

Hive Workshop

Contents

Hive Workshop.....	1
Lesson 0 – Hello World	3
Lesson 1 – Hive Queries	4
Simple selects - selecting columns.....	4
Simple selects – selecting rows.....	4
Creating new columns	5
Hive Functions.....	6
Simple functions.....	6
Aggregations	7
date functions	8
Lesson 2 Advanced Queries	10
More ways of creating new columns	10
Table Joins.....	10
An example join statement.....	12
Sampling in Hive.....	13
Sub Queries (Sampling the Smart meter dataset)	13
Lesson 3 - Loading data into HDFS.....	15
Hive Storage and HDFS	15
Managed Tables	15
External Tables	15
Creating Tables.....	15
Loading Data into tables	17
HDFS Commands.....	19
Commonly used commands.....	19
HDFS Commands for moving data	20
Moving data into HDFS	20
Moving data out of HDFS	20
Partitioned tables.....	21
Lesson 4 – Accessing Hive from the command line.....	23
Using Putty.....	23

Using Parameters with Hive queries.....	27
Lesson 5 – Accessing Hive via ODBC	29

Lesson 0 – Hello World

[These examples are included in the 'hello world.sql' file]

This is just a short introduction to the Toad for Hadoop environment

Lesson 1 – Hive Queries

This lesson will cover the following topics:

- Simple selects - selecting columns
- Simple selects – selecting rows
- Creating new columns
- Hive Functions

In SQL, of which HQL is a dialect, querying data is performed by a SELECT statement. A select statement has 6 key component;

```
SELECT colnames
FROM tablename
GROUP BY colnames
WHERE conditions
HAVING conditions
ORDER by colnames
```

In practice very few queries will have all of these clauses in them simplifying many queries. On the other hand, conditions in the WHERE clause can be very complex and if you need to JOIN two or more tables together then more clause (JOIN and ON) are needed.

All of the clause names above have been written in uppercase for clarity. HQL is not case sensitive. Neither do you need to write each clause on a new line, but it is often clearer to do so for all but the simplest of queries.

In this lesson we will start with the very simple and work our way up to the more complex.

Simple selects - selecting columns

[These examples are included in the '01 – simple queries.sql' file]

The simplest query is effectively one which returns the contents of the whole table

```
SELECT *
FROM geog_all;
```

It is better practice and generally more efficient to explicitly list the column names that you want returned.

```
SELECT anonid, fueltypes, acorn_type
FROM geog_all;
```

Simple selects – selecting rows

In addition to limiting the columns returned by a query, you can also limit the rows returned. The simplest case is to say how many rows are wanted using the Limit clause.

```
SELECT anonid, fueltypes, acorn_type
FROM geog_all
LIMIT 10;
```

This is useful if you just want to get a feel for what the data looks like.

Usually you will want to restrict the rows returned based on some criteria. i.e. certain values or ranges within one or more columns.

```
SELECT anonid, fueltypes, acorn_type
FROM geog_all
WHERE fueltypes = "ElecOnly";
```

The Expression in the where clause can be more complex and involve more than one column.

```
SELECT anonid, fueltypes, acorn_type
FROM geog_all
WHERE fueltypes = "ElecOnly" AND acorn_type > 42;

SELECT anonid, fueltypes, acorn_type
FROM geog_all
WHERE fueltypes = "ElecOnly" AND acorn_type > 42 AND nuts1 <> "--";
```

Notice that the columns used in the conditions of the Where clause don't have to appear in the Select clause.

Other operators can also be used in the where clause. For complex expressions, brackets can be used to enforce precedence.

```
SELECT anonid,
       fueltypes,
       acorn_type,
       nuts1,
       ldz
FROM geog_all
WHERE fueltypes = "ElecOnly"
      AND acorn_type BETWEEN 42 AND 47
      AND (nuts1 NOT IN ("UKM", "UKI") OR ldz = "--");
```

Creating new columns

It is possible to create new columns in the output of the query. These columns can be from combinations from the other columns using operators and/or builtin Hive functions.

```
SELECT anonid,
       eprofileclass,
       acorn_type,
       (eprofileclass * acorn_type) AS multiply,
       (eprofileclass + acorn_type) AS added
FROM edrp_geography_data b;
```

A full list of the operators and functions available within Hive can be found in the documentation, <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF>

When you create a new column it is usual to provide an 'alias' for the column. This is essentially the name you wish to give to the new column. The alias is given immediately after the expression to which it refers. Optionally you can add the AS keyword for clarity.

If you do not provide an alias for your new columns, Hive will generate a name for you.

Although the term alias may seem a bit odd for a new column which has no natural name, alias' can also be used with any existing column to provide a more meaningful name in the output.

Tables can also be given an alias, this is particularly common in join queries involving multiple tables where there is a need to distinguish between columns with the same name in different tables.

In addition to using operators to create new columns there are also many Hive built-in functions that can be used.

Hive Functions

[These examples are included in the '02 - functions.sql' file]

Simple functions

Concat can be used to add strings together

```
SELECT anonid,
       acorn_category,
       acorn_group,
       acorn_type,
       concat (acorn_category,
              ",",
              acorn_group,
              ",",
              acorn_type)
       AS acorn_code
FROM geog_all;
```

substr can be used to extract a part of a string

```
SELECT anon_id,
       advancedatetime,
       substr (advancedatetime, 1, 2) AS day,
       substr (advancedatetime, 3, 3) AS month,
       substr (advancedatetime, 6, 2) AS year
FROM elec_c;
```

examples of **length**, **instr** and **reverse**

```
SELECT anonid,
       acorn_code,
```

```

length (acorn_code),
instr (acorn_code, ',') AS a_catpos,
instr (reverse (acorn_code), ",") AS reverse_a_typepos
FROM geog_all;

```

Where needed functions can be nested within each other

cast and type conversions

```

SELECT anonid,
       substr (acorn_code, 7, 2) AS ac_type_string,
       cast (substr (acorn_code, 7, 2) AS INT) AS ac_type_int,
       substr (acorn_code, 7, 2) +1 AS ac_type_not_sure
FROM geog_all;

```

Aggregations

[These examples are included in the '03 - aggregations.sql' file]

Aggregate functions are used perform some kind of mathematical or statistical calculation across a group of rows. The rows in each group are determined by the different values in a specified column or columns. A list of all of the available functions are available in the apache documentation.

```

SELECT anon_id,
       count (electkwh) AS total_row_count,
       sum (electkwh) AS total_period_usage,
       min (electkwh) AS min_period_usage,
       avg (electkwh) AS avg_period_usage,
       max (electkwh) AS max_period_usage
FROM elec_c
GROUP BY anon_id;

```

In the above example, the aggregation were performed over the single column anon_id. It is possible to aggregate over multiple columns by specifying them in both the select and the group by clause. The grouping will take place based on the order of the columns listed in the group by clause. What is not allowed is specifying a non-aggregated column in the select clause which is not mentioned in the group by clause.

```

SELECT anon_id,
       substr (advancedatetetime, 6, 2) AS reading_year,
       count (electkwh) AS total_row_count,
       sum (electkwh) AS total_period_usage,
       min (electkwh) AS min_period_usage,
       avg (electkwh) AS avg_period_usage,
       max (electkwh) AS max_period_usage
FROM elec_c
GROUP BY anon_id, substr (advancedatetetime, 6, 2);

```

Unfortunately, the group by clause will not accept alias'.

```

SELECT anon_id,

```

```

        substr (advancedatettime, 6, 2) AS reading_year,
        count (electkwh) AS total_row_count,
        sum (electkwh) AS total_period_usage,
        min (electkwh) AS min_period_usage,
        avg (electkwh) AS avg_period_usage,
        max (electkwh) AS max_period_usage
    FROM elec_c
GROUP BY anon_id, substr (advancedatettime, 6, 2)
ORDER BY anon_id, reading_year;

```

But the Order by clause does.

The Distinct keyword provides a set of unique combination of column values within a table without any kind of aggregation.

```

SELECT DISTINCT eprofileclass, fueltypes
FROM geog_all;

```

date functions

[These examples are included in the '04 - date functions.sql' file]

In the elec_c and gas_c tables, the advancedatettime column, although it contains a timestamp type information, it is defined as a string type. For much of the time this can be quite convenient, however there will be times when we really do need to be able to treat the column as a Timestamp. Perhaps the most obvious example is when you need to sort rows based on the advancedatettime column.

Hive provides a variety of date related functions to allow you to convert strings into Timestamp and to additionally extract parts of the Timestamp.

unix_timestamp returns the current data and time – as an integer!

from_unixtime takes an integer and converts in into a recognisable Timestamp string

```

SELECT unix_timestamp () AS currenttime
FROM sample_07
LIMIT 1;

SELECT from_unixtime (unix_timestamp ()) AS currenttime
FROM sample_07
LIMIT 1;

```

There are various date part functions which will extract the relevant parts from a Timestamp string

```

SELECT anon_id,
        from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMMy'))
        AS proper_date,
        year (from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMMy')))
        AS full_year,

```



```
month (from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMyy'))
      AS full_month,
day (from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMyy'))
    AS full_day,
last_day (from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMyy')))
  AS last_day_of_month,
date_add ( (from_unixtime (UNIX_TIMESTAMP (reading_date, 'ddMMyy'))),
          10)
  AS added_days
FROM elec_days_c
ORDER BY proper_date;
```

Lesson 2 Advanced Queries

This lesson will cover the following topics:

- More ways of creating new columns
- Table Joins
- Sampling in Hive
- Sub Queries

More ways of creating new columns

New columns can also be created by using the if and case clauses

[These examples are included in the '05 - if and Case.sql' file]

if

```
SELECT anonid,
       elec_tout,
       gas_tout,
       if (elec_tout = 1, elec_tout, gas_tout) AS tout
FROM   geog_all;
```

The expression in the first parameter is evaluated, if True, the value in the second parameter is used and if it evaluates to false, the value of the third parameter is used. In this example other columns from the table have been used, but you could use literal values as we do for the case statement example next.

case

```
SELECT anon_id,
       reading_month,
       monthlykwh,
       CASE
         WHEN monthlykwh BETWEEN 0 AND 400 THEN 'Low'
         WHEN monthlykwh BETWEEN 401 AND 900 THEN 'Medium'
         WHEN monthlykwh BETWEEN 901 AND 1400 THEN 'High'
         ELSE 'Far too high!'
       END
       AS usage_type
FROM   elec_months_c;
```

The case statement is like a multi expression if statement.

Table Joins

In any relational database system, the ability to join tables together is a key querying requirement.

Joins are used to combine the rows from two (or more) tables together to form a single table. A join between tables will only be possible if they have at least one column in common. The column doesn't have to have the same name in each table, and quite often they won't, but they do have to have a common usage. For example, in the geography table there is the anonid column and in the

elec or gas tables there is the anon_id column in both cases they represent an anonymised household.

There are several different types of join possible.

Join Type	What it does
Inner Join	Matched rows in both tables are returned
Left outer join	All row in the left hand table are returned along with the matches from the right hand table or NULLs if there is no match
Right outer join	All row in the right hand table are returned along with the matches from the left hand table or NULLs if there is no match
Full outer join	All rows from both tables are returned, with NULLs where there are no matches
Cross join	

By far the inner join is the most commonly used. The outer joins can be useful for exploring or discovering missing data. The cross join is rarely used and could create extremely large tables.

Examples

using the two small tables below

Animals

Id_A	Name
1	Elephant
2	Monkey
3	Cat
4	Dog
8	Goat
10	Pig
11	Mouse

Animal_Eats

Id_E	Eats
1	Hay
3	Fish
4	Meat
6	Goldfish food
7	Lettuce
8	Flowers
10	Anything

an **Inner Join** results in

Id_A	Name	Id_E	Eats
1	Elephant	1	Hay
3	Cat	3	Fish
4	Dog	4	Meat
8	Goat	8	Flowers
10	Pig	10	Anything

a **Left Outer Join** results in

Id_A	Name	Id_E	Eats
------	------	------	------

1	Elephant	1	Hay
2	Monkey	NULL	NULL
3	Cat	3	Fish
4	Dog	4	Meat
8	Goat	8	Flowers
10	Pig	10	Anything
11	Mouse	NULL	NULL

a **Right Outer Join** results in

Id_A	Name	Id_E	Eats
1	Elephant	1	Hay
3	Cat	3	Fish
4	Dog	4	Meat
NULL	NULL	6	Goldfish food
NULL	NULL	7	Lettuce
8	Goat	8	Flowers
10	Pig	10	Anything

a **Full Outer Join** results in

Id_A	Name	Id_E	Eats
1	Elephant	1	Hay
2	Monkey	NULL	NULL
3	Cat	3	Fish
4	Dog	4	Meat
NULL	NULL	6	Goldfish food
NULL	NULL	7	Lettuce
8	Goat	8	Flowers
10	Pig	10	Anything
11	Mouse	NULL	NULL

An example join statement

```
SELECT e.anon_id, g.anon_id
FROM distinct_elec_c AS e
JOIN distinct_gas_c AS g
ON e.anon_id = g.anon_id;
```

[Additional examples of joins are in the '06 - table joins.sql' file]

Sampling in Hive

Hive provides a mechanism by which table rows can be sampled.

```
SELECT *
FROM source TABLESAMPLE(BUCKET 3 OUT OF 32 ON rand()) s;
```

In the above example the rows of a table called source are randomly distributed across 32 'buckets'. Only the rows in bucket 3 are returned. The number of buckets is a user option. The higher the value, the less rows there will be in each. As the distribution is random there is no guarantee that each bucket will contain the same number of rows.

In the code above because of the rand() clause, if it were executed again, bucket 3 would contain a different set of rows. This can be controlled by providing a 'seed' to the rand function, such as rand(1). Now if the code is re-run bucket 3 (and all of the other buckets) will always contain the same set of rows as on the previous run. This will help you to reproduce your results.

[Examples using the geog_all tables are in the '07 - sampling geog.sql' file]

```
SELECT anonid
FROM geog_all TABLESAMPLE(BUCKET 3 OUT OF 150 ON rand(1)) ;
```

Sub Queries (Sampling the Smart meter dataset)

The problem with this sampling approach is that it does not take in to account the fact that the electricity usage for a given consumer on a given day is spread over 48 rows of the table. If a sample were taken using the above approach directly it is highly unlikely that there would be any complete days of usage for any of the consumers making any further useful analysis all but impossible.

However, by using this method to sample the geography table, which contains a single row only for each household instead of sampling the meter readings table directly it is possible to select a sample of unique households based on the anonid variable values. The anonid variable in the geography table represents the households and can be used to join the geography file data with the meter readings data.

In the query below we are selecting from the elec_all table all of the row associated with the anonids in bucket 3.

```
SELECT *
FROM elec_all e
WHERE anon_id IN
      (SELECT anonid
       FROM geog_all
        TABLESAMPLE (BUCKET 3 OUT OF 150 ON rand (1)) s);
```

[Examples using the geog_all tables are in the '08 - create c tables.sql' file]

Lesson 3 - Loading data into HDFS

Topics in this Lesson will include:

- Hive Storage and HDFS
- Creating Tables
- Loading Data into tables
- HDFS Commands
- HDFS Commands for moving data
- Partitioned tables

Hive Storage and HDFS

[Examples of table creation are in the file '09 - create table examples.sql']

Hive uses HDFS to store the data associated with Hive tables. It can do this in either of two ways. Which one it uses for any given table is decided by the user when the table is defined.

Managed Tables

In a managed table both the table data and the table schema are controlled by Hive. The data will be located in a folder named after the table within the Hive data warehouse, which is essentially just a file location in HDFS. In the case of the Sandbox this is at **/apps/hive/warehouse**. The location is user configurable when Hive is installed.

By controlled we mean that if you drop (delete) a managed table, then Hive will delete both the Schema (the description of the table) and the data files associated with the table.

External Tables

An external table is one where only the table schema is controlled by Hive. In most cases the user will set up the folder location within HDFS and copy the data file(s) there. This location is included as part of the table definition statement. When an external table is deleted, Hive will only delete the schema associated with the table. The data files are not affected.

Creating Tables

In Hive there is quite a loose association between a table definition and the actual data which will form the rows and columns of the table.

When you explicitly create a table definition and you specify a file location for the data, for a managed table, Hive will create a folder as based on the table name, for an External table it does not check that the folder exists where the user has specified. Hive does but doesn't check whether or not there is a data file within the folder specified.

As a consequence, it also does not check that the layout of the data in a datafile matches the description of the layout in the schema you define.

The not quite complete syntax for creating tables is here.

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]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]
  [SKEWED BY (col_name, col_name, ...)
    ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
    [STORED AS DIRECTORIES]
  ]
  [
  [ROW FORMAT row_format]
  [STORED AS file_format]
  | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]
  ]
  [LOCATION hdfs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]
  [AS select_statement]; -- internal table only
```

The bare minimum for defining a table is

```
CREATE TABLE table_name
  [(col_name data_type, ...)]
```

an example might be:

```
CREATE TABLE IF NOT EXISTS smartmeterdata
(
  anon_id          INT,
  advancedatetime STRING,
  hh               INT,
  gaskwh           DOUBLE
)
```

The full documentation for table creation is available here.

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-Create/Drop/TruncateTable>

Although it is possible to use the minimum, in doing so you are allowing Hive to make assumptions about your data which may not be the case. Here is another more realistic example.

```
CREATE EXTERNAL TABLE IF NOT EXISTS gas_all
(
```



```

anon_id          INT,
advancedatetime STRING,
hh              INT,
gaskwh          DOUBLE
)
ROW FORMAT DELIMITED
      FIELDS TERMINATED BY ','
      ESCAPED BY '\\'
      LINES TERMINATED BY '\n'
STORED AS
      INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
      OUTPUTFORMAT
'org.apache.hadoop.hive ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION 'hdfs://sandbox.hortonworks.com:8020/user/ukdstain1/gas'
TBLPROPERTIES ('skip.header.line.count' = '1')

```

The additional clauses have the following effects;

EXTERNAL – This is an External table; the user will be responsible for managing the deletion of the data.

ROW FORMAT ... - fields in the data will be separated by the ‘,’ character as they would be in a csv file. Note that this is not the default value, which is ‘\001’. Should the ‘,’ character be part of the data, then it will be escaped with the ‘\’ character. Each row of the datafile will be terminated by the newline character.

STORED AS ... these are in fact the defaults and they equate to the Textfile format. Other formats are available (see documentation for full list)

LOCATION – This is where the data associated with the table is to be stored. It can always be specified, and you can use any hdfs location you have access to. It is more usual to use it for External Tables where you may be using some other system to place the data files into folders.

TBLPROPERTIES – These are generic items which take the format of ‘key’=‘value’ pairs. You can make up your own as a form of documentation. There are however some key values that Hive will make use of. In the example above, Hive is being told that the first line in the file contains header information (e.g. row names from a csv file) and that it is to be ignored as not part of the data.

Loading Data into tables

The easiest way to create a table and load data into it, is to have Hive do it for you.

```

CREATE TABLE smartmeterdata_2728 AS
SELECT *
FROM gas_c
WHERE anon_id = 2728;

```

The above will create a managed table with the same structure as the gas_c table and load into it the results from the select query.

If you have a table already created such as;

```
CREATE TABLE IF NOT EXISTS smartmeterdata
(
  anon_id          INT,
  advancedatetime  STRING,
  hh               INT,
  gaskwh           DOUBLE
)
```

you could either use this directly to load data into or you could treat it as a template to create a more specifically named table

```
CREATE TABLE smartmeterdata_15224
LIKE smartmeterdata;
```

and then load data into it from a select query

```
INSERT OVERWRITE TABLE smartmeterdata_15224
SELECT *
  FROM gas_c
 WHERE anon_id = 15224;
```

Data can be loaded into tables either from a file in HDFS or from a file in the local (Linux VM) filesystem. The format of the command in each case is similar;

```
-- from hdfs data is moved
LOAD DATA INPATH '/user/ukdstrain1/smartmeterdata_hdfs' OVERWRITE INTO
TABLE smartmeterdata_from_hdfs;

-- from local data is copied
-- no OVERWRITE means append
LOAD DATA LOCAL INPATH '/data/smartmeterdata_hdfs' INTO TABLE
smartmeterdata_from_hdfs;

-- Loading to HDFS and local files
INSERT OVERWRITE directory "/user/ukdstrain1/smartmeterdata_hdfs"
select * from smartmeterdata_15224;

INSERT OVERWRITE local directory "/data/smartmeterdata_hdfs"
select * from smartmeterdata_15224;
```

HDFS Commands

The general format of an HDFS command is:

```
hdfs dfs -command name  command_option [command_option]
```

In earlier versions of Hadoop the command was;

```
hadoop fs ...
```

Although this is now depreciated it does still work and more importantly much of the official documentation still uses this format.

A full list of the available hdfs dfs commands can be found on the Apache site at:

<http://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/FileSystemShell.html>

The commands are in general very similar to equivalent Linux commands, although with far fewer command options. There are some commands specifically for moving data to/from hdfs, we will look at those separately. For the others, here are a few of the more common ones.

Commonly used commands

Command	Description
chgrp	change the group associated with the file/directory
chmod	change the access permissions on the file/directory
chown	change the owner of the file/directory
ls	list the contents of a directory
cat	displays the contents of a file
tail	displays the last 1kb of the file
cp	copy files within hdfs
mv	moves files within hdfs
mkdir	create a directory
rmdir	remove a directory
rm	delete a file

help	general help
usage	specific help for a particular command

HDFS Commands for moving data

Data can be moved in and out of HDFS storage using a variety of 'put' and 'get' type commands. Examples of these are shown below.

Moving data into HDFS

Command	Example
put	<pre>hdfs dfs -put localfile /user/hadoop/hadoopfile</pre> <pre>hdfs dfs -put localfile1 localfile2 /user/hadoop/hadoopdir</pre>
copyFromLocal	<pre>hdfs dfs -copyFromLocal <localsrc> <dst></pre>
moveFromLocal	<pre>hdfs dfs -moveFromLocal <localsrc> <dst></pre>
appendToFile	<pre>hdfs dfs -appendToFile localfile /user/hadoop/hadoopfile</pre>

The 'put' command is the most flexible, not all of the possible options are shown above. 'CopyFromLocal' is more descriptive but 'MoveFromLocal' as the name suggests may be more practical if space is at a premium (like in a sandbox).

Local refers to the Linux filesystem in these commands.

Moving data out of HDFS

Command	Example
get	<pre>hdfs dfs -get /user/hadoop/file localfile</pre> <pre>hdfs dfs -get hdfs://nn.example.com/user/hadoop/file localfile</pre>
copyToLocal	<pre>hdfs dfs -copyFromLocal <localsrc> <dst></pre>
moveToLocal	<pre>hdfs dfs -moveFromLocal <localsrc> <dst></pre>
getmerge	<pre>hdfs dfs -getmerge [-nl] <src> <localdst></pre>

These commands mirror the 'put' commands above.

The getmerge command allows a directory of files to be combined into a single output file. This can be very convenient when the 'source' is the outputs from a map-reduce job.

Other software such as the Ambari Web interface or the Toad for Hadoop application offer alternative methods for moving datasets from the client environment directly into HDFS or even into a Hive table. This will be demonstrated later.

Partitioned tables

[An example of creating a partitioned table is in the file 'partitioned table.sql']

Hive allows table to be partitioned by one or more columns. What this means is that a partitioned table instead of being stored as a set of files in a folder named after the table name as shown below,

▼ elec_dates [2]	175.73 MB
000000_0	87.89 MB
000001_0	87.83 MB

under the table name folder there will be a set of folders representing the first partitioned column of the table, if there is a second partitioned column then this folder will contain the folders for the second level partition. In the example below the are two portioned columns; month and year.

▼ elec_dates [12]
▼ month=1 [2]
> year=2009 [12]
> year=2010 [12]
▼ month=10 [2]
> year=2008 [12]
> year=2009 [12]

There main advantage to using partitioned tables is that of query efficiency. If a query uses the partitioned columns in the where clause, Hive can restrict itself to reading only the files from the appropriate partition. This can amount to a considerable efficiency for large tables.

The elec_dates_partitioned table is based on the elec_dates table. Part of the elec_dates table definition is given below.

```
CREATE TABLE elec_dates
(
  anon_id          INT,
  advancedatetime  STRING,
  hh               INT,
  eleckwh         DOUBLE,
  day             INT,
  month           INT,
  year            INT,
  fulldate        STRING
)
```

The partitioned version was defined as follows.

```

CREATE EXTERNAL TABLE elec_dates_partitioned
(
  anon_id          INT,
  advancedatetime STRING,
  hh               INT,
  electkwh         DOUBLE,
  day              INT,
  fulldate         STRING
)
PARTITIONED BY
  (month INT,
   year  INT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION
'hdfs://sandbox.hortonworks.com:8020/user/ukdstrain1/elec_dates_partitioned'
';

```

Notice that the partitioned columns (month and year) do not appear in the columns list in the body of the table definition. Similarly, if you look at the contents of the files that make up the `elec_dates_partitioned` table in HDFS you will find that they are not there.

```

239,16OCT08:17:00:00,34,0.8,16,2008-10-16 00:00:00
239,07OCT08:07:30:00,15,0.12,7,2008-10-07 00:00:00
239,19OCT08:17:00:00,34,0.71,19,2008-10-19 00:00:00
239,09OCT08:17:30:00,35,1.64,9,2008-10-09 00:00:00
239,14OCT08:04:30:00,9,0.15,14,2008-10-14 00:00:00

```

The partitioned column values are contained in the names of the folders.

Lesson 4 – Accessing Hive from the command line

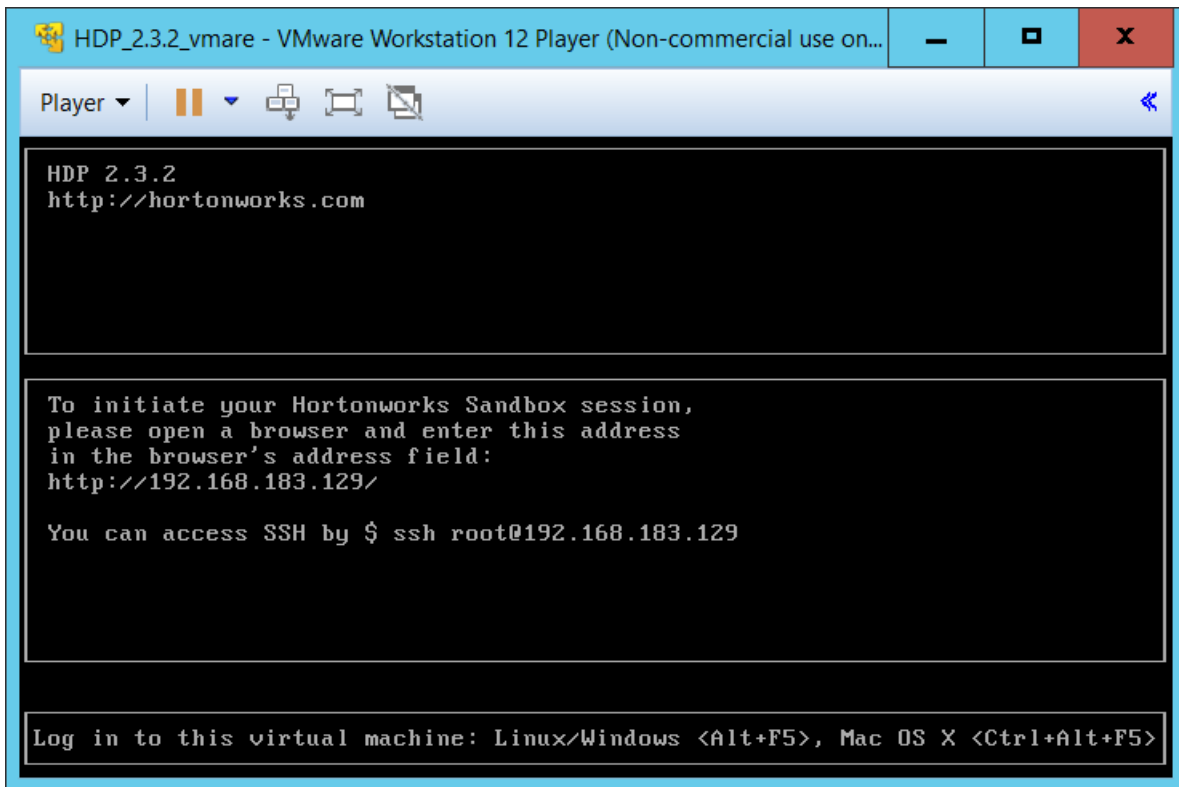
Topics in this lesson will include:

- Using Putty
- running beeline shell
- running queries directly using beeline
- running queries from a file
- creating and running parameterised queries

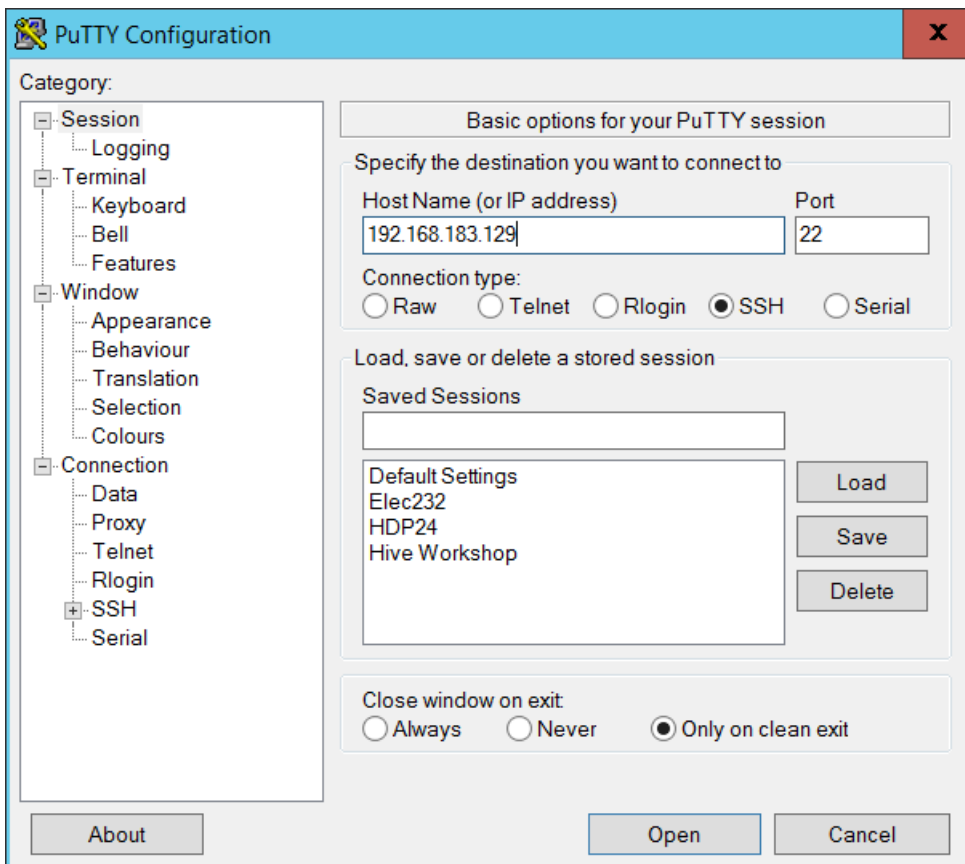
Although accessing Hive from the command line is unlikely to be preferred over web interfaces or client programs offering a GUI like IDE environment, there are situations where it can be beneficial.

Using Putty

The Putty application is on the desktop of the Windows laptop. It provides a simple mechanism for accessing the Sandbox environment. When the program is started, you simply have to provide the IP address of the Sandbox. This is available to you from the VM Player window.

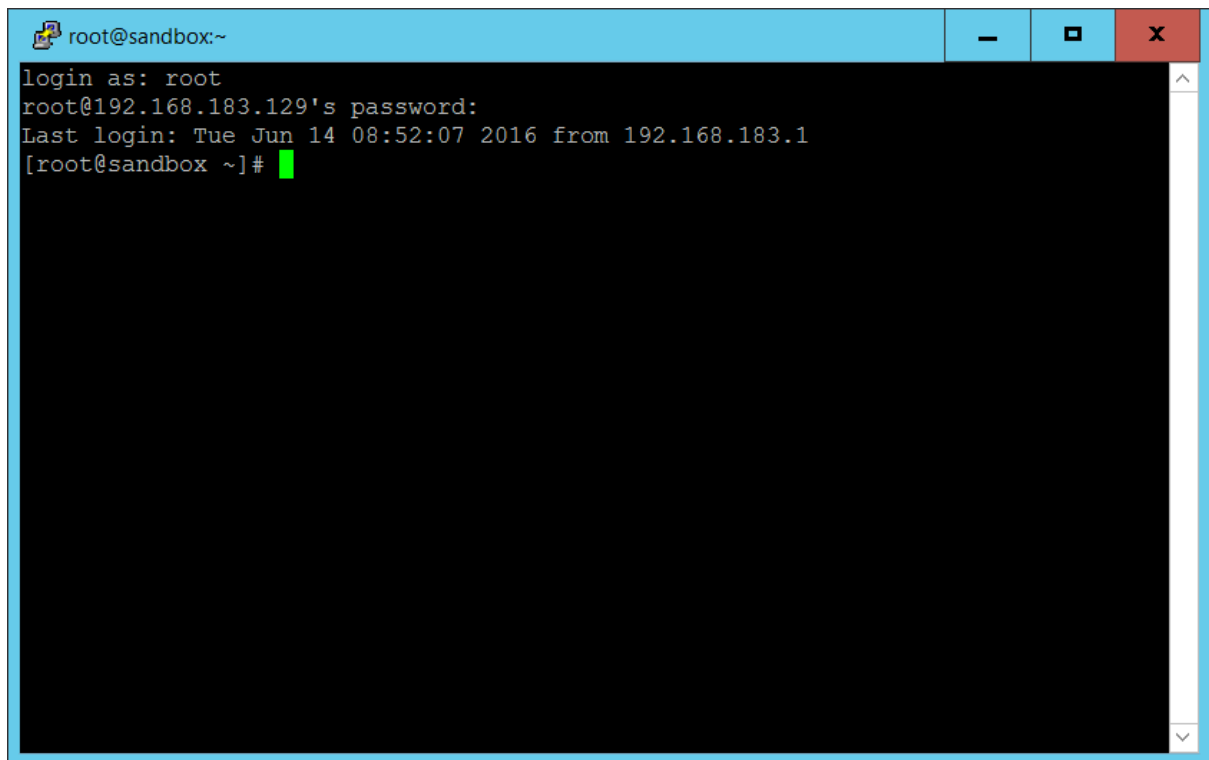


The VM Player screen



The PuTTY opening Window

After clicking Open the user is presented with a terminal window into which they can logon using the credentials provided by Hortonworks for the Sandbox.

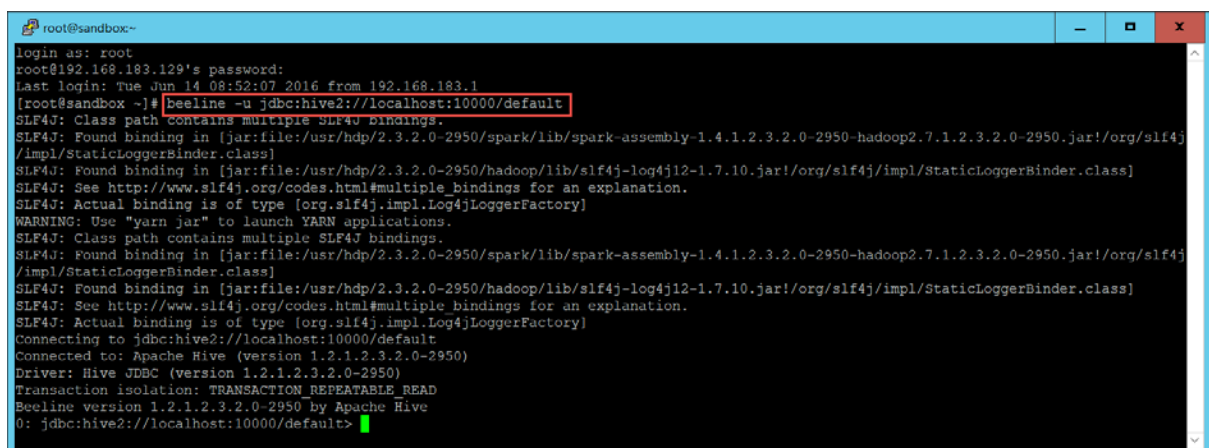


```
root@sandbox:~
login as: root
root@192.168.183.129's password:
Last login: Tue Jun 14 08:52:07 2016 from 192.168.183.1
[root@sandbox ~]#
```

Once you have logged in to the Linux system using the terminal session you can enter the Hive system.

The hive command, although still available is depreciated. The preferred way to access the hive command shell is to use the beeline command;

```
beeline -u jdbc:hive2://localhost:10000/default
```



```
root@sandbox:~
login as: root
root@192.168.183.129's password:
Last login: Tue Jun 14 08:52:07 2016 from 192.168.183.1
[root@sandbox ~]# beeline -u jdbc:hive2://localhost:10000/default
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/spark/lib/spark-assembly-1.4.1.2.3.2.0-2950-hadoop2.7.1.2.3.2.0-2950.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/hadoop/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
WARNING: Use "yarn jar" to launch YARN applications.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/spark/lib/spark-assembly-1.4.1.2.3.2.0-2950-hadoop2.7.1.2.3.2.0-2950.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/hadoop/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
Connecting to jdbc:hive2://localhost:10000/default
Connected to: Apache Hive (version 1.2.1.2.3.2.0-2950)
Driver: Hive JDBC (version 1.2.1.2.3.2.0-2950)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.1.2.3.2.0-2950 by Apache Hive
0: jdbc:hive2://localhost:10000/default>
```

localhost refers to the locally installed Hadoop/Hive system. If you were wanting to access a remote system, then you would replace this with the appropriate IP address. The 10000 is the port on which Hive listens for connections. This is the default port number on the Sandbox. Again, if you are trying to access a remote system you will need to check if this needs changing.

default refers to the database to use. If you need to use another database, you could change this to the correct database name. The default is the default database, so if this is what you are using you could miss it out which will shorten the command prompt slightly. Even if you do require another database, you can still miss it out and issue a use database command from within the shell.

Full documentation on using the beeline command and the hive shell environment is available on-line at; <https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients#HiveServer2Clients-Beeline-CommandLineShell>

From the command prompt you can issue any Hive command or query that you like.

For Example;

```
show tables;
describe formatted elec_c;
select * from elec_c limit 10;
```

Although the shell has full functionality, its use is likely to be limited in face of the more appealing GUI type interfaces. However the command line access is not limited to using the shell.

It is also possible to execute a Hive query directly from the beline (or hive) command

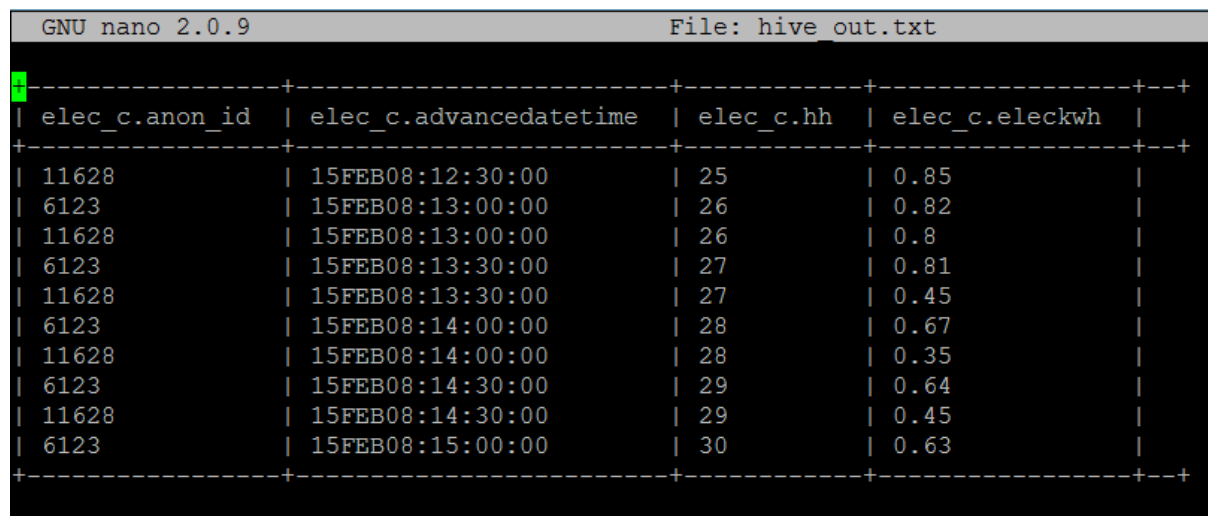
```
beeline -u jdbc:hive2://localhost:10000 -e "select * from elec_c limit 10;"
```

In this case the results are returned directly to the screen without entering the shell.

It is also possible to have the results directed to a file using;

```
beeline -u jdbc:hive2://localhost:10000 -e "select * from elec_c limit 10;" > hive_out.txt
```

Unfortunately, the default output format is not particularly suitable for further use.

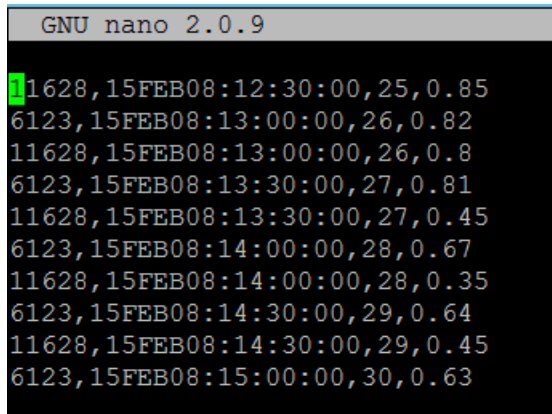


elec_c.anon_id	elec_c.advancedatettime	elec_c.hh	elec_c.eleckwh
11628	15FEB08:12:30:00	25	0.85
6123	15FEB08:13:00:00	26	0.82
11628	15FEB08:13:00:00	26	0.8
6123	15FEB08:13:30:00	27	0.81
11628	15FEB08:13:30:00	27	0.45
6123	15FEB08:14:00:00	28	0.67
11628	15FEB08:14:00:00	28	0.35
6123	15FEB08:14:30:00	29	0.64
11628	15FEB08:14:30:00	29	0.45
6123	15FEB08:15:00:00	30	0.63

It is however possible to overcome this with appropriate command line options to beeline.

```
beeline -u jdbc:hive2://localhost:10000 --showHeader=false --
outputformat=csv2 -e "select * from elec_c limit 10;" > hive_out.csv
```

This results in the following output being written to the file



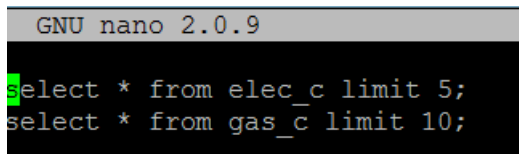
```
GNU nano 2.0.9
11628,15FEB08:12:30:00,25,0.85
6123,15FEB08:13:00:00,26,0.82
11628,15FEB08:13:00:00,26,0.8
6123,15FEB08:13:30:00,27,0.81
11628,15FEB08:13:30:00,27,0.45
6123,15FEB08:14:00:00,28,0.67
11628,15FEB08:14:00:00,28,0.35
6123,15FEB08:14:30:00,29,0.64
11628,15FEB08:14:30:00,29,0.45
6123,15FEB08:15:00:00,30,0.63
```

which is a lot more useful for further processing.

Using Parameters with Hive queries

An alternative to specifying the HQL query as string within the call to beeline, it is also possible to reference a file which contains the HQL statements. This makes it more practical to execute multiple statements within a single call.

With a file containing the following HQL queries



```
GNU nano 2.0.9
select * from elec_c limit 5;
select * from gas_c limit 10;
```

and using the following call to beeline

```
beeline -u jdbc:hive2://localhost:10000 -f test.hql
```

you can get the output

```

0: jdbc:hive2://localhost:10000> select * from elec_c limit 5;
+-----+-----+-----+-----+
| elec_c.anon_id | elec_c.advancedatetime | elec_c.hh | elec_c.eleckwh |
+-----+-----+-----+-----+
| 11628          | 15FEB08:12:30:00      | 25        | 0.85            |
| 6123           | 15FEB08:13:00:00      | 26        | 0.82            |
| 11628          | 15FEB08:13:00:00      | 26        | 0.8             |
| 6123           | 15FEB08:13:30:00      | 27        | 0.81            |
| 11628          | 15FEB08:13:30:00      | 27        | 0.45            |
+-----+-----+-----+-----+
5 rows selected (0.546 seconds)
0: jdbc:hive2://localhost:10000> select * from gas_c limit 10;
+-----+-----+-----+-----+
| gas_c.anon_id | gas_c.advancedatetime | gas_c.hh | gas_c.gaskwh |
+-----+-----+-----+-----+
| 6814          | 25APR10:07:30:00      | 15        | 0.0             |
| 5814          | 25APR10:07:30:00      | 15        | 0.0             |
| 8622          | 25APR10:07:30:00      | 15        | 2.6            |
| 9482          | 25APR10:07:30:00      | 15        | 3.9            |
| 4314          | 25APR10:07:30:00      | 15        | 0.0            |
| 12803         | 25APR10:07:30:00      | 15        | 0.0            |
| 15995         | 25APR10:07:30:00      | 15        | 3.0            |
| 14855         | 25APR10:07:30:00      | 15        | 0.131          |
| 4635          | 25APR10:07:30:00      | 15        | 3.368          |
| 5866          | 25APR10:07:30:00      | 15        | 0.713          |
+-----+-----+-----+-----+
10 rows selected (1.44 seconds)
0: jdbc:hive2://localhost:10000>
Closing: 0: jdbc:hive2://localhost:10000

```

Within a file of HQL commands it is possible to make use of variables which can be substituted when the queries are executed. The small test.hql file above could be written as

```

GNU nano 2.0.9
set hivevar:x=5;
set hivevar:y=10;
select * from elec_c limit ${x};
select * from gas_c limit ${y};

```

with exactly the same results. This can be taken a step further and the setting of the variables can be included on the command line.

So now we will use a new file with the single HQL query of

```

GNU nano 2.0.9 File: table_x_limit_y
select * from ${x} limit ${y};

```

in the file and use

```

beeline -u jdbc:hive2://localhost:10000 -f table_x_limit_y --hivevar
x=elec_c --hivevar y=5

```

will output 5 lines from the elec_c table. Changing the values of x and y on the call allows any number of rows from any table to be output.

Lesson 5 – Accessing Hive via ODBC

Topics in this lesson will include:

- Accessing Hive tables from R
- Accessing Hive data from Excel

ODBC – Open Database Connector (or Connectivity) represents a standard application programming interface (API) for accessing database management systems (DBMS). Hive is an DBMS and an ODBC driver, which is used as a translation layer between an application and Hive, is available from Hortonworks (Microsoft provide one as well).

Once the Hive ODBC driver has been installed on the Windows machine, it is available to all applications which know how to use it. This includes applications such as Excel as well as programming environments such as R and Python.

3rd party application such as Excel treat ODBC connections as just another external data source. In the programming environments access to the ODBC driver is itself managed by other 3rd party API (Application Programming Interface) libraries such as RODBC in R and PyODBC in Python.

Demonstration - Accessing Hive from R using the Hortonworks ODBC

Demonstration – Accessing Hive from Excel using the Hortonworks ODBC