Applying Structure to Hadoop Data with Hive: Saying Hello to Hive, Seeing How the Hive is Put Together, Getting Started with Apache Hive, Examining the Hive Clients, Working with Hive Data Types, Creating and Managing Databases and Tables, Seeing How the Hive Data Manipulation Language Works, Querying and Analyzing Data

Saying Hello to Hive

Hive provides Hadoop with a bridge to the RDBMS world and provides an SQL dialect known as Hive Query Language (HiveQL), which can be used to perform SQL-like tasks. Hive also makes possible the concept known as enterprise data warehouse (EDW) augmentation, a leading use case for Apache Hadoop, where data warehouses are set up as RDBMSs built specifically for data analysis and reporting. Hive closely associated with RDBMS/EDW technology is extract, transform, and load (ETL) technology. For example, a company or an organization might extract unstructured text data from an Internet forum, transform the data into a structured format that's both valuable and useful, and then load the structured data into its EDW. Apache Hive gives you powerful analytical tools, all within the framework of HiveQL.

Seeing How the Hive is Put Together

In this section, we illustrate the architecture of Apache Hive and explain its various components, as shown in the illustration in Figure 1.

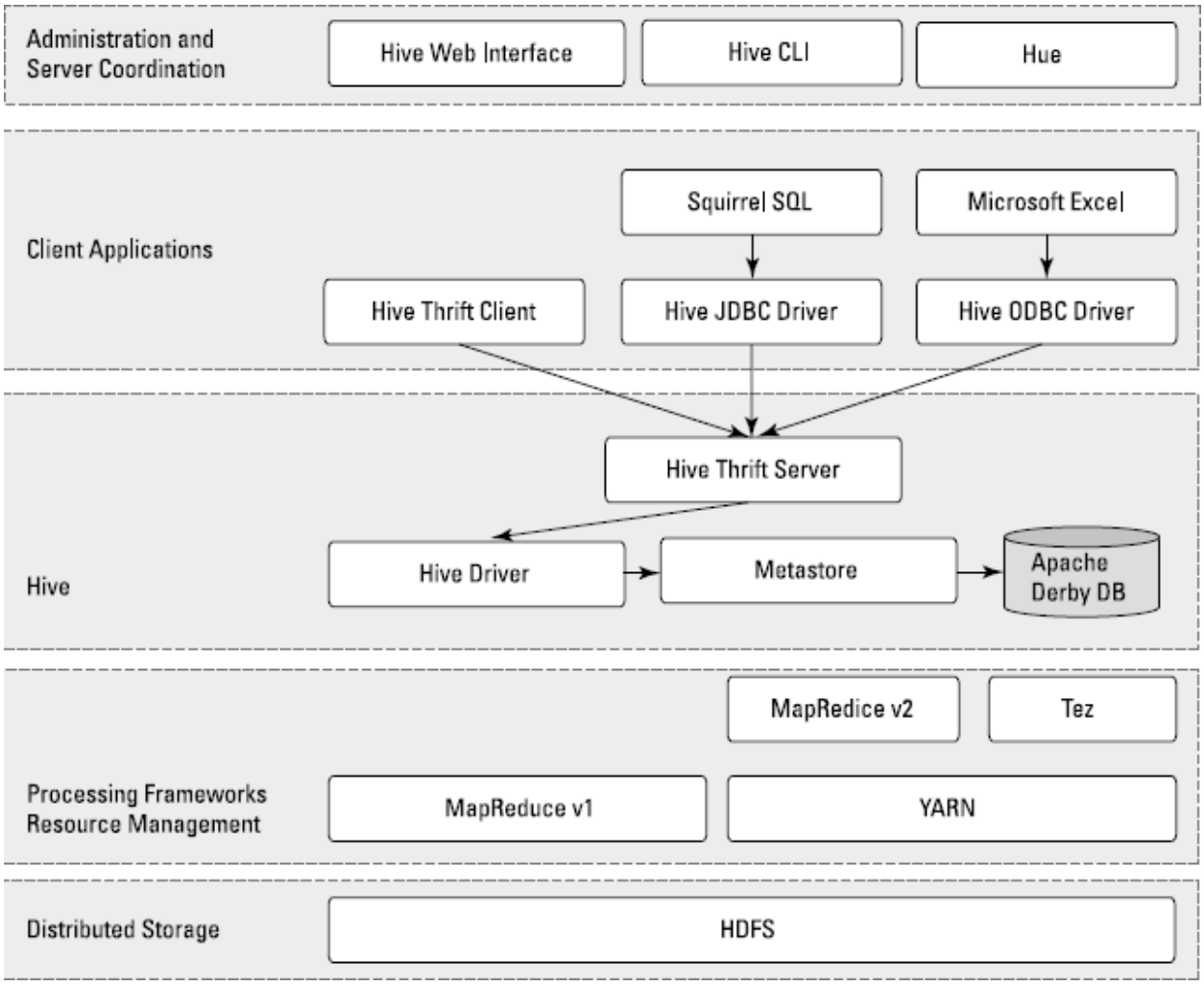


Figure 1: The Apache Hive architecture.

In the above figure we can see at the bottom that Hive sits on top of the Hadoop Distributed File System (HDFS) and MapReduce systems. In the case of MapReduce, Figure 1 shows both the Hadoop 1 and Hadoop 2 components. With Hadoop 1, Hive queries are converted to MapReduce code and executed using the MapReduce v1 (MRv1) infrastructure, like the JobTracker and TaskTracker. With Hadoop 2, YARN has decoupled resource management and scheduling from the MapReduce framework. Hive queries can still be converted to MapReduce code and executed, now with MapReduce v2 (MRv2) and the YARN infrastructure. There is a new framework under development called Apache Tez, which is designed to improve Hive performance for batch-style queries and support smaller interactive (also known as *real-time*) queries. HDFS provides the storage, and MapReduce provides the parallel processing capability for higher-level functions within the Hadoop ecosystem.

Moving up the diagram, you find the Hive Driver, which compiles, optimizes, and executes the HiveQL. The Hive Driver may choose to execute HiveQL statements and commands locally or spawn a MapReduce job, depending on the task at hand. By default, Hive includes the Apache Derby RDBMS configured with the metastore in what's called embedded mode. *Embedded mode* means that the Hive Driver, the metastore, and Apache Derby are all running in one Java Virtual Machine (JVM). This configuration is fine for learning purposes, but embedded mode can support only a single Hive session, so it normally isn't used in multi-user production environments. Two other modes exist — *local* and *remote* — which can better support multiple Hive sessions in production environments. Also, you can configure any RDBMS that's compliant with the Java Database Connectivity (JDBC) Application Programming Interface (API) suite.

The key to application support is the Hive Thrift Server, which enables a rich set of clients to access the Hive subsystem. The main point is that any JDBC-compliant application can access Hive via the bundled JDBC driver. The same statement applies to clients compliant with Open Database Connectivity (ODBC) — for example, unixODBC and the isql utility, which are typically bundled with Linux, enable access to Hive from remote Linux clients. Additionally, if you use Microsoft Excel, you'll be pleased to know that you can access Hive after you install the Microsoft ODBC driver on your client system. Finally, if you need to access Hive from programming languages other than Java (PHP or Python, for example), Apache Thrift is the answer.

Apache Thrift clients connect to Hive via the Hive Thrift Server, just as the JDBC and ODBC clients do. Hive includes a Command Line Interface (CLI), where you can use a Linux terminal window to issue queries and administrative commands directly to the Hive Driver. If a graphical approach is more your speed, there's also a handy web interface so that you can access your Hive-managed tables and data via your favourite browser.

Getting Started with Apache Hive

We are running Hive in stand-alone mode rather than in a real-life Apache Hadoop cluster, configure the system to use local storage rather than the HDFS: Simply set the hive.metastore.warehouse.dir parameter. When you start a Hive client, the $HIVE_HOME environment variable tells the client that it should look for our configuration file (hivesite.xml) in the conf directory. If you already have a Hadoop cluster configured and running, you need to set the hive.metastore.warehouse.dir configuration variable to the HDFS directory where you intend to store your Hive warehouse, set the mapred.job.tracker configuration variable to point to your Hadoop JobTracker, and set up a distributed metastore.

Examining the Hive Clients

There are quite a number of client options for Hive as below.

1. Hive command-line interface (CLI)
2. Hive Web Interface (HWI) Server
3. Open source SQuirreL client using the JDBC driver.

The Hive CLI client

The Figure 2. Shows components that are required when running the CLI on a Hadoop cluster.

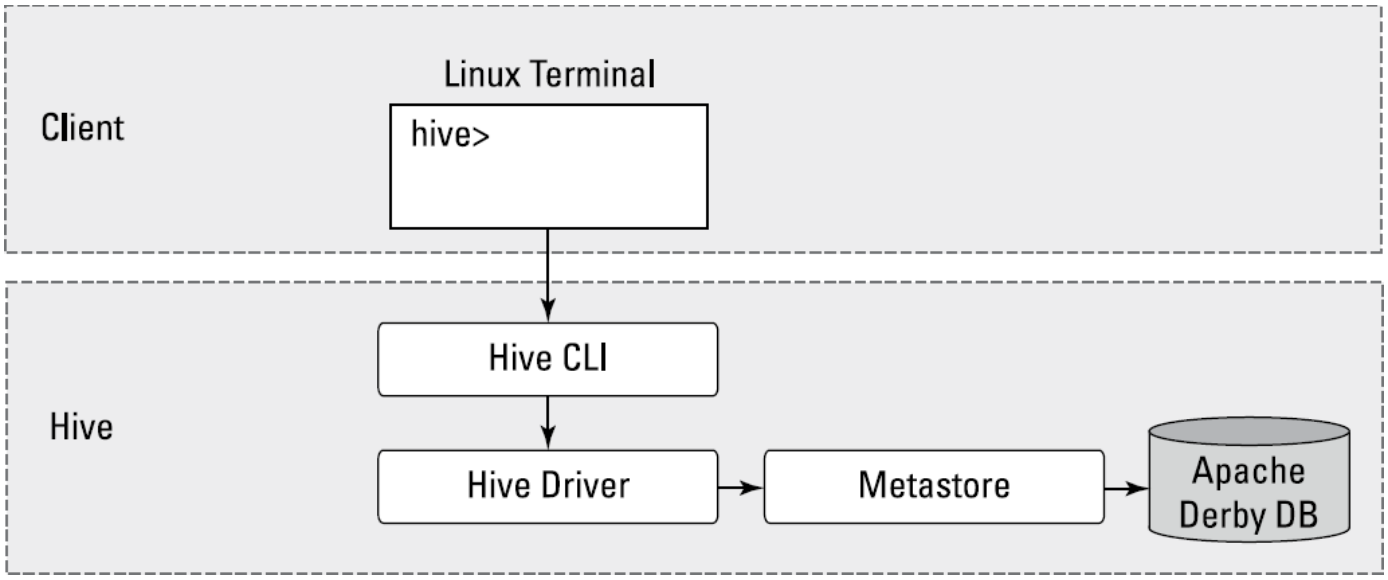


Figure 2: The Hive command line interface mode.

The examples in this chapter, we run Hive in local mode, which uses local storage, rather than the HDFS, for your data. To run the Hive CLI, you execute the hive command and specify the CLI as the service you want to run. The following instructions shows some of our first HiveQL statements.

Command	Description
$ $HIVE_HOME/bin hive --service cli	• Starts the Hive CLI using the $HIVE_HOME environment variable. • The –service cli command-line option directs the Hive system to start the command-line
hive> set hive.cli.print.current.db=true;	• We tell the Hive CLI to print our current working database so that we know where you are in the namespace.
hive (default)> CREATE DATABASE ourfirstdatabase;	• HiveQL's to tell the system to create a database called ourfirstdatabase.
hive (default)> USE ourfirstdatabase;	• Make this database as the default for subsequent HiveQL DDL commands
hive (ourfirstdatabase)> CREATE TABLE our_first_table (> FirstName STRING, > LastName STRING, > EmployeeId INT);	• We create our first table and give it the name our_first_table.
$ ls /home/biadmin/Hive/warehouse/ourfirstdatabase.db our_first_table	• Hive warehouse directory that stores our_first_table on disk

Hive Web Interface (HWI) Server

When we want to access Hive using a web browser, you first need to start the Hive Web Interface (HWI) Server and then point your browser to the port on which the server is listening. Figure 3. Shows the HWI client configuration.

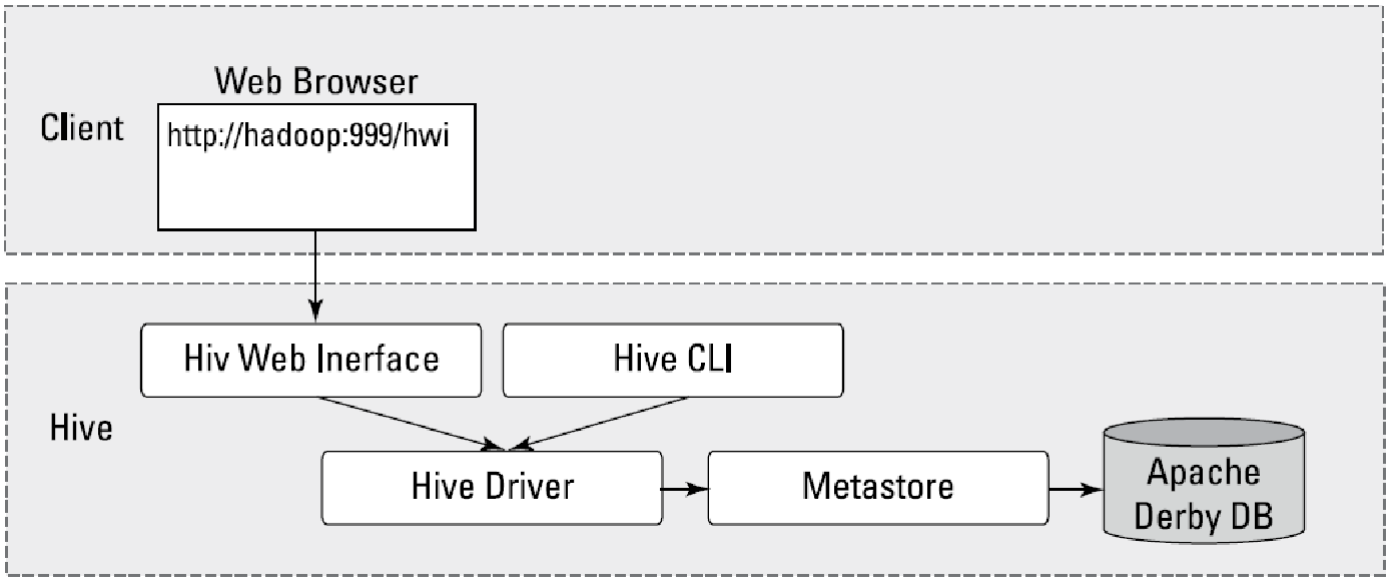


Figure 3: The Hive Web Interface client configuration.

The following steps show you what you need to do before you can start the HWI Server:

1. **Configure the** $HIVE_HOME/conf/hive-site.xml **file as below to ensure that Hive can find and load the HWI's Java server pages.**

   <property>
   <name>hive.hwi.war.file</name>
   <value>${HIVE_HOME}/lib/hive_hwi.war</value>
   <description>
   This is the WAR file with the jsp content for Hive Web Interface
   </description>
   </property>
2. **The HWI Server requires Apache Ant libraries to run, so download Ant from the Apache site at** http://ant.apache.org/bindownload.cgi.
3. **Install Ant using the following commands:**

   mkdir ant
   cp apache-ant-1.9.2-bin.tar.gz ant; cd ant
   gunzip apache-ant-1.9.2-bin.tar.gz
   tar xvf apache-ant-1.9.2-bin.tar
4. **Set the** $ANT_LIB **environment variable and start the HWI Server by using the following commands:**

   $ export ANT_LIB=/home/user/ant/apache-ant-1.9.2/lib
   $ bin/hive --service hwi

In addition to above in a production environment, you'd probably configure two other properties:

1. hive.hwi.listen.host: It is used to set the IP address of the system running your HWI Server
2. hive.hwi.listen.port: It is used to set the port that the HWI Server listens on.

Here we use the default settings: With the HWI Server now running, you simply enter the URL http://localhost:9999/hwi/ into your web browser and view the metadata for our_first_table as shown in Figure4.

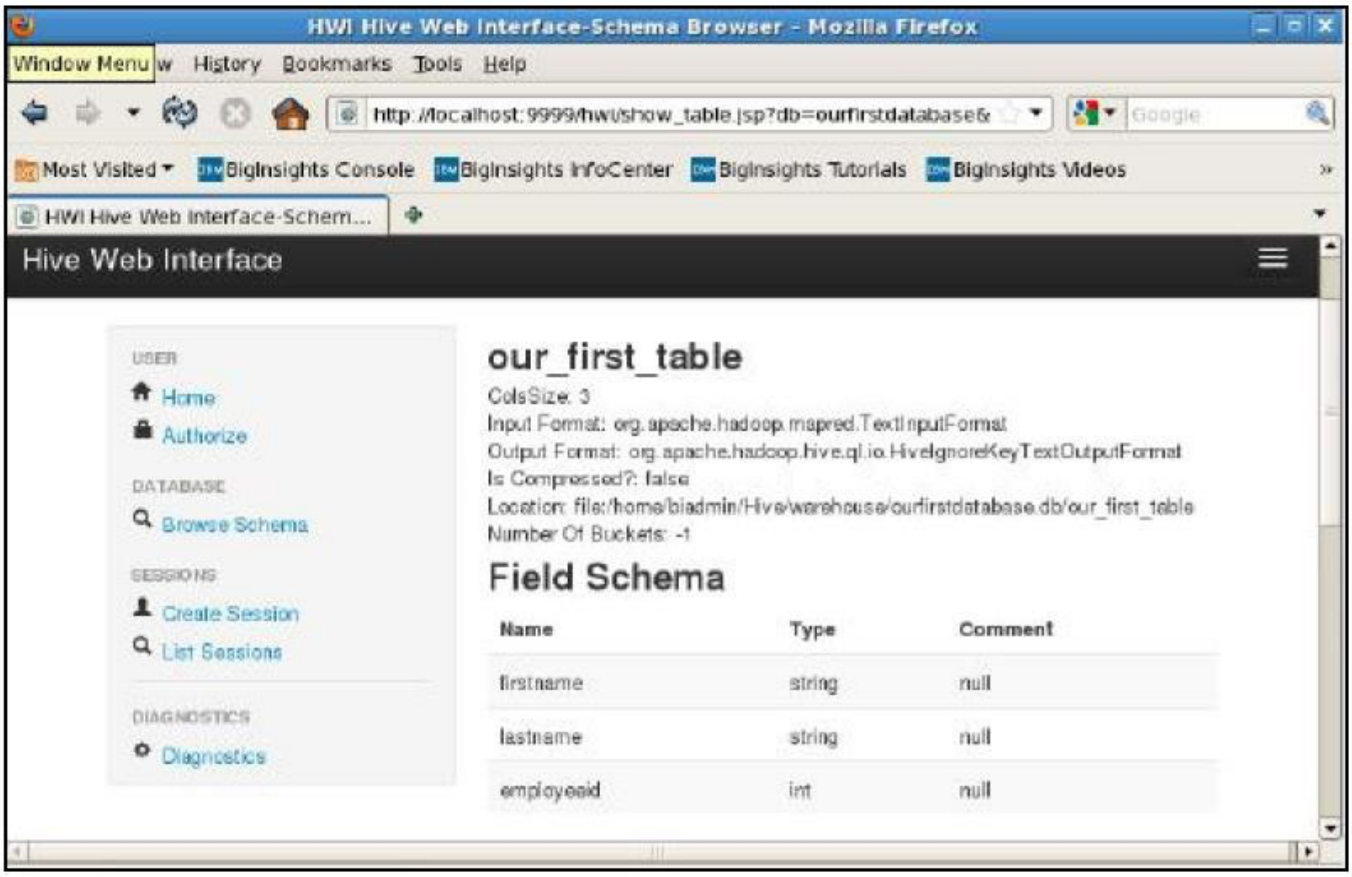


Figure 4: Using the Hive Web Interface to browse the metadata.

SQuirreL as Hive client with the JDBC Driver

The last Hive client is the open source tool SQuirreL SQL. It provides a user interface to Hive and simplifies the tasks of querying large tables and analyzing data with Apache Hive. Figure 5. illustrates how the Hive architecture would work when using tools such as SQuirreL.

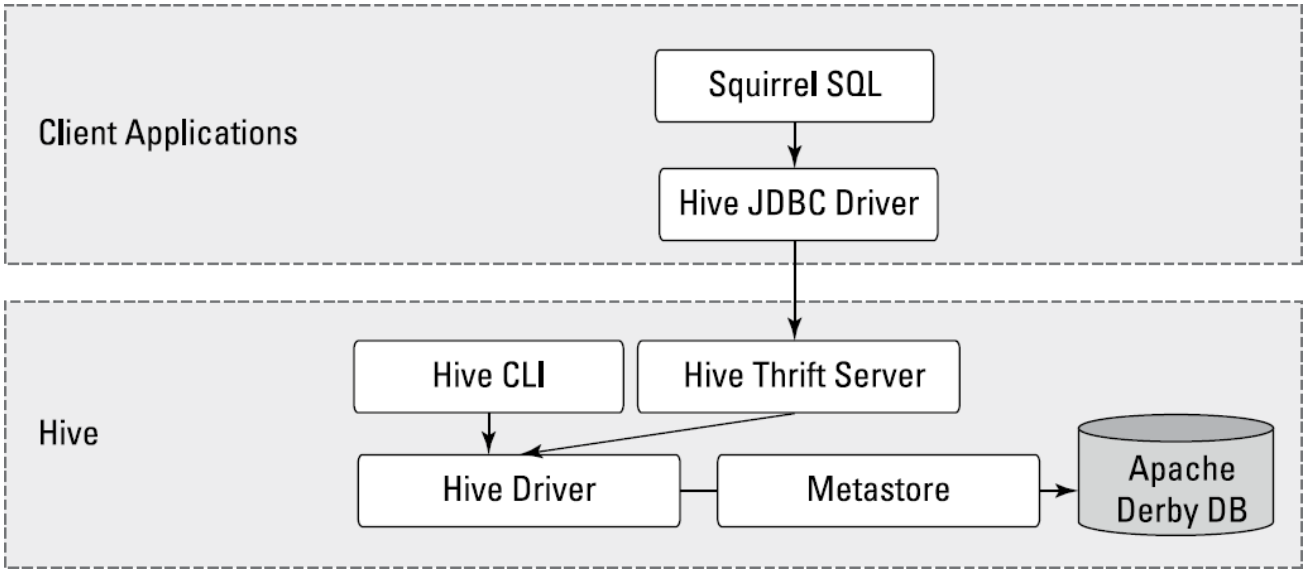


Figure 5: Using the SQuirreL client with Apache Hive.

In the above figure, we can see that the SQuirreL client uses the JDBC APIs to pass commands to the Hive Driver by way of the Hive Thrift Server. Figure 6. shows some HiveQL commands running against the Hive Driver —similar to the commands you ran earlier, with the CLI.

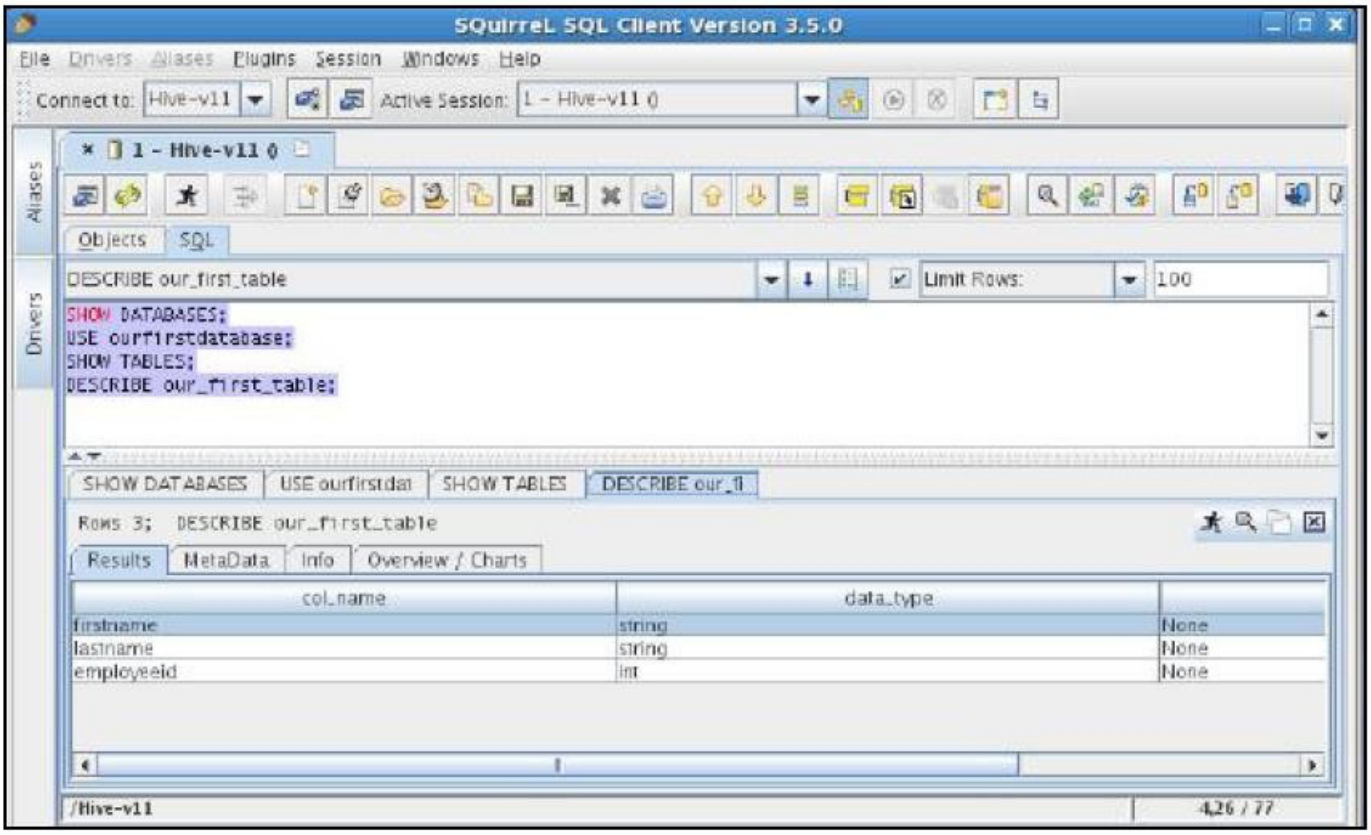


Figure 6: Using the SQuirreL SQL client to run HiveQL commands.

Working with Hive Data Types

The following list shows all Hive-supported data types.

```
$ ./hive --service cli
hive> CREATE DATABASE data_types_db;
OK
Time taken: 0.119 seconds
hive> USE data_types_db;
OK
Time taken: 0.018 seconds
(1) Hive> CREATE TABLE data_types_table (
(2)  > our_tinyint     TINYINT     COMMENT '1 byte signed integer',
(3)  > our_smallint    SMALLINT    COMMENT '2 byte signed integer',
(4)  > our_int         INT         COMMENT '4 byte signed integer',
(5)  > our_bigint      BIGINT      COMMENT '8 byte signed integer',
(6)  > our_float       FLOAT       COMMENT 'Single precision floating point',
(7)  > our_double      DOUBLE      COMMENT 'Double precision floating point',
(8)  > our_decimal     DECIMAL     COMMENT 'Precise decimal type based
(9)  >                                     on Java BigDecimal Object',
(10) > our_timestamp   TIMESTAMP   COMMENT 'YYYY-MM-DD HH:MM:SS.fffffffff"
(11) >                                     (9 decimal place precision)',
(12) > our_boolean     BOOLEAN     COMMENT 'TRUE or FALSE boolean data type',
(13) > our_string      STRING      COMMENT 'Character String data type',
(14) > our_binary      BINARY      COMMENT 'Data Type for Storing arbitrary
(15) >                                     number of bytes',
(16) > our_array       ARRAY<TINYINT>  COMMENT 'A collection of fields all of
(17) >                                     the same data type indexed BY
(18) >                                     an integer',
(19) > our_map       MAP<STRING,INT> COMMENT 'A Collection of Key,Value Pairs
(20) >                                     where the Key is a Primitive
(21) >                                     Type and the Value can be
(22) >                                     anything.  The chosen data
(23) >                                     types for the keys and values
(24) >                                     must remain the same per map',
(25) > our_struct    STRUCT<first : SMALLINT, second : FLOAT, third : STRING>
(26) >                                COMMENT 'A nested complex data
(27) >                                     structure',
(28) > our_union      UNIONTYPE<INT,FLOAT,STRING>
(29) >                                COMMENT 'A Complex Data Type that can
(30) >                                     hold One of its Possible Data
(31) >                                     Types at Once')
```

```
(32) > COMMENT 'Table illustrating all Apache Hive data types'
(33) > ROW FORMAT DELIMITED
(34) > FIELDS TERMINATED BY ','
(35) > COLLECTION ITEMS TERMINATED BY '|'
(36) > MAP KEYS TERMINATED BY '^'
(37) > LINES TERMINATED BY '\n'
(38) > STORED AS TEXTFILE
(39) > TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Sat Sep 21
            20:46:32 EDT 2013');
OK
Time taken: 0.886 seconds
```

Listing : HiveQL-Supported Data Types

Hive has primitive data types as well as complex data types. The last four columns (see Lines 16–31) in our_datatypes_table are complex data types: ARRAY, MAP, STRUCT, and UNIONTYPE. Line 32 allows us to add a comment for the entire table. Line 39 starts with the keyword TBLPROPERTIES, which provides a way for you to add metadata to the table. This information can be viewed later, after the table is created, with other HiveQL commands such as DESCRIBE EXTENDED table_name. Lines 33–38 in the CREATE TABLE statement specifies the file format when your table gets stored in HDFS and define how fields and rows are delimited.

Creating and Managing Databases and Tables

Creating, Dropping, and Altering Databases in Apache Hive is as below.

```
(1) $ $HIVE_HOME/bin hive --service cli
(2) hive> set hive.cli.print.current.db=true;
(3) hive (default)> USE ourfirstdatabase;
(4) hive (ourfirstdatabase)> ALTER DATABASE
          ourfirstdatabase SET DBPROPERTIES
          ('creator'='Bruce Brown',
          'created_for'='Learning Hive DDL');
OK
Time taken: 0.138 seconds
(5) hive (ourfirstdatabase)> DESCRIBE DATABASE EXTENDED
          ourfirstdatabase;
OK
ourfirstdatabase
          file:/home/biadmin/Hive/warehouse/
          ourfirstdatabase.db   {created_for=Learning
          Hive DDL, creator=Bruce Brown}
Time taken: 0.084 seconds, Fetched: 1 row(s)CREATE
          (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
(6) hive (ourfirstdatabase)> DROP DATABASE
          ourfirstdatabase CASCADE;
OK
Time taken: 0.132 seconds
```

In Line 4 of above instructions, we're now altering the database which have already created with the name ourfirstdatabase to include two new metadata items: creator and created_for. These two can be quite useful for documentation purposes and coordination within your working group. The command in Line 5 is used to view the metadata. With the help of command in Line 6 we're dropping the entire database — removing it from the server. We can use the DROP TABLE command to delete individual tables.

Creating and managing tables with Hive

Apache Hive lets you define the record format separately from the file format. Hive tables default to the configuration in below Listing unless you override the default settings.

```
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
```

The following listing specifies on how fields will be separated or delimited whenever you insert or load data into the table.

```
(1)Hive> CREATE TABLE data_types_table (
...
(33) > ROW FORMAT DELIMITED
(34) > FIELDS TERMINATED BY ','
(35) > COLLECTION ITEMS TERMINATED BY '|'
(36) > MAP KEYS TERMINATED BY '^'
(37) > LINES TERMINATED BY '\n'
(38) > STORED AS TEXTFILE
...
(39) > TBLPROPERTIES ('creator'='Bruce Brown',
'created_at'='Sat Sep 21 20:46:32 EDT 2013');
```

In the above listing Lines 33–37 define the Hive *row format* for our data_types_table. Line 38 defines the Hive *file format* — a text file — when the data is stored in the HDFS.

So far, we have been using the default TEXTFILE format for your Hive table records. However, as you know, text files are slower to process, and they consume a lot of disk space unless you compress them. For these reasons and more, the Apache Hive community came up with several choices for storing our tables on the HDFS.

File formats of Hive

The following list describes the file formats you can choose from as of Hive version 0.11.

- **TEXTFILE:** The default file format for Hive records. Alphanumeric characters from the Unicode standard are used to store your data.
- **SEQUENCEFILE:** The format for binary files composed of key/value pairs. Sequence files, which are used heavily by Hadoop, are often good choices for Hive table storage, especially if you want to integrate Hive with other technologies in the Hadoop ecosystem.
- **RCFILE:** RCFILE stands for *r*ecord *c*olumnar *file.* Stores records in a column-oriented fashion rather than a row-oriented fashion — like the TEXTFILE format approach
- **ORC:** ORC stands for *o*ptimized *r*ow *c*olumnar. A format (new as of Hive 0.11) that has significant optimizations to improve Hive reads and writes and the processing of tables. For example, ORC files include optimizations for Hive complex types and new types such as DECIMAL. Also lightweight indexes are included with ORC files to improve performance.

- **INPUTFORMAT, OUTPUTFORMAT:** INPUTFORMAT will read data from the Hive table. OUTPUTFORMAT does the same thing for writing data to the Hive table. To see the default settings for the table, simply execute a DESCRIBE EXTENDED *tablename* HiveQL statement and we'll see the INPUTFORMAT and OUTPUTFORMAT classes for your table.

Defining table record formats

The Java technology that Hive uses to process records and map them to column data types in Hive tables is called *SerDe,* which is short for *Ser*ializer*De*serializer. Figure 7 will help us to understand how Hive keeps file formats separate from record formats.

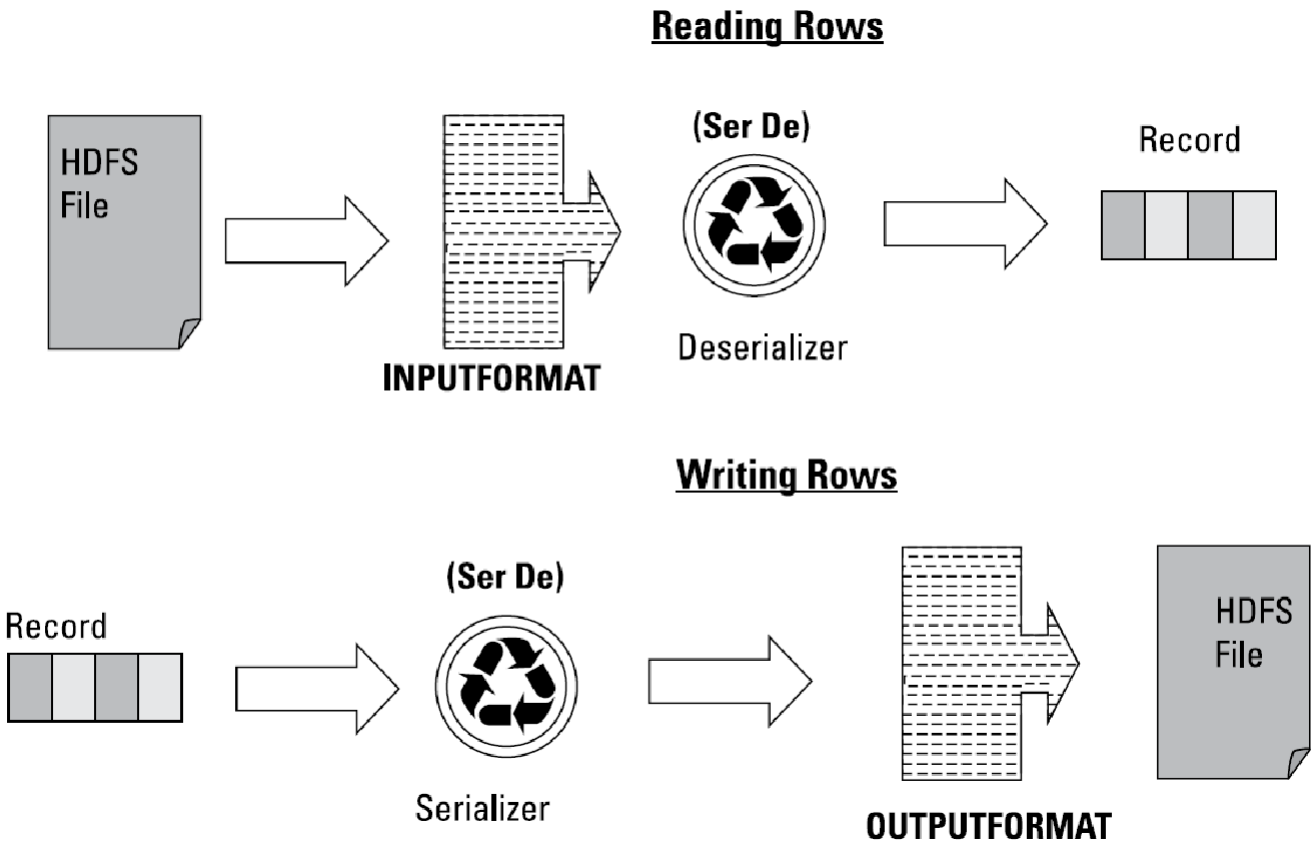


Figure 7: How Hive Reads and Writes Records

When Hive is reading data from the HDFS (or local file system), a Java Deserializer formats the data into a record that maps to table column data types. It is used at the time of HiveQL SELECT statement. When Hive is writing data, a Java Serializer accepts the record Hive uses and translates it such that the OUTPUTFORMAT class can write it to the HDFS (or local file system). It is used at the time of HiveQL CREATE-TABLE-AS-SELECT statement. So the INPUTFORMAT, OUTPUTFORMAT and SerDe objects allow Hive to separate the table record format from the table file format.

Hive bundles a number of SerDes for us. We can also develop your own SerDes if you have a more unusual data type that you want to manage with a Hive table. Some of those are specified as below.

LazySimpleSerDe: The default SerDe that's used with the TEXTFILE format;

ColumnarSerDe: Used with the RCFILE format.

RegexSerDe: RegexSerDe can form a powerful approach for building structured data in Hive tables from unstructured blogs, semi-structured log files, e-mails, tweets, and other data from social media. Regular expressions allow us to extract meaningful information.

HBaseSerDe: Included with Hive to enables it to integrate with HBase.

JSONSerDe: A third-party SerDe for reading and writing JSON data records with Hive.

AvroSerDe: Included with Hive so that you can read and write Avro data in Hive tables.

The following example shows us all of the options we've been discussing in this section.

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS]
          [db_name.]table_name
  ...  (Skipping some lines for brevity)
   [ROW FORMAT row_format] [STORED AS file_format]
   | STORED BY 'storage.handler.class.name' [WITH
          SERDEPROPERTIES (...)] ]
  ...  (Skipping some lines for brevity)
row_format
  : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY
          char]] [COLLECTION ITEMS TERMINATED BY char]
        [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY
          char] [NULL DEFINED AS char]
        | SERDE serde_name [WITH SERDEPROPERTIES
          (property_name=property_value, property_
          name=property_value, ...)]
file_format:
  : SEQUENCEFILE | TEXTFILE | RCFILE | ORC
  | INPUTFORMAT input_format_classname OUTPUTFORMAT
          output_format_classname
```

Tying it all together with an example

We want to tie things together in this section with two examples. In this first example, we revisit data_types_table from Listing. Here we leverage the DESCRIBE EXTENDED data_types_table HiveQL command to illustrate what Hive does with our CREATE TABLE statement under the hood.

```
hive> DESCRIBE EXTENDED data_types_table;
OK
our_tinyint             tinyint                 1 byte
          signed integer
our_smallint            smallint                2 byte
          signed integer
...
(A)inputFormat:org.apache.hadoop.mapred.TextInputFormat,
outputFormat:
(B)org.apache.hadoop.hive.ql.io.
          HiveIgnoreKeyTextOutputFormat
, ...
serializationLib:
  org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe,
(C)parameters:{collection.delim=|, mapkey.delim=^, line.
          delim=
(D), serialization.format= ,, field.delim=,}),
...
```

Seeing How the Hive Data Manipulation Language Works

Hive's data manipulation language (DML) allows us to load and insert data into tables and create tables from other tables.

LOAD DATA examples

Now we have to place data into the data_types_table with LOAD DATA command. The syntax for the LOAD DATA command is given below.

```
"LOAD DATA [LOCAL] INPATH 'path to file' [OVERWRITE] INTO
          TABLE 'table name' [PARTITION partition column1
          = value1, partition column2 = value2,...]
```

In the above syntax optional LOCAL keyword tells Hive to copy data from the input file on the local file system into the Hive data warehouse directory. Without the LOCAL keyword, the data is simply moved (not copied) into the warehouse directory. The optional OVERWRITE keyword, causes the system to overwrite data in the specified table if it already has data stored in it. Finally, the optional PARTITION list tells Hive to partition the storage of the table into different directories in the data warehouse directory structure. This powerful concept improves query performance in Hive, by Rather than run a MapReduce job over the entire table to find the data you want to view or analyze, you can isolate a segment of the table and save a lot of system time with partitions. The following Listing shows the commands to use to load the data_types_table with data.

```
(A) $ cat data.txt
100,32000,2000000,92000000000000000000,0.15625,4.9406564584
          124654,
1.23E+3,2013-09-21 20:19:52.025,true,
test string,\0xFFFFDDDDEEEEAAAA,1|2|3|4,key^1024,
1|3.1459|test struct,2|test union
(B) hive (data_types_db)> LOAD DATA LOCAL INPATH
          '/home/biadmin/Hive/data.txt' INTO TABLE
          data_types_table;
Copying data from file:/home/biadmin/Hive/data.txt
Copying file: file:/home/biadmin/Hive/data.txt
Loading data to table data_types_db.data_types_table
Table data_types_db.data_types_table stats:
          [num_partitions: 0, num_files: 1, num_rows: 0,
          total_size: 185, raw_data_size: 0]
OK
Time taken: 0.287 seconds
(C) hive> SELECT * FROM data_types_table;
OK
100     32000   2000000 92000000000000000000     0.15625
          4.940656458412465
1230    2013-09-21 20:19:52.025      true    test string
\0xFFFFDDDDEEEEAAAA     [1,2,3,4]       {"key":1024}
{"first":1,"second":3.1459,"third":"test struct"}
(D) {2:"test union"}
Time taken: 0.201 seconds, Fetched: 1 row(s)
```

Listing : Loading our_first_table with Data

In the above listing Step (A) shows listing of data you intend to load. This data file has only one record in it, but there's a value for each field in the table. As we specified at table creation time, fields are separated by a comma; collections (such as STRUCT and UNIONTYPE) are separated by the vertical bar or pipe character (|); and the MAP keys and values are separated by the caret character (^).Step (B) has the LOAD DATA command, and in Step (C) we're retrieving the record we just loaded in Step (B) so that we can view the data.

Example:

In the below listing we created two identical tables, named FlightInfo2007 and FlightInfo2008, as you can see in steps (A) and (F) in below Listing

```
(A) CREATE TABLE IF NOT EXISTS FlightInfo2007 (
  Year SMALLINT, Month TINYINT, DayofMonth TINYINT,
          DayOfWeek TINYINT,
  DepTime SMALLINT, CRSDepTime SMALLINT, ArrTime SMALLINT,
          CRSArrTime SMALLINT,
  UniqueCarrier STRING, FlightNum STRING, TailNum STRING,
  ActualElapsedTime SMALLINT, CRSElapsedTime SMALLINT,
  AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,
  Origin STRING, Dest STRING,Distance INT,
  TaxiIn SMALLINT, TaxiOut SMALLINT, Cancelled SMALLINT,
  CancellationCode STRING, Diverted SMALLINT,
  CarrierDelay SMALLINT, WeatherDelay SMALLINT,
  NASDelay SMALLINT, SecurityDelay SMALLINT,
          LateAircraftDelay SMALLINT)
COMMENT 'Flight InfoTable'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Thu
          Sep 19 10:58:00 EDT 2013');

(B) hive (flightdata)> LOAD DATA INPATH '/home/biadmin/
          Hive/Data/2007.csv' INTO TABLE FlightInfo2007;
Loading data to table flightdata.flightinfo2007
Table flightdata.flightinfo2007 stats: [num_partitions:
          0, num_files: 2, num_rows: 0, total_size:
          1405756086, raw_data_size: 0]
OK
Time taken: 0.284 seconds;
(C) hive (flightdata)> SELECT * FROM FlightInfo2007 LIMIT
          2;
OK
NULL    NULL    NULL    NULL    NULL    NULL    NULL
          NULL    UniqueCarrier   FlightNum       TailNum
          NULL    NULLNULL     NULL    NULL    Origin
          Dest    NULL    NULL    NULL    NULL
          CancellationCode        NULL    NULL
          NULLNULL     NULL    NULL
```

```
2007    1       1       1       1232    1225    1341
        1340    WN      2891    N351    69      75
        54      1   7SMF    ONT     389     4       11
        0               0       0       0       0
        0       0
Time taken: 0.087 seconds, Fetched: 2 row(s)

(D) LOAD DATA INPATH '/home/biadmin/Hive/Data/2007.csv'
        OVERWRITE INTO TABLE FlightInfo2007;
(E) hive (flightdata)>  SELECT * FROM FlightInfo2007 LIMIT
        2;
OK
2007    1       1       1       1232    1225    1341
        1340    WN      2891    N351    69      75
        54      1   7SMF    ONT     389     4       11
        0               0       0       0       0
        0       0
2007    1       1       1       1918    1905    2043
        2035    WN      462     N370    85      90
        74      8   13      SMF     PDX     479     5
        6       0               0       0       0
        0       0       0
Time taken: 0.089 seconds, Fetched: 2 row(s)

(F) CREATE TABLE IF NOT EXISTS FlightInfo2008 LIKE
        FlightInfo2007;
(G) LOAD DATA INPATH '/home/biadmin/Hive/Data/2008.csv'
        INTO TABLE FlightInfo2008;
```

Listing : Flight Information Tables from 2007 and 2008

In Step (B) of above listing we didn't use the LOCAL keyword. That's because these files are large; you'll move the data into your Hive warehouse, not make another copy on your small and tired laptop disk. You'd likely want to do the same thing on a real cluster and not waste the storage. In Step (B) of above listing we use the LIMIT keyword because this table is huge. In Step (F), the LIKE keyword instructs Hive to copy the existing FlightInfo2007 table definition when creating the FlightInfo2008 table. In Step (G) you're using the same technique as in Step (B).

In the above Listing, Hive could not (at first) match the first record with the data types you specified in your CREATE TABLE statement. So the system showed NULL values in place of the real data, and the command completed successfully. This behavior illustrates that Hive uses a Schema on Read verification approach as opposed to the Schema on Write verification approach, which you find in RDBMS technologies. This is one reason why Hive is so powerful for big data analytics.

INSERT examples

Another Hive DML command to explore is the INSERT command. To demonstrate this new DML command, we have you create a new table that will hold a subset of the data in the FlightInfo2008 table you created in the previous example. We basically have three INSERT variants.Two of those are specified in below listing.

```
(A) CREATE TABLE IF NOT EXISTS myFlightInfo (
  Year SMALLINT, DontQueryMonth TINYINT, DayofMonth
          TINYINT, DayOfWeek TINYINT,
  DepTime SMALLINT, ArrTime SMALLINT,
  UniqueCarrier STRING, FlightNum STRING,
  AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,
  Origin STRING, Dest STRING, Cancelled SMALLINT,
  CancellationCode STRING)
COMMENT 'Flight InfoTable'
PARTITIONED BY(Month TINYINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
(B) STORED AS RCFILE
TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Mon
          Sep  2 14:24:19 EDT 2013');

(C) INSERT OVERWRITE TABLE myflightinfo
  PARTITION (Month=1)
  SELECT Year, Month, DayofMonth, DayOfWeek, DepTime,
          ArrTime, UniqueCarrier,
         FlightNum, AirTime, ArrDelay, DepDelay, Origin,
          Dest, Cancelled,
         CancellationCode
  FROM FlightInfo2008 WHERE Month=1;

(D) FROM FlightInfo2008
INSERT INTO TABLE myflightinfo
  PARTITION (Month=2)
  SELECT Year, Month, DayofMonth, DayOfWeek, DepTime,
          ArrTime, UniqueCarrier, FlightNum,
  AirTime, ArrDelay, DepDelay, Origin, Dest, Cancelled,
          CancellationCode WHERE Month=2
... (Months 3 through 11 skipped for brevity)
INSERT INTO TABLE myflightinfo
  PARTITION (Month=12)
  SELECT Year, Month, DayofMonth, DayOfWeek, DepTime,
          ArrTime, UniqueCarrier, FlightNum,
  AirTime, ArrDelay, DepDelay, Origin, Dest, Cancelled,
          CancellationCode WHERE Month=12;

(E) hive (flightdata)> SHOW PARTITIONS myflightinfo;
OK
month=1
month=10
month=11
month=12
```

```
(F) $ ls
/home/biadmin/Hive/warehouse/flightdata.db/myflightinfo
month=1   month=11  month=2  month=4  month=6  month=8
month=10  month=12  month=3  month=5  month=7  month=9

(G) $HIVE_HOME/bin/hive --service rcfilecat
  /home/biadmin/Hive/warehouse/flightdata.db/myflightinfo/
          month=12/000000_0
...
2008    12      13      6       655     856     DL
          1638    85      0       -5      PBI     ATL
          0
2008    12      13      6       1251    1446    DL
          1639    89      9       11      IAD     ATL
          0
2008    12      13      6       1110    1413    DL
          1641    104     -5      7       SAT     ATL
          0
```

Listing : Partitioned Version of 2008 Flight Information Table

In the above listing in Step (A), we create this new table and in Step(B), we specify that the file format will be row columnar instead of text. This format is more compact than text and often performs better, depending on our access patterns. (If you're accessing a small subset of columns instead of entire rows, try the RCFILE format.) . In step(C) we use the INSERT OVERWRITE command to insert data via a SELECT statement from the FlightInfo2008 table.

Note that we're partitioning our data using the PARTITION keyword based on the Month field. After we're finished, we'll have 12 table partitions, or actual directories, under the warehouse directory in the file system on our virtual machine, corresponding to the 12 months of the year. As we explain earlier, partitioning can dramatically improve our query performance if we want to query data in the myFlightInfo table for only a certain month. We can see the results of the PARTITION approach with the SHOW PARTITIONS command in Steps (E) and (F). Notice in Step (D) that we're using a variant of the INSERT command to insert data into multiple partitions at one time. We have only shown month 2 and 12 for brevity but months 3 through 11 would have the same syntax.

You can also use this FROM table1 INSERT INTO table2 SELECT ... format to insert into multiple tables at a time. We have you use INSERT instead of OVERWRITE here to show the option of inserting instead of overwriting. Hive allows only appends, not inserts, into tables, so the INSERT keyword simply instructs Hive to append the data to the table. Finally, note in Step (G) that we have to use a special Hive command service (rcfilecat) to view this table in your warehouse, because the RCFILE format is a binary format, unlike the previous TEXTFILE format examples. Third one is the Dynamic Partition Inserts variant. In below Listing, you partition the myFlightInfo table into 12 segments, 1 per month. If you had hundreds of partitions, this task would have become quite difficult, and it would have required scripting to get the job done. Instead, Hive supports a technique for dynamically creating partitions with the INSERT OVERWRITE statement.

Create Table As Select (CTAS) examples

The powerful technique in Hive known as *Create Table As Select,* or *CTAS.* Its constructs allow us to quickly derive Hive tables from other tables as we build powerful schemas for big data analysis. The following Listing shows you how CTAS works.

```
(A) hive> CREATE TABLE myflightinfo2007 AS
    > SELECT Year, Month, DepTime, ArrTime, FlightNum,
          Origin, Dest FROM FlightInfo2007
    > WHERE (Month = 7 AND DayofMonth = 3) AND
          (Origin='JFK' AND Dest='ORD');
(B) hive> SELECT * FROM myFlightInfo2007;
OK
2007    7       700     834     5447    JFK     ORD
2007    7       1633    1812    5469    JFK     ORD
2007    7       1905    2100    5492    JFK     ORD
2007    7       1453    1624    4133    JFK     ORD
2007    7       1810    1956    4392    JFK     ORD
2007    7       643     759     903     JFK     ORD
2007    7       939     1108    907     JFK     ORD
2007    7       1313    1436    915     JFK     ORD
2007    7       1617    1755    917     JFK     ORD
2007    7       2002    2139    919     JFK     ORD
```

```
Time taken: 0.089 seconds, Fetched: 10 row(s)
hive> CREATE TABLE myFlightInfo2008 AS
    > SELECT Year, Month, DepTime, ArrTime, FlightNum,
          Origin, Dest FROM FlightInfo2008
    > WHERE (Month = 7 AND DayofMonth = 3) AND
          (Origin='JFK' AND Dest='ORD');
hive> SELECT * FROM myFlightInfo2008;
OK
2008    7       930     1103    5199    JFK     ORD
2008    7       705     849     5687    JFK     ORD
2008    7       1645    1914    5469    JFK     ORD
2008    7       1345    1514    4392    JFK     ORD
2008    7       1718    1907    1217    JFK     ORD
2008    7       757     929     1323    JFK     ORD
2008    7       928     1057    907     JFK     ORD
2008    7       1358    1532    915     JFK     ORD
2008    7       1646    1846    917     JFK     ORD
2008    7       2129    2341    919     JFK     ORD
Time taken: 0.186 seconds, Fetched: 10 row(s)
```

Listing: An Example of Using CREATE TABLE . . . AS SELECT

In Step A, we build two smaller tables derived from the FlightInfo2007 and FlightInfo2008 by selecting a subset of fields from the larger tables for a particular day (in this case, July 3), where the origin of the flight is New York's JFK airport (JFK) and the destination is Chicago's O'Hare airport (ORD). Then in Step B we simply dump the contents of these small tables so that you can view the data.

Querying and Analyzing Data

Joining tables with Hive

Well, remember that the underlying operating system for Hive is (surprise!) Apache Hadoop: MapReduce is the engine for joining tables, and the Hadoop File System (HDFS) is the underlying storage. Disk and network access is a lot slower than memory access, so minimize HDFS reads and writes as much as possible. Hive table reads and writes via HDFS usually involve very large blocks of data, the more data you can manage altogether in one table, the better the overall performance.

Now we show you a Hive join example using our flight data tables. The above Listing shows you how to create and display a myflightinfo2007 table and a myflightinfo2008 table from the larger FlightInfo2007 and FlightInfo2008 tables. The plan all along was to use the CTAS created myflightinfo2007 and myflightinfo2008 tables to illustrate how you can perform joins in Hive. The plan all along was to use the CTAS created myflightinfo2007 and myflightinfo2008 tables to illustrate how you can perform joins in Hive. Figure 8 shows the result of an inner join with the myflightinfo2007 and myflightinfo2008 tables using the SQuirreL SQL client.

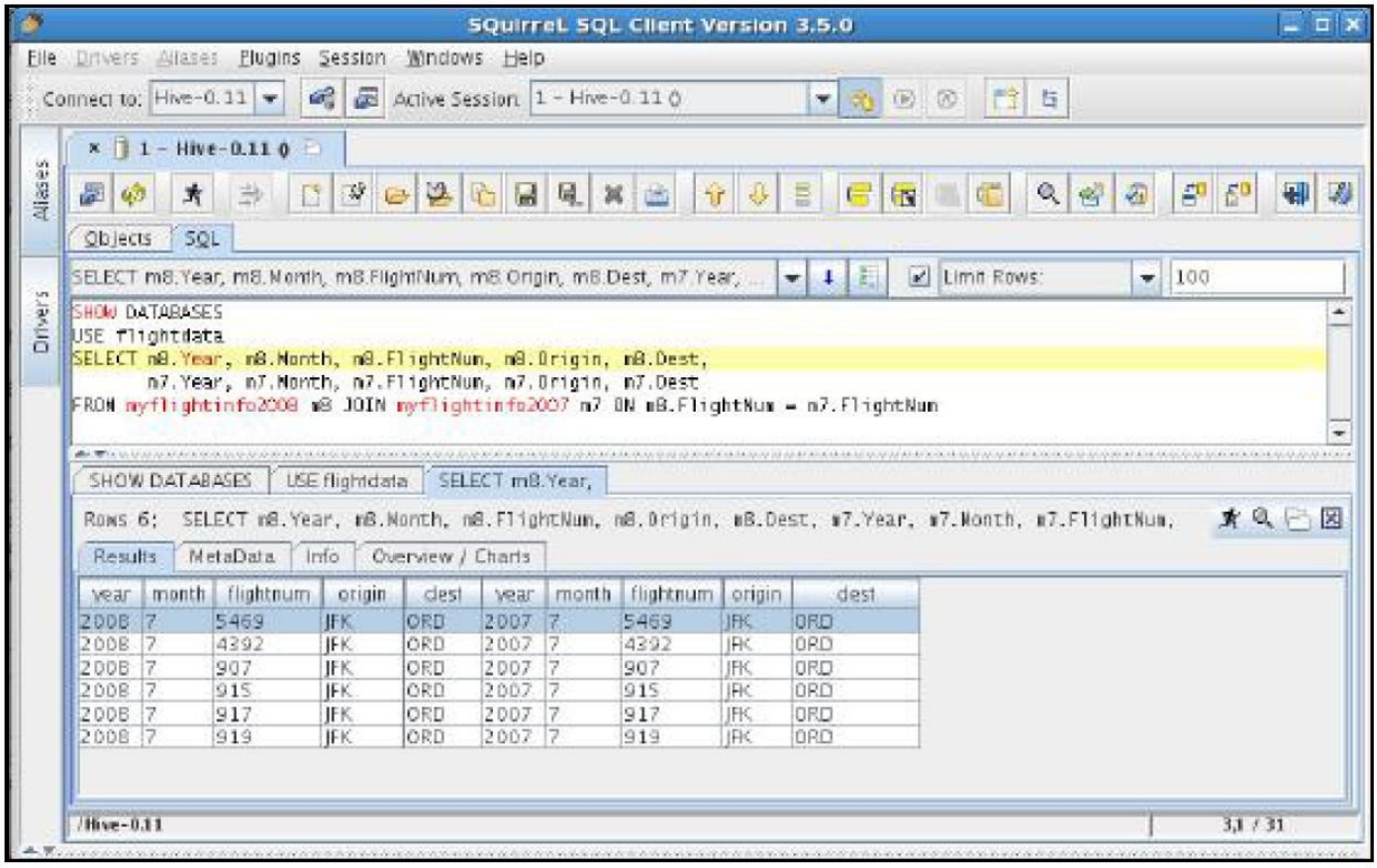


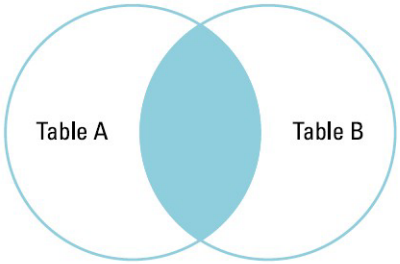
Figure 8: The Hive inner join

Hive supports *equi-joins,* a specific type of join that only uses equality comparisons in the join predicate. Other comparators such as Less Than (<) are not supported. This restriction is only because of limitations on the underlying MapReduce engine. Also, you cannot use OR in the ON clause.

Figure 9 illustrates how an inner join works using a Venn diagram technique. The basic idea here is that an inner join returns the records that match between two tables. So an inner join is a perfect analysis tool to determine which flights are the same from JFK (New York) to ORD (Chicago) in July of 2007 and July of 2008.

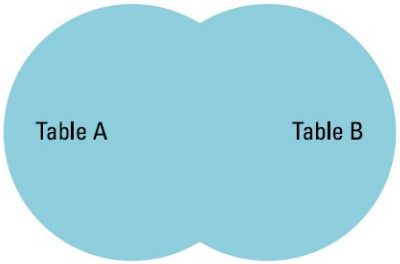
Hive Join Examples

Inner Join



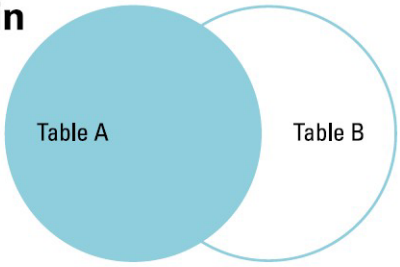

```
SELECT m8.Year, m8.Month, m8.FlightNum, m8.0rigin, m8.Dest,
    m7.Year, m7.Month, m7.FlightNum, m7.0rigin, m7.Dest
 FROM myflightinfo2008 m8JOIN myflightinfo2007 m 7
ON m8.FlightNum = m7.FlightNum;
```

Full Outer Join




```
SELECT m8.FlightNum, m8.0rigin, m8.Dest,
    m7.FlightNum, m7.0rigin, m7.Dest
  FROM myflightinfo2008 m8
 FULL OUTER JOIN myflightinfo2007 m7
ON m8.FlightNum = m7.FlightNum;
```

Left Outer Join



Note: Hive also supports:
- Right Outer Joins,
- Left Semi Joins, and
- Cross Joins (Cartesian Product)

```
SELECT m8.Year, m8.Month, m8.FlightNum, m8.Origin, m8.Dest,
    m7.Year, m7.Month, m7.FlightNum, m7.0rigin, m7.Dest
FROM myflightinfo2008 m8 LEFT OUTER JOIN myflightinfo2007 m7 ON m8.FlightNum = m 7.FlightNum;
```

Figure 9: Hive inner join, full outer join, and left outer join.

Improving your Hive queries with indexes

Creating an index is common practice with relational databases when we want to speed access to a column or set of columns in your database. Without an index, the database system has to read all rows in the table to find the data we have selected. Indexes become even more essential when the tables grow extremely large. Hive supports index creation on tables. In Listing 18, we list the steps necessary to index the FlightInfo2008 table.

```
(A) CREATE INDEX f08_index ON TABLE flightinfo2008
          (Origin) AS 'COMPACT' WITH DEFERRED REBUILD;
(B) ALTER INDEX f08_index ON flightinfo2008 REBUILD;
(C) hive (flightdata)> SHOW INDEXES ON FlightInfo2008;
OK
f08index                flightinfo2008          origin
          flightdata__flightinfo2008_f08index__ compact
Time taken: 0.079 seconds, Fetched: 1 row(s)
(D) hive (flightdata)> DESCRIBE
          flightdata__flightinfo2008_f08index__;
OK
origin                  string                  None
_bucketname             string
_offsets                array<bigint>
Time taken: 0.112 seconds, Fetched: 3 row(s)
(E) hive (flightdata)> SELECT Origin, COUNT(1) FROM
          flightinfo2008 WHERE Origin = 'SYR' GROUP BY
          Origin;
SYR     12032
Time taken: 17.34 seconds, Fetched: 1 row(s)
(F) hive (flightdata)> SELECT Origin, SIZE(`_offsets`)
          FROM flightdata__flightinfo2008_f08index__
          WHERE origin = 'SYR';
SYR     12032
Time taken: 8.347 seconds, Fetched: 1 row(s)
(G) hive (flightdata)> DESCRIBE
          flightdata__flightinfo2008_f08index__;
OK
origin                  string                  None
_bucketname             string
_offsets                array<bigint>
Time taken: 0.12 seconds, Fetched: 3 row(s)
```

Listing : Creating an Index on the FlightInfo2008 Table

Step (A) creates the index using the 'COMPACT' index handler on the Origin column. Hive also offers a bitmap index handler as of the 0.8 release, which is intended for creating indexes on columns with a few unique values. The keywords WITH DEFERRED REBUILD instructs Hive to first create an empty index; Step (B) is where we actually build the index with the ALTER INDEX ... REBUILD command. Deferred index builds can be very useful in workflows where one process creates the tables and indexes, another loads the data and builds the indexes and a final process performs data analysis. Hive doesn't provide automatic index maintenance, so you need to rebuild the index if you overwrite or append data to the table. Also, Hive indexes support table partitions, so a rebuild can be limited to a partition. Step (C) illustrates how we can list or show the indexes created against a particular table. Step (D) illustrates an important point regarding

Hive indexes:

Hive indexes are implemented as tables. This is why we need to first create the index table and then build it to populate the table. Therefore, we can use indexes in at least two ways:

✓ Count on the system to automatically use indexes that you create.

✓ Rewrite some queries to leverage the new index table

In Step (E) we write a query that seeks to determine how many flights left the Syracuse airport during 2008. To get this information, we leverage the COUNT aggregate function. In Step (F), you leverage the new index table and use the SIZE function instead.

Windowing in HiveQL

The concept of *windowing,* introduced in the SQL:2003 standard, allows the SQL programmer to create a frame from the data against which aggregate and other window functions can operate. HiveQL now supports windowing per the SQL standard.One question we had when we first discovered this data set was, "What exactly is the average flight delay per day?" So we created a query in below Listing that produces the average departure delay per day in 2008.

```
(A) hive (flightdata)> CREATE VIEW avgdepdelay AS
                    > SELECT DayOfWeek, AVG(DepDelay) FROM
             FlightInfo2008 GROUP BY DayOfWeek;
OK
Time taken: 0.121 seconds
(B) hive (flightdata)> SELECT * FROM avgdepdelay;
...
OK
1       10.269990244459473
2       8.97689712068735
3       8.289761053658728
4       9.772897177836702
5       12.158036387869656
6       8.645680904903614
7       11.568973392595312
Time taken: 18.6 seconds, Fetched: 7 row(s)
```

Listing : Finding the Average Departure Delay per Day in 2008

As sown in step(A) of above Listing Hive's Data Definition Language (DDL) also includes the CREATE VIEW statement, which can be quite useful. In Hive, views allow a query to be saved but data is not stored as with the Create Table as Select (CTAS) statement.

Suppose if you want to know "What is the first flight between Airport X and Y?" Suppose that in addition to this information, you want to know about subsequent flights, just in case you're not a "morning person." Write the query as below.

```
(A) hive (flightdata)> SELECT f08.Month, f08.DayOfMonth,
          cr.description, f08.Origin, f08.Dest,
          f08.FlightNum, f08.DepTime, MIN(f08.DepTime)
OVER (PARTITION BY f08.DayOfMonth ORDER BY f08.DepTime)
FROM flightinfo2008 f08 JOIN Carriers cr ON
          f08.UniqueCarrier = cr.code
WHERE f08.Origin = 'JFK' AND f08.Dest = 'ORD' AND
          f08.Month = 1 AND f08.DepTime != 0;
...
OK
1     1  JetBlue Airways          JFK ORD 903   641 641
1     1  American Airlines Inc.   JFK ORD 1323  833  641
1     1  JetBlue Airways          JFK ORD 907   929  641
1     1  Comair Inc.              JFK ORD 5083  945  641
1     1  Comair Inc.              JFK ORD 5634  1215 641
1     1  JetBlue Airways          JFK ORD 915   1352 641
1     1  American Airlines Inc.   JFK ORD 1323  833  641
1     1  JetBlue Airways          JFK ORD 907   929  641
1     1  Comair Inc.              JFK ORD 5083  945  641
1     1  Comair Inc.              JFK ORD 5634  1215 641
1     1  JetBlue Airways          JFK ORD 915   1352 641
1     1  American Airlines Inc.   JFK ORD 1815  1610 641
1     1  JetBlue Airways          JFK ORD 917   1735 641
1     1  Comair Inc.              JFK ORD 5469  1749 641
1     1  Comair Inc.              JFK ORD 5492  2000 641
1     1  JetBlue Airways          JFK ORD 919   2102 641
1     31 JetBlue Airways          JFK ORD 919   48   48
```

Listing : Using Aggregate Window Functions on the Flight Data

In Step (A), we've replaced the GROUP BY clause with the OVER clause where we specify the PARTITION or window over which we want the MIN aggregate function to operate. We've also included the ORDER BY clause so that we can see those subsequent flights after the first one.