckets:

```
CREATE TABLE bucketed users (id INT,
name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

Here we are using the user ID to determine the bucket (which Hive does by hashing the value and reducing modulo the number of buckets), so any particular bucket will effectively have a random set of users in it. The data within a bucket may additionally be sorted by one or columns.

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4
BUCKETS;
```

To populate the bucketed table, we need to set the *hive.enforce.bucketing* property to true, sot that Hive knows to create the number of buckets declared in the table definitions. Then it is a matter of just using the INSERT command:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

We can sample the table by using the TABLESAMPLE clause, which restricts the query to a fraction of the buckets in the table rather than the whole table:

```
hive> SELECT * FROM bucketed_users
    > TABLESAMPLE (BUCKET 1 OUT OF 4 ON id);
      0 Nat
      4 Ann
```

Sampling a bucketed table is very efficient, since the query only has to read the buckets that match the TABLESAMPLE clause.

STORAGE FORMATS

There are two dimensions that govern table storage in Hive: the *row format* and the *file format.* The row format dictates how rows, and the fields in a particular row, are stored. In Hive parlance, the row format is defined by a SerDe, a portmanteau word for a Serializer-Deserializer .When you create a table with no ROW FORMAT or STORED AS clauses, the default format is delimited text, with a row per line. The default row delimiter is not a tab character, but the *Control-A* character from the set of ASCII control codes (it has ASCII code 1). The choice of *Control-A*, sometimes written as *^A* in documentation, came about since it is less likely to be a part of the field text than a tab character. There is no means for escaping delimiter characters in Hive, so it is important to choose ones that don't occur in data fields. The default collection item delimiter is a *Control-B* character, used to delimit items in an ARRAY or STRUCT, or key-value pairs in a MAP. The default map key delimiter is a *Control-C* character, used to delimit the key and value in a MAP. Rows in a table are delimited by a newline character.

Thus, the statement:

```
CREATE TABLE ...;
```

is identical to the more explicit

```
CREATE TABLE ...
        ROW FORMAT DELIMITED
            FIELDS TERMINATED By '\001'
            COLLECTION ITEMS TERMINATED by '\002'
            MAP KEYS TERMINATED BY '\003'
            LINES TERMINATED by '\n'
        STORED AS TEXTFILE;
```

BINARY STORAGE FORMATS: Sequence files and RCFiles

Hadoop's sequence file format is a general purpose binary format for sequences of records (key-value pairs). You can use sequence files in Hive by using the declaration

```
STORED AS SEQUENCE FILE in CREATE TABLE statement
```

One of the main benefits of using sequence files is their support for splittable compression. If you have a collection of sequence files that were created outside Hive, then Hive will read them with extra configuration. If, on the other hand, you want tables populated from Hive to use compressed sequence files for their storage, you need to set a few properties to enable compression:

```
hive> SET hive.exec.compress.output=true;
hive> SET mapred.output.compress=true;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
hive> INSERT OVERWRITE TABLE;
```

RCFIle(Record Columnar File) – is another binary storage format. They are similar to sequence files, except that they store data in a column-oriented fashion. RCFile breaks up the table into row splits, then within each split stores the values for each row in the first column, followed by the values for each row in the second column, and so on. In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time. Use the following CREATE TABLE clauses to enable column-oriented storage in Hive:

```
CREATE TABLE ...
ROW FORMAT SERDE,
'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
STORED AS RCFILE;
```

Example:

Let's use another SerDe for storage. We'll use a contrib SerDe that uses a regular expression for reading the fixed-width station metadata froma a text file

```
hive>CREATE TABLE stations (usaf STRING, wban STRING, name STRING) ROW
FORMAT SERDE'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES(
"input.regex" = "(\\d{6} (\\d{5}) (.{29}) .*"
);
hive>LOAD DATA LOCAL INPATH "inpt/ncdc/metadata/stations-fixed-width.txt"
INTO TABLE stations;
```

When we retrieve data from the table, the SerDe is invoked for deserialization, which correctly parses the fields for each row:

```
hive> SELECT * FROM stations LIMIT 4;
010000        99999 BOGUS NORWAY
010003        99999 BOGUS NORWAY
010010        99999 JAN MAYEN
010013        99999 ROST
```

MAD REDUCE SCRIPT

Using an approach like Hadoop Streaming, the TRANSFORM, MAP and REDUCE clauses make it possible to invoke an external script or program from Hive.

Example - script to filter out rows to remove poor quality readings.

```
import re
import sys
for line in sys.stdin:
        (year, temp, q) = line.strip().split()
        if (temp != "9999" and re.match("[01459]", q))
                print "%s\t%s" % (year, temp)
```

```
hive> ADD FILE /path/to/is_good_quality.py;
hive> FROM records2
      > SELECT TRANSFORM(year, temperature, quality)
      > USING 'is_good)quality.py'
      > AS year, temperature;

1949   111
1949   78
1950   0
1950   22
1950   -11
```

DDL OPERATIONS

Hive Data Definition Language is a dialect of SQL, that transforms SQL statements into MapReduce jobs. Documentation can be found at

https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

Examples:

Creating Hive Tables

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

Creates a table called pokes with two columns, the first being an integer and the other a string.

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

Creates a table called invites with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partition that a particular dataset is loaded into. By default tables are assumed to be of text input format and the delimiters are assumed to be ^A(ctrl-a).

Browsing through tables

```
hive> SHOW TABLES;
```

Lists all of the tables

```
hive> DESCRIBE invites;
```

Shows the list of columns

Altering and Dropping Tables

Table names can be changed and columns can be added or replaced.

```
hive> ALTER TABLE events RENAME TO 3koobecaf;
hive> ALTER TABLE pokes ADD COLUMNS (new_col INT);
hive> ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment');
hive> ALTER TABLE invites REPLACE COLUMNS
      >(foo INT, bar STRING, baz INT COMMENT 'baz replaces new_col2');
```

Note that REPLACE COLUMNS replaces all existing columns and only changes the table's schema, not the data. The table must use a native SerDe. REPLACE COLUMNS can also be used to drop columns from the table's schema:

```
hive> ALTER TABLE invites REPLACE COLUMNS
      > (foo INT COMMENT 'only keep the first column');
```

DROPPING TABLES

```
hive> DROP TABLE pokes
```

DML OPERATIONS

The Hive DML operations are documented in:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML
Examples:

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

Loads a file that contains two columns separated by ctr-a into pokes table.
'LOCAL' signifies that the input is on the local file system. If 'LOCAL' is omitted then it looks for the file in HDFS.
The keyword 'OVERWRITE' signifies that existing data in the table is deleted. If the 'OVERWRITE' keyword is omitted, data files are appended to existing data sets.

```
hive> LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invite
PARTITION (ds='2008-08-15');
hive> LOAD DATA LOCAL INPATH './examples/files/kv3.txt' OVERWRITE INTO TABLE invites
PARTITION (ds='2008-08-08');
```

The two LOAD statements above load data into two different partitions of the table invites.
Table invites must be created as partitioned by the key ds for this to succeed.

```
hive> LOAD DATA INPATH '/user/myname/kv2.txt';
hive>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.

SQL OPERATIONS

The Hive query operations are documented in:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Select

Examples:

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

Selects column 'foo' from all rows of partition ds=2008-08-15 of the invites table. The results are not stored anywhere, but are displayed on the console.

```
hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM
        invites a WHERE a.ds='2008-08-15';
```

Partitioned tables must always have partition selected in the WHERE clause of the statement.

```
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.

WEATHER DATA EXAMPLE

Create table to hold the weather data using the CREATE TABLE statement

```
Hive> CREATE TABLE records (year STRING, temperature INT , quality INT)
    >  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

ROW FORMAT clause is particular to HiveQL. It says that each row in the data file is tab-delimited text.

For exploratory purposes let's populate records with data from *sample.txt*

```
1950   0      1
1950   22     1
1950   -11    1
1949   111    1
1949   78     1
```

sample.txt

```
hive > LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
       > OVERWRITE INTO TABLE records;
```

In this example, we are storing Hive tables on the local filesystem (fs.default.name is set to its default value of file:///). Tables are stored as directories under Hive's warehouse directory, which is controlled by the hive.metastore.warehouse.dir, and defaults to /user/hive/warehouse. Thus, we can run Linux command:

```
% ls /user/hive/warehouse/records/sample.txt
```

The OVERWRITE keyword in the LOAD DATA statement tells Hive to delete any existing files in the directory for the table. If it is omitted, then the new files are simply added to the table's directory (unless they have the same names, in which case they replace the old files).

After loading data we can run a query.

```
hive> SELECT year , MAX(temperature)
      >  FROM records
      > WHERE temperature != 9999
      > AND (quality = 0 OR quality = 1 OR quality = 4 quality = 5 OR quality = 9)
      > GROUP BY year;
        1949 111
        1950  22
```

What is remarkable is that Hive transforms this query into a MapReduce job , which it executes on our behalf, then prints the results to the console. We see that Hive is able to execute SQL queries against raw data stored in HDFS.

USER-DEFINED FUNCTIONS

User Defined Function (UDF) has to be written in Java, the language that Hive itself is written in. There are three types of UDF in Hive:

- *regular UDFs* - operates on a single row and produces a single row as its output, most functions, such as mathematical functions and string functions, are of this type

- *UDAFs (user-defined aggregate functions)* - works on multiple input rows and creates single output row, include COUNT and MAX functions
- *UDTFs (user-defined table-generating functions)* - operates on a single row and procudes multiple rows a table - as output

TABLE GENERATING FUNCTION EXAMPLE

Consider a table with a single column, x, which contains arrays of strings.

```
CREATE TABLE arrays (x ARRAY<STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002';
```

The example file has the following contents, where ^B is a representation of the Control-B character to make it suitable for printing:

```
a^B
b^B
c^B
d^B
e^B
```

example.txt

After running a LOAD DATA command, the following query confirms that the data was loaded correctly:

```
hive> SELECT * FROM arrays;
        [ "a", "b"]
        ["c", "d", "e"]
```

Next, we can use the explode UDTF to transform this table. This function emits a row for each entry in the array, so in this case the type of the output column y is STRING. The result is that the table is flattened into five rows.

```
hive> SELECT explode(x) AS y FROM arrays;
        a
        b
        c
        d
        e
```

EXAMPLE OF UDF –simple UDF to trim characters.

```
package com.hadoopbook.hive
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.TEXT;
public class Strip extends UDF {
private Text result = new Text();
public Text evaluate(Text str) {
              if(str == null) {
              return null; }
              result.set(StringUtils.strip(str.toString()));
              return result; }

       public Text evaluate(Text str, String stripChars) {
              if(str == null) {
                     return null;
              }
              result.set(StringUtils.strip(str.toString(),
stripChars))
              return result;
              }
}
```

UDF – strip.

The evaluate() method is not defined by an interface, so it may take an arbitrary number of arguments, of arbitrary types, and it may return a value of arbitrary type. To use the UDF in Hive, we need to package the compiled Java class in a JAR file and register the file with Hive.

```
ADD JAR /path/to/hive/hive-example.jar;
```

We also need to create an alias for the Java classname.

```
CREATE TEMPORARY FUNCTION strip AS 'com.hadoopbook.hive.Strip';
```

The UDF is now ready to be used, just like a built-in function:

```
hive> SELECT strip(' bee ') FROM dummy;
        bee
hive> SELECT strip ('banana' , 'ab') FROM dummy;
        nan
```

Notice that the UDF's name is not case-sensitive.

EXAMPLE - function for calculating the maximum of a collection of integers (UDAF).

```
package com.hadoop.hive;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;

public class Maximum extends UDAF {

        public static class MaximumIntUDAFEvaluator implements UDAFEvaluator {

                private IntWritable result;

                public void init() {
                        result = null;
                }

                public boolean iterate(IntWritable value) {
                        if (value == null) {
                                return true;
                        }

                        if (result == null) {
                                result = new IntWritable(value.get());
                        } else {
                                result.set(Math.max(result.get(), value.get()));
                        }
                        return true;
                }

                public IntWritable terminatePartial() {
                        return result;
                }

                public boolean merge(IntWritable other) {
                        return iterate(other);
                }

                public IntWritable terminate() {
                        return result;
                }

        }

}
```

A UDAF must be a subclass of *org.apache.hadoop.hive.ql.exec.UDAF* (note the "A" in UDAF) and contain one or more nested static classes implementing *org.apache.hadoop.hive.ql.exec.UDAFEvaluator*.

Lets' exercise our new function:

```
hive> CREATE TEMPORARY FUNCTION maximum AS 'com.hadoopbook.hive.Maximum';
hive> SELECT maximum(temperature) FROM records;
        110
```

IMPORTING DATA SETS

The easiest way to import dataset from relational database into Hive, is to export database from table to CSV file. After this is accomplished you should create table in hive:

```
hive> CREATE TABLE SHOOTING (arhivesource string, text string, to_user_id string, from_user string, id string, from_user_id string , iso_language_code string, source string , profile_image_url string, geo_type string, geo_coordinates_0 double, geo_coordinates_1 double, created_at string, time int, month int, day int, year int) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

DDL statement that creates Hive table. Notice that it has fields delimited by coma.

Next load data from local directory.

```
>hive LOAD DATA LOCAL INPATH '/dlrlhive/shooting/shooting.csv' INTO TABLE shooting;
```

CONCLUSIONS

The purpose of this tutorial is to give a user a brief and basic introduction to Hive and its SQL like features. For more in-depth information and instructions on how to explore Hive, please visit:

https://cwiki.apache.org/confluence/display/Hive/Home#Home-UserDocumentation