



­

Assignment Document:

Hive

### Hands-On Exercises

**HIVE**

**What is Hive ?**

Hive is a data warehousing infrastructure based on the Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing (using the map-reduce programming paradigm) on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides a simple query language called Hive QL, which is based on SQL and which enables users familiar with SQL to do ad-hoc querying, summarization and data analysis easily. At the same time, Hive QL also allows traditional map/reduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis that may not be supported by the built-in capabilities of the language.

**Hive is NOT**

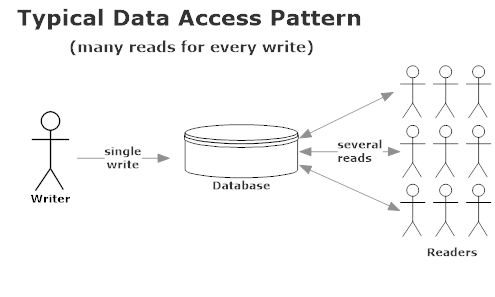
Hive is not designed for online transaction processing and does not offer real-time queries and row level updates. It is best used for batch jobs over large sets of immutable data (like web logs).

Hadoop is a batch processing system and Hadoop jobs tend to have high latency and incur substantial overheads in job submission and scheduling. As a result - latency for Hive queries is generally very high (minutes) even when data sets involved are very small (say a few hundred megabytes). As a result it cannot be compared with systems such as Oracle where analyses are conducted on a significantly smaller amount of data. Hive aims to provide acceptable (but not optimal) latency for interactive data browsing, queries over small data sets or test queries.

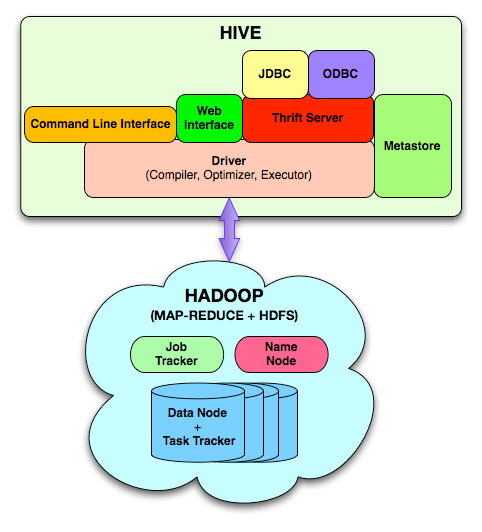
**Why hive**

Mapreduce is good for massive amount of data processingImplementation of Mapreduce is required java and MR is very low level and requires customers to write custom programs.

HIVE supports queries expressed in SQL-like language called HiveQL which are compiled into MR jobs that are executed on Hadoop.

[](http://markedupblog.blob.core.windows.net/wordpress/2013/02/traditionalreadwritecharactertistics.png)

Hive Architecture



**Metastore: stores system catalog**

All the metadata for Hive tables and partitions are stored in Hive Metastore and provides clients (including Hive) access to this information via the metastore service API. Most of the commercial relational databases and many open source datstores are supported. Any datastore that has JDBC driver can probably be used.

Ex: Relational database, (mysql)

**Query compiler:** Compiles HiveQL into a directed acyclic graph of map/reduce tasks

**Execution engines:** The component executes the tasks in proper dependency order; interacts with Hadoop

**Thrift Server:** provides Thrift interface and JDBC/ODBC for integrating other applications.

**Client components:** CLI, web interface, jdbc/odbc interface

Hive data types

|  |  |
| --- | --- |
| Type | Size |
| TINYINT | 1 byte |
| SMALLINT | 2 byte |
| INT | 4 byte |
| BIGINT | 8 byte |
| FLOAT | 4 byte (single precision floating point numbers) |
| DOUBLE | 8 byte (double precision floating point numbers) |
| BOOLEAN | TRUE/FALSE value |
| STRING | Max size is 2GB. |

Relational Operators

|  |  |  |
| --- | --- | --- |
| Relational Operator | Operand types | Description |
| A = B | all primitive types | TRUE if expression A is equivalent to expression B otherwise FALSE |
| A != B | all primitive types | TRUE if expression A is not equivalent to expression B otherwise FALSE |
| A < B | all primitive types | TRUE if expression A is less than expression B otherwise FALSE |
| A <= B | all primitive types | TRUE if expression A is less than or equal to expression B otherwise FALSE |
| A > B | all primitive types | TRUE if expression A is greater than expression B otherwise FALSE |
| A >= B | all primitive types | TRUE if expression A is greater than or equal to expression B otherwise FALSE |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE |
| A IS NOT NULL | all types | FALSE if expression A evaluates to NULL otherwise TRUE |
| A LIKE B | strings | TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The \_ character in B matches any character in A (similar to . in posix regular expressions), and the % character in B matches an arbitrary number of characters in A (similar to .\* in posix regular expressions). For example, 'foobar' LIKE 'foo' evaluates to FALSE where as 'foobar' LIKE 'foo\_\_\_' evaluates to TRUE and so does 'foobar' LIKE 'foo%'. To escape % use \ (% matches one % character). If the data contains a semi-colon, and you want to search for it, it needs to be escaped, columnValue LIKE 'a\;b' |
| A RLIKE B | strings | TRUE if string A matches the Java regular expression B (See [Java regular expressions syntax](http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html)), otherwise FALSE. For example, 'foobar' rlike 'foo' evaluates to FALSE whereas 'foobar' rlike '^f.\*r$' evaluates to TRUE |
| A REGEXP B | strings | Same as RLIKE |

Arithmetic Operators

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

Logical Operators

The following operators provide support for creating logical expressions. All of them return boolean TRUE or FALSE depending upon the boolean values of the operands.

|  |  |  |
| --- | --- | --- |
| Logical Operators | Operands types | Description |
| A AND B | boolean | TRUE if both A and B are TRUE, otherwise FALSE |
| A && B | boolean | Same as A AND B |
| A OR B | boolean | TRUE if either A or B or both are TRUE, otherwise FALSE |
| A | B | boolean | Same as A OR B |
| NOT A | boolean | TRUE if A is FALSE, otherwise FALSE |
| !A | boolean | Same as NOT A |

Operators on Complex Types

|  |  |  |
| --- | --- | --- |
| Operator | Operand types | Description |
| A[n] | A is an Array and n is an int | returns the nth element in the array A. The first element has index 0 e.g. if A is an array comprising of ['foo', 'bar'] then A[0] returns 'foo' and A[1] returns 'bar' |
| M[key] | M is a Map<K, V> and key has type K | returns the value corresponding to the key in the map e.g. if M is a map comprising of {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} then M['all'] returns 'foobar' |
| S.x | S is a struct | returns the x field of S e.g for struct foobar {int foo, int bar} foobar.foo returns the integer stored in the foo field of the struct. |

Built in functions

The following built in functions are supported in hive:  
[(Function list in source code: FunctionRegistry.java)](http://svn.apache.org/viewvc/hive/trunk/ql/src/java/org/apache/hadoop/hive/ql/exec/FunctionRegistry.java?view=markup)

|  |  |  |
| --- | --- | --- |
| Return Type | Function Name (Signature) | Description |
| BIGINT | round(double a) | returns the rounded BIGINT value of the double |
| BIGINT | floor(double a) | returns the maximum BIGINT value that is equal or less than the double |
| BIGINT | ceil(double a) | returns the minimum BIGINT value that is equal or greater than the double |
| double | rand(), rand(int seed) | returns a random number (that changes from row to row). Specifiying the seed will make sure the generated random number sequence is deterministic. |
| string | concat(string A, string B,...) | returns the string resulting from concatenating B after A. For example, concat('foo', 'bar') results in 'foobar'. This function accepts arbitrary number of arguments and return the concatenation of all of them. |
| string | substr(string A, int start) | returns the substring of A starting from start position till the end of string A. For example, substr('foobar', 4) results in 'bar' |
| string | substr(string A, int start, int length) | returns the substring of A starting from start position with the given length e.g. substr('foobar', 4, 2) results in 'ba' |
| string | upper(string A) | returns the string resulting from converting all characters of A to upper case e.g. upper('fOoBaR') results in 'FOOBAR' |
| string | ucase(string A) | Same as upper |
| string | lower(string A) | returns the string resulting from converting all characters of B to lower case e.g. lower('fOoBaR') results in 'foobar' |
| string | lcase(string A) | Same as lower |
| string | trim(string A) | returns the string resulting from trimming spaces from both ends of A e.g. trim(' foobar ') results in 'foobar' |
| string | ltrim(string A) | returns the string resulting from trimming spaces from the beginning(left hand side) of A. For example, ltrim(' foobar ') results in 'foobar ' |
| string | rtrim(string A) | returns the string resulting from trimming spaces from the end(right hand side) of A. For example, rtrim(' foobar ') results in ' foobar' |
| string | regexp\_replace(string A, string B, string C) | returns the string resulting from replacing all substrings in B that match the Java regular expression syntax(See [Java regular expressions syntax](http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html)) with C. For example, regexp\_replace('foobar', 'oo|ar', ) returns 'fb' |
| int | size(Map<K.V>) | returns the number of elements in the map type |
| int | size(Array<T>) | returns the number of elements in the array type |
| *value of <type>* | cast(*<expr>* as *<type>*) | converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) will convert the string '1' to it integral representation. A null is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | Return the date part of a timestamp string: to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | Return the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970 |
| int | month(string date) | Return the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11 |
| int | day(string date) | Return the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1 |
| string | get\_json\_object(string json\_string, string path) | Extract json object from a json string based on json path specified, and return json string of the extracted json object. It will return null if the input json string is invalid |

Aggregate functions

|  |  |  |
| --- | --- | --- |
| Return Type | Aggregation Function Name (Signature) | Description |
| BIGINT | count(\*), count(expr), count(DISTINCT expr[, expr\_.]) | count(\*) - Returns the total number of retrieved rows, including rows containing NULL values; count(expr) - Returns the number of rows for which the supplied expression is non-NULL; count(DISTINCT expr[, expr]) - Returns the number of rows for which the supplied expression(s) are unique and non-NULL. |
| DOUBLE | sum(col), sum(DISTINCT col) | returns the sum of the elements in the group or the sum of the distinct values of the column in the group |
| DOUBLE | avg(col), avg(DISTINCT col) | returns the average of the elements in the group or the average of the distinct values of the column in the group |
| DOUBLE | min(col) | returns the minimum value of the column in the group |
| DOUBLE | max(col) | returns the maximum value of the column in the group |

**Hive Query Language (HQL)**

Hive support SQL like language called hive query language

Language capabilities

* [Hive query language](https://cwiki.apache.org/confluence/display/Hive/LanguageManual) provides the basic SQL like operations. These operations work on tables or partitions. These operations are:
* Ability to filter rows from a table using a where clause.
* Ability to select certain columns from the table using a select clause.
* Ability to do equi-joins between two tables.
* Ability to evaluate aggregations on multiple "group by" columns for the data stored in a table.
* Ability to store the results of a query into another table.
* Ability to download the contents of a table to a local (e.g., nfs) directory.
* Ability to store the results of a query in a hadoop dfs directory.
* Ability to manage tables and partitions (create, drop and alter).
* Ability to plug in custom scripts in the language of choice for custom map/reduce jobs UDFS.

**Creating Tables**

Using below syntax we can create tables:

Create [external ] table <table name> (schema (col1 <col name > data type <col datatype> ,………) row format delimited fields terminated by < delimiter/ seard > [location ]

Example:

CREATE TABLE new\_table (usercode STRING,cr\_date String, Status int) row format delimited fields terminated by ',';

Loading files into tables

Hive does not do any transformation while loading data into tables. Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables.

Syntax

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename

Loading data from local (UNIX ) file system

You can load the text file into a Hive table

Hive is providing a provision to store the local files into HDFS hive table location If your file is located in ‘/tmp/data/’ location with file name of ‘Demodata.csv’.you can use below command to load your comma delimited file into you hive table and then you can query the data.

LOAD DATA LOCAL INPATH '/tmp/data/Demodate.csv' INTO table new\_table;

Loading data from HDFS (DFS) file system.

Hive is providing a provision to store file located in dfs file system to hive table location

If your file is located in ‘/user/hadoop/userdata/’ location with file name of ‘student.csv’.you can use below command to load your comma delimited file into you hive table and then you can query the data.

LOAD DATA INPATH '‘/user/hadoop/userdata/ student.csv’ ' INTO table student;

**External tables in Hive**

Usually when you create tables in hive using raw data in HDFS, it moves them to a different location - "/user/hive/warehouse". If you created a simple table, it will be located inside the data warehouse. The following hive command creates a table with data location at "/user/hive/warehouse/user". 

hive> CREATE TABLE user(id INT, name STRING) ROW FORMAT

DELIMITED FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n' STORED AS TEXTFILE;

Consider that the raw data is located at "/home/admin/data1.txt" and if you issues the following hive command, the data would be moved to a new location at "/user/hive/warehouse/user/data1.txt". 

hive> LOAD DATA INPATH '/home/admin/userdata/data1.txt' INTO TABLE user;

If we want to just do hive queries, it is all fine. When you drop the table, the raw data is lost as the directory corresponding to the table in warehouse is deleted.   
You may also not want to delete the raw data as some one else might use it in map-reduce programs external to hive analysis. It is far more convenient to retain the data at original location via "EXTERNAL" tables.   
To create external table, simply point to the location of data while creating the tables. This will ensure that the data is not moved into a location inside the warehouse directory. 

hive> CREATE TABLE user(id INT, name STRING) ROW FORMAT

DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n' STORED AS TEXTFILE

LOCATION '/home/admin/userdata';

Now you could happily use both Hive HQL queries as well as hand-crafted map-reduce programs on the same data. However, when you drop the table, hive would attempt to delete the externally located data. This can be addressed by explicitly marking the table "EXTERNAL". Try dropping the table; you will see that the raw data is retained. 

hive> CREATE EXTERNAL TABLE user(id INT, name STRING) ROW FORMAT

DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'

STORED AS TEXTFILE

LOCATION '/home/admin/userdata';

There are few more goodies in Hive that surprised me. You can overlay multiple tables all pointing to the same raw data. The following command below will ensure that there are two tables with different schema overlay over the same raw data. This allows you to experiment and create a new table which improves on the previous schema. It also allows you to use different schema for different hive queries. 

**Hive Managed tables**

A Hive table that's managed by hive is called as managed table. One of the main differences between an external and a managed table in Hive is that when an external table is dropped, the data associated with it doesn't get deleted, only the metadata (number of columns, type of columns, terminators, etc.) gets dropped from the Hive metastore. When a managed table gets dropped, both the metadata and data get dropped. I have so far always preferred making tables external because if the schema of my Hive table changes, I can just drop the external table and re-create another external table over the same HDFS data with the new schema.

In hive, the raw data is in HDFS and there is a metadata layer that defines the structure of the raw data. Table is usually a reference to metadata, probably in a mySQL server and it contains a reference to the location of the data in HDFS, type of delimiter to use and so on.

1. With hive managed tables, when you drop a table, both the metadata in

mysql and raw data on the cluster gets deleted.

2. With external tables, when you drop a table, just the metadata gets deleted and the raw data continues to exist on the cluster.

**Hive Queries**

Displaying contents of the table

hive> select \* from <table-name>;

Dropping tables

hive> drop table <table-name>;

Altering tables

Table names can be changed and additional columns can be dropped:

hive> ALTER TABLE events ADD COLUMNS (new\_col INT);

hive> ALTER TABLE events RENAME TO pokes;

Using WHERE Clause

The where condition is a boolean expression. Hive does not support IN, EXISTS or sub queries in the WHERE clause.

hive> SELECT \* FROM <table-name> WHERE <condition>

Using Group by

hive> SELECT deptid, count(\*) FROM department GROUP BY deptid;

Using Join

Only equality joins, outer joins, and left semi joins are supported in Hive. Hive does not support join conditions that are not equality conditions as it is very difficult to express such conditions as a MapReduce job. Also, more than two tables can be joined in Hive.

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

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

DROP TABLE

DROP TABLE [IF EXISTS] table\_name

Removes metadata and data for this table. The data is actually moved to the .Trash/Current directory if Trash is configured. The metadata is completely lost.When dropping an EXTERNAL table, data in the table will NOT be deleted from the file system.When dropping a table referenced by views, no warning is given (the views are left dangling as invalid and must be dropped or recreated by the user).

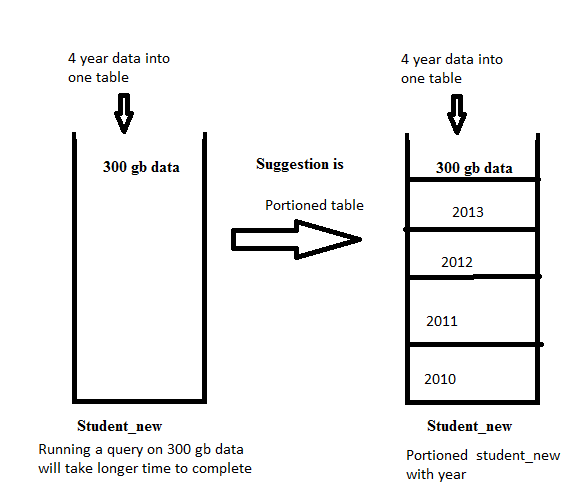
Otherwise, the table information is removed from the metastore and the raw data is removed as if by 'hadoop dfs -rm'. In many cases, these results in the table data being moved into the user's .Trash folder in their home directory; users who mistakenly DROP TABLEs mistakenly may thus be able to recover their lost data by re-creating a table with the same schema, re-creating any necessary partitions, and then moving the data back into place manually using Hadoop. This solution is subject to change over time or across installations as it relies on the underlying implementation; users are strongly encouraged not to drop tables capriciously.

Hive Partitions

Partitioned tables can be created using PARTIONED BY clause. A table can have one or more partition columns and a separate data directory is created for each set of partition columns values. Further

Nested sub-directories in HDFS for each combination of partition column values.

Allows users to efficiently retrieve rows



CREATE TABLE student\_new (name STRING,id int) PARTITIONED BY (date STRING) row format delimited fields terminated by ',';

LOAD DATA LOCAL INPATH '/tmp/student\_new.csv' OVERWRITE INTO TABLE student\_new PARTITION (date='2013-10-22');

Internals of Partitions

In mine, it's /usr/hive/warehouse. Once I navigate to that location, I see the names of my tables. Clicking on a table name (which is just a folder) will then expose the partitions of the table. In my case, I currently only have it partitioned on date. When I click on the folder at this level, I will then see files (more partitioning will have more levels). These files are where the data is actually stored on the HDFS.

Partitions in HDFS

Partition columns: date

/usr/hive/warehouse /student\_new /date=2010

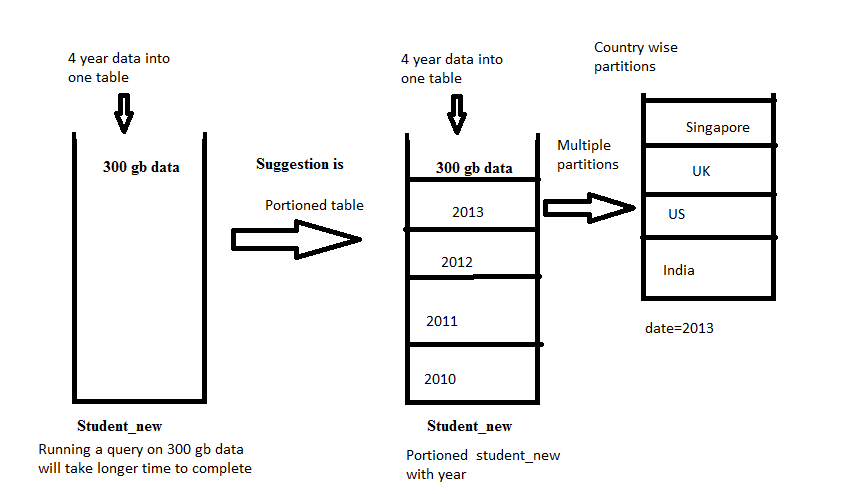
/usr/hive/warehouse/student\_new /date=2011

/usr/hive/warehouse/student\_new /date=2012

/usr/hive/warehouse/student\_new /date=2013

With multiple partitions

Country wise partitions



Partitions in HDFS

Partition columns: date and country

/usr/hive/warehouse /student\_new /date=2010

/usr/hive/warehouse /student\_new /date=2010/country=India

/usr/hive/warehouse /student\_new /date=2010/country=US

/usr/hive/warehouse /student\_new /date=2010/country=UK

/usr/hive/warehouse /student\_new /date=2010/country=singapore

/usr/hive/warehouse/student\_new /date=2011

/usr/hive/warehouse /student\_new /date=2011/country=India

/usr/hive/warehouse /student\_new /date=2011/country=US

/usr/hive/warehouse /student\_new /date=2011/country=UK

/usr/hive/warehouse /student\_new /date=2011/country=singapore

/usr/hive/warehouse/student\_new /date=2012

/usr/hive/warehouse/student\_new /date=2013

Problem: I have a huge table in my EDW that holds 5 billion rows. Every record has a column pay\_location which has 300 distinct values across the table. Need to do some processing on the same within Hadoop environment and at a time my processing involves data only from certain pay\_locations.

Table Schema in DWH

|  |  |
| --- | --- |
| Invoice\_Details | |
| Column Name | Data Type |
| Invoice\_Id | Double |
| Invoice\_Date | Date |
| Invoice\_Amount | Double |
| Pay\_Location | VarChar |
| Paid\_Date | Date |

Solution: We can accomplish the same in 2 easy steps

Step 1: SQOOP import the table into hive from DWH

Step 2: Analyze the data using Hive QL

Fist we need to SQOOP import the data into hive table ‘invoice\_details’ using the basic SQOOP import command as follows.

sqoop import --driver <driver name> --connect <connection string> --username<username> -P --table Invoice\_Details --split-by Invoice\_Id --num-mappers <num of mappers> --warehouse-dir <hdfs dir> --hive-import --hive-table invoice\_details\_hive

We’d look into the second part here in more detail. How to effectively and efficiently analyze the same in hive. In our requirement it is clear that we can go ahead with a partitioned table approach for our data set, as the data analysis is made pay\_location by pay\_location we can do the partition based on pay\_location itself.

Now how can we do the partition? Simple, need to create a partitioned table and load data into each partition. So first we can create an equivalent partitioned table in hive

CREATE TABLE invoice\_details\_hive \_partitioned(Invoice\_Id double, Invoice\_Date string, Invoice\_Amount double,Paid\_Date string)PARTITIONED BY(pay\_location string);

Once table creation is completed we need to load data into the partitions on invoice\_details\_hive \_partitioned from invoice\_details\_hive . How can we do it? Can we go ahead for individual insert statements for each pay\_location like?

INSERT OVERWRITE TABLE invoice\_details\_hive \_partitioned PARTITION(pay\_location=’USA’)

SELECT idh.Invoice\_Id, idh.Invoice\_Date, idh.Invoice\_Amount, idh.Paid\_Date FROM invoice\_details\_hive idh WHERE pay\_location=’USA’;

If we follow this approach we may have to go in for 300 insert statements as there are 300 distinct values for pay\_location in invoice\_details\_hive table. This type of implementation can be called as STATIC PARTIONS. But in our scenario static partitions won’t serve the purpose or rather it is too tedious. We’d have to implement the concept of DYNAMIC PARTITIONS introduced from hive 0.6 onwards. With Dynamic partitions we just need a single Insert Overwrite statement to create and load data into all the partitions.

INSERT OVERWRITE TABLE invoice\_details\_hive \_partitioned PARTITION(pay\_location)

SELECT idh.Invoice\_Id, idh.Invoice\_Date, idh.Invoice\_Amount, idh.Paid\_Date, idh.pay\_location FROM invoice\_details\_hive idh;

This Single Query would implement dynamic partition for you, when you use dynamic partitions the last column from the select query on the source table should be column used for partitioning in the destination table (idh.pay\_location)

When you try executing the query you can see hive throwing some fatal errors, like dynamic partition mode is strict and dynamic partition not enabled. So we need to set the following parameters in hive shell

1.       set hive.exec.dynamic.partition=true;

To enable dynamic partitions, by default it is false

2.       set hive.exec.dynamic.partition.mode=nonstrict;

We are using the dynamic partition without a static partition (A table can be partitioned based on multiple columns in hive) in such case we have to enable the non strict mode. In strict mode we can use dynamic partition only with a Static Partition.

3.       set hive.exec.max.dynamic.partitions.pernode=300;

The default value is 100, we have to modify the same according to the possible no of partitions that would come in your case

4.       set hive.exec.max.created.files=150000;

The default values is 100000 but for larger tables it can exceed the default, so we may have to update the same

In practical scenarios I did find the Dynamic Partition not working with the above query on really large tables and shooting a java print error after completion of first map process. This could be due to the larger number of files created on the first map process. However a slight modification of the job can help you overcome the same, group the records in your hive query on the map process and process them on the reduce side. ie use a map reduce process to achieve your goal rather than two map process. You can implement the same in your hive query itself with the usage of DISTRIBUTE BY, so the modified query would be

FROM invoice\_details\_hive idh

INSERT OVERWRITE TABLE invoice\_details\_hive \_partitioned PARTITION(pay\_location)

SELECT idh.Invoice\_Id, idh.Invoice\_Date, idh.Invoice\_Amount, idh.Paid\_Date, idh.pay\_location

DISTRIBUTE BY pay\_location;

With this approach you don’t need to overwrite the hive.exec.max.created.files parameter.

NUMERIC AND MATHEMATICAL FUNCTIONS IN HIVE

The Numerical functions are listed below in alphabetical order. Use these functions in SQL queries.   
ABS( double n )  
  
The ABS function returns the absolute value of a number.  
Example: ABS(-100)  
  
ACOS( double n )  
  
The ACOS function returns the arc cosine of value n. This function returns Null if the value n is not in the range of -1<=n<=1.  
Example: ACOS(0.5)  
  
ASIN( double n )  
  
The ASIN function returns the arc sin of value n. This function returns Null if the value n is not in the range of -1<=n<=1.  
Example: ASIN(0.5)  
  
BIN( bigint n )  
  
The BIN function returns the number n in the binary format.  
Example: BIN(100)  
  
CEIL( double n ), CEILING( double n )  
  
The CEILING or CEILING function returns the smallest integer greater than or equal to the decimal value n.  
Example: CEIL(9.5)  
  
CONV( bigint n, int from\_base, int to\_base )  
  
The CONV function converts the given number n from one base to another base.  
EXAMPLE: CONV(100, 10,2)  
  
COS( double n )   
  
The COS function returns the cosine of the value n. Here n should be specified in radians.  
Example: COS(180\*3.1415926/180)  
  
EXP( double n )  
  
The EXP function returns e to the power of n. Where e is the base of natural logarithm and its value is 2.718.  
Example: EXP(50)  
  
FLOOR( double n )  
  
The FLOOR function returns the largest integer less than or equal to the given value n.  
Example: FLOOR(10.9)  
  
HEX( bigint n)  
  
This function converts the value n into hexadecimal format.  
Example: HEX(16)  
HEX( string n )  
  
This function converts each character into hex representation format.  
Example: HEX(‘ABC’)  
  
LN( double n )  
  
The LN function returns the natural log of a number.  
Example: LN(123.45)  
  
LOG( double base, double n )  
  
The LOG function returns the base logarithm of the number n.  
Example: LOG(3, 66)  
  
LOG2( double n )  
  
The LOG2 function returns the base-2 logarithm of the number n.  
Example: LOG2(44)  
  
LOG10( double n )  
  
The LOG10 function returns the base-10 logarithm of the number n.  
Example: LOG10(100)  
  
NEGATIVE( int n ),  NEGATIVE( double n )   
  
The NEGATIVE function returns –n  
Example: NEGATIVE(10)  
  
PMOD( int m, int n ), PMOD( double m, double n )   
  
The PMOD function returns the positive modulus of a number.  
Example: PMOD(3,2)  
  
POSITIVE( int n ), POSITIVE( double n )  
  
The POSITIVE function returns n  
Example: POSITIVE(-10)  
  
POW( double m, double n ), POWER( double m, double n )  
  
The POW or POWER function returns m value raised to the n power.  
Example: POW(10,2)  
  
RAND( [int seed] )  
  
The RAND function returns a random number. If you specify the seed value, the generated random number will become deterministic.  
Example: RAND( )  
  
ROUND( double value [, int n] )  
  
The ROUND function returns the value rounded to n integer places.  
Example: ROUND(123.456,2)  
  
SIN( double n )   
  
The SIN function returns the sin of a number. Here n should be specified in radians.  
Example: SIN(2)  
  
SQRT( double n )  
  
The SQRT function returns the square root of the number  
Example: SQRT(4)  
  
UNHEX( string n )  
  
The UNHEX function is the inverse of HEX function. It converts the specified string to the number format.  
Example: UNHEX(‘AB’)