INCEPTEZ TECHNOLOGIES HIVE WORKOUTS

HIVE Installation with metastore configuration

1) Copy the hive tar file to your home path (/home/hduser/install) and extract using below command

```
cd /home/hduser/install/
tar xvzf apache-hive-1.2.2-bin.tar.gz
sudo mv apache-hive-1.2.2-bin /usr/local/hive
```

2) After you complete the above steps execute below commands to create directories and give permissions

```
hadoop fs -mkdir -p /user/hive/warehouse/
hadoop fs -chmod g+w /user/hive/warehouse
hadoop fs -mkdir -p /tmp
hadoop fs -chmod g+w /tmp
```

HIVE Workouts & Usecases

Login to hive cli

```
hive
```

Create Database

The Create Database statement is used to create a database in Hive. By default, there is a database in Hive named default.

The general format of creating a database is as follows:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
[COMMENT database_comment]
[LOCATION hdfs_path]
[WITH DBPROPERTIES (property_name=property_value, ...)];
```

Where:

DATABASE|SCHEMA: These are the same thing. These words can be used interchangeably.

[IF NOT EXISTS]: This is an optional clause. If not used, an error is thrown when there is an attempt to create a database that already exists.

[COMMENT]: This is an optional clause. This is used to place a comment for the database. This comment clause can be used to add a description about the database. The comment must be in single quotes.

[LOCATION]: This is an optional clause. This is used to override the default location with the

preferred one.

[WITH DBPROPERTIES]: This is an optional clause. This clause is used to set properties for the database. These properties are key-value pairs that can be associated with the database to attach additional information with the database.

```
DESCRIBE DATABASE [EXTENDED] db_name;
```

```
create database retail;
```

```
create database if not exists retail_tmp
comment 'retail database for holding retail cust info'
location '/user/hduser/hivestore'
with dbproperties ('Created by' = 'Inceptez', 'Created on' = '2019-01-01');
```

```
describe database retail_tmp;
```

Drop DataBases:

DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];

Where:

DATABASE|SCHEMA: These are the same thing. These words can be used interchangeably.

[IF EXISTS]: This is an optional clause. If not used, an error is thrown when there is an attempt to drop a database that does not exist.

[RESTRICT|CASCADE]: This is an optional clause. RESTRICT is used to restrict the database from getting dropped if there are one or more tables present in the database. RESTRICT is the default behavior of the database. CASCADE is used to drop all the tables present in the database before dropping the database.

```
drop database retail_tmp;
drop database if exists retail_tmp;
drop database if exists retail_tmp cascade;
```

Use Database:

The USE DATABASE command is used to switch to the database, or it sets the database as the working database. It is analogous to the one used in the other RDBMS. The general format of using a database is as follows:

USE (DATABASE|SCHEMA) database_name;

Where:
DATABASE|SCHEMA: These are the same thing. These words can be used interchangeably.

```
use retail;
```

To Print the current database name in the CLI.

```
set hive.cli.print.current.db=true;
```

The **SHOW DATABASE** command is used to list all the databases in the Hive metastore. The general format of using the SHOW DATABASE command is as follows:

SHOW (DATABASES|SCHEMAS) [LIKE identifier_with_wildcards];

```
show databases like 'ret*';
```

Create table for storing transactional records

The CREATE TABLE statement creates metadata in the database. The table in Hive is the way to read data from files present in HDFS in the table or a structural format.

The general format of using the CREATE TABLE command is as follows:

```
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]
[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];
```

Let us take a look at all the parameters involved:

[TEMPORARY]: This is an optional clause. This clause is used to create temporary tables. These tables once created are only present in the database until the session is active. Once the session comes to an end, all the temporary tables are deleted. Once a temporary table is created, you cannot access the permanent table in that session so you need to either drop or rename the temporary table to access the original one. You cannot create a partition or index on temporary tables.

[EXTERNAL]: This is an optional clause. This clause is used to create external tables the same as in the case of RDBMS. The external table works as a window for the data present in the file format in HDFS. For an external table, the data or file need not be present in the default location but can be kept anywhere in the filesystem and can be referred to from that location. Once the external table is dropped, data is not lost from that location.

[IF NOT EXISTS]: This is an optional clause. If there is an attempt to create a table that is already present in the database, an error is thrown. To avoid such an error, the IF NOT EXISTS clause is used. When this clause is used, Hive ignores the statement if the table already exists.

[db_name]: This is an optional clause. This clause is used to create tables in the specified database.

[COMMENT col_comment]: This is an optional clause. This is used to attach comments to a particular column. This comment clause can be used to add a description about the column. The comment must be in single quotes.

[COMMENT table_comment]: This is an optional clause. This is used to attach comments to a table. This comment clause can be used to add a description about the table. The comment must be in single quotes.

[PARTITIONED BY]: This is an optional clause. This clause is used to create partitioned tables. There can be more than one partition columns in a table. Partitions in Hive work in the same way as

in any RDBMS. They speed up the query performance by keeping the data in specific partitions.

[CLUSTERED BY]: This is an optional clause. This clause is used for bucketing purposes. The table or partitions can be bucketed using CLUSTERED BY columns.

[LOCATION hdfs_path]: This option is used while creating external tables. This is the location where files are placed, which is referred to by the external table for the data.

[TBLPROPERTIES]: This is an optional clause. This clause allows you to attach more information about the table in the form of a key-value pair.

[AS select_statement]: Create Table As Select, popularly known as **CTAS**, is used to create a table based on the output of the other table or existing table.

Types of Tables in Hive:

1. **Managed/internal**
2. **External**

Managed Table	External Table
Schema and data will be lost when dropped	Only Schema will be lost when dropped
Used for staging layer	Used for Consumption layer
Hive only loads and access	External applications loads and access
Location will not be specified, default warehouse location will be taken	External location will be specified

Creating managed tables:

```
create table txnrecords(txnno INT, txndate STRING, custno INT, amount DOUBLE, category STRING, product STRING, city STRING, state STRING, spendby STRING)
row format delimited fields terminated by ','
lines terminated by '\n'
stored as textfile;
```

Load the data into the table [From Linux client]

```
LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns' INTO TABLE txnrecords;

LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns' OVERWRITE INTO TABLE txnrecords;
```

Load the data into the table [From HDFS]

In Linux terminal

```
cd /home/hduser/hive/data/

hadoop fs -copyFromLocal txns txns1
```

In Hive cli

```
LOAD DATA INPATH '/user/hduser/txns1' OVERWRITE INTO TABLE txnrecords;
```

Select the loaded data

Map/reduce execution check?

```
select * from txnrecords limit 10;
select * from txnrecords where category='Puzzles';
select * from txnrecords order by 1 limit 10;
```

Creating External tables:

```
create external table externaltxnrecords (txnno INT, txndate STRING, custno INT, amount DOUBLE, category
STRING, product STRING, city STRING, state STRING, spendby STRING)
row format delimited fields terminated by ','
stored as textfile
location '/user/hduser/hiveexternaldata';
```

Describing metadata or schema of the table

```
describe formatted txnrecords;

describe formatted externaltxnrecords;

show create table txnrecords;
```

Using complex data types

Hive supports a few complex data types: Struct, MAP, and Array. Complex data types are also known as collection data types. Most relational databases don't support such data types.

Complex data types can be built from primitive data types:

STRUCT: The struct data type is a record type that holds a set of named fields that can be of any primitive data types. Fields in the

STRUCT type are accessed using the DOT (.) notation.

Syntax:

STRUCT<col_name : data_type [COMMENT col_comment], ...>

For example, if a column address is of the type

STRUCT {city STRING; state STRING}, then the city field can be referenced using address.city.

MAP: The map data type contains key-value pairs. In Map, elements are accessed using the keys.

For example, if a column name is of type Map: 'firstname' -> 'john' and 'lastname' -> 'roy', then the last name can be accessed using the name ['lastname'].

ARRAY: This is an ordered sequence of similar elements. It maintains an index in order to access the elements; for example, an array day, containing a list of elements ['Sunday', 'Monday', 'Tuesday', 'Wednesday']. In this, the first element Sunday can be accessed using day[0], and similarly, the third element can be accessed using day[2].

Array Example

```
create table arraytbl (id int,name string,sal bigint,desig array<string>,city string)
row format delimited fields terminated by ','
collection items terminated by '$';
```

View the data with below contents:

```
!cat /home/hduser/hive/data/arraydata;
```

Data

1,Arvindh,40000,SWE$Analyst$SAnalyst$PL,hyd
2,Bala,30000,SWE$Analyst$SAnalyst$PM,hyd
3,Chandra,50000,SWE$Analyst$PL$PM,Che
4,Gokul,30000,SWE$Analyst$SAnalyst,hyd

```
load data local inpath '/home/hduser/hive/data/arraydata' overwrite into table arraytbl;

select name,desig[2],sal from arraytbl;
```

Struct Example

```
create table Struct(id int,name string,sal bigint,addr struct<city:string,state:string,pin:bigint>)
row format delimited
fields terminated by ','
collection items terminated by '$';

!cat /home/hduser/hive/data/struct;
```

Data

1,Arvindh,40000,HyderabadAP800042
2 ,Bala,30000,Chennai$TamilNadu$ 600042

load data local inpath '/home/hduser/hive/data/struct' overwrite into table Struct;

select addr.city from struct;

Map Example

```
create table map1 (id int,name string,sal bigint,Mark map<string,int>,city string)
row format delimited
fields terminated by ','
collection items terminated by '$'
map keys terminated by '#';
```

Data

1,Arvindh,40000,mat#100$Eng#99,hyd
2,Bala,30000,Sci#100$Eng#99,chn

load data local inpath '/home/hduser/hive/data/map1' overwrite into table map1;

Select Mark["mat"] from map1;

Exploring indexes

Indexes are useful for increasing the performance of frequent queries based on certain columns. But Hive has limited a capability to index data as indexing large datasets requires sufficient additional storage space and processing overheads. Hive can index the columns to speed up some operations. It stores the indexed data in another table.

```
create index idx_custno on table txnrecords(custno) AS
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' WITH DEFERRED REBUILD;

ALTER INDEX idx_custno ON txnrecords REBUILD;

SHOW FORMATTED INDEX ON txnrecords;

DROP INDEX idx_custno ON txnrecords;
```

Hive partitioning

Huge amount of data which is in the range of petabytes is getting stored in **HDFS**. So due to this, it becomes very difficult for Hadoop users to query this huge amount of data. The Hive was introduced to lower down this burden of data querying. Apache Hive converts the SQL queries into MapReduce jobs and then submits it to the **Hadoop cluster**. When we submit a SQL query, Hive read the entire data-set. So, it becomes inefficient to run **MapReduce jobs** over a large table. Thus this is resolved by creating partitions in tables. Apache Hive makes this job of implementing partitions very easy by creating partitions by its automatic partition scheme at the time of table creation. In Partitioning method, all the table data is divided into multiple partitions. Each partition corresponds to a specific value(s) of partition column(s). It is kept as a sub-record inside the table's record present in the HDFS. Therefore on querying a particular table, appropriate partition of the table is queried which contains the query value. Thus this decreases the I/O time required by the query. Hence increases the performance speed. Hive Partitions is a way to organizes tables into partitions by dividing tables into different parts based on partition keys. Partition is helpful when the table has one or more Partition keys. Partition keys are basic elements for determining how the data is stored in the table.

Suppose a telecom organization generates 1 TB of data every day and different regional managers query this data based on their own state. For each query by a regional manager, Hive scans the complete data in HDFS and files the results for a particular state. The manager runs the same query daily for his own state analysis and the query gives the result in few hours on a 1 TB dataset. For analytics, the same query could be executed daily on a one-month or six month dataset. The query would take few hours on a month's data. If the data is somehow partitioned based on state, then when a regional manager runs the same query for his state, only the data of that state is scanned and the execution time could be reduced significantly to minutes.

Hive Partitioning Advantages

- Partitioning in Hive distributes execution load horizontally.
- In partition faster execution of queries with the low volume scan of data takes place.
- Partition avoids full table scan.

Hive Partitioning Disadvantages

- There is the possibility of too many small partition creations- too many directories.

Partition
Low cardinal columns (date, country etc)
Creates folder
Improves where clause performance

Partitioning can be done in one of the following two ways:

1. **Static partitioning**
2. **Dynamic partitioning**

Static Partition	Dynamic Partition
Need to understand and define partition	No need to know about the data, automatically created
Load command or insert select	Load command or insert select
Can be derived from external parameters such as file name or region or timezone etc	Can be derived from one or more columns from the data
Faster when using load command	Slower load because of mapreduce invocation always.

Static partitioning

In static partitioning, you need to manually insert data in different partitions of a table. Let's use a table partitioned on the states of India. For each state, you need to manually insert the data from the data source to a state partition in the partitioned table. So for 29 states, you need to write the equivalent number of Hive queries to insert data in each partition.

Hive Static Partitioning

- Insert input data files individually into a partition table is Static Partition.
- Usually when loading files (big files) into **Hive tables** static partitions are preferred.
- Static Partition saves your time in loading data compared to dynamic partition.
- You "statically" add a partition in the table and move the file into the partition of the table.
- You can perform Static partition on Hive Manage table or external table.

I. Create partitioned table

In Linux terminal

```
cp /home/hduser/hive/data/txns /home/hduser/hive/data/txns_20181212_PADE
cp /home/hduser/hive/data/txns /home/hduser/hive/data/txns_20181212_NY
```

a. Static Partition Load in managed table:

1. load command method:

```
create table txnrecsbycatdtreg (txnno INT, txndate STRING, custno INT, amount DOUBLE, category
STRING,product STRING, city STRING, state STRING, spendby STRING)
partitioned by (datadt date,region string)
row format delimited fields terminated by ','
stored as textfile;

LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns_20181212_PADE'
OVERWRITE INTO TABLE txnrecsbycatdtreg
PARTITION (datadt='2018-12-12',region='PADE');

LOAD DATA LOCAL INPATH '/home/hduser/hive/data/txns_20181212_NY'
```

```
OVERWRITE INTO TABLE txnrecsbycatdtreg
PARTITION (datadt='2018-12-12',region='NY');

dfs -ls /user/hive/warehouse/retail.db/txnrecsbycatdtreg/datadt=2016-12-12/region=PADE;

select count(1) from  txnrecsbycatdtreg where datadt='2016-12-12' and region in ( 'PADE', 'NY');

show partitions txnrecsbycatdtreg partition(datadt='2018-12-12');
```

2. **Insert select method**

Note: As we are deriving the partition column from one of the column (category) from the file data, we need to ignore that column from the create table column list.

```
create table managedtxnrecsByCat(txnno INT, txndate STRING, custno INT, amount DOUBLE, product
STRING, city STRING, state STRING, spendby STRING)
partitioned by (category STRING) row format delimited
fields terminated by ','
stored as textfile;

Insert into table managedtxnrecsbycat partition (category='Games')
select txnno,txndate,custno,amount,product,city,state,spendby
from txnrecords
where category='Games';
```

External table with partition

```
create external table txnrecsByCat(txnno INT, txndate STRING, custno INT, amount DOUBLE, product
STRING, city STRING, state STRING, spendby STRING) partitioned by (category STRING)
row format delimited fields terminated by ','
stored as textfile
location '/user/hduser/hiveexternaldata';

dfs -mkdir -p /user/hduser/hiveexternaldata;
```

Load data into dynamic partition table

Dynamic Partition:

- If hive dynamically decides on the creation of the partitions based on the data then it is dynamic partition.
- Dynamic partitions can be only loaded using insert select.
- Dynamic Partition takes more time in loading data compared to static partition as it uses mapreduce.

- You can perform dynamic partition on hive external table and managed table.
- If you want to use the Dynamic partition in the hive then the mode is in non-strict mode.

Set the below environmental variables

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

However, a query across all partitions could trigger an enormous MapReduce job if the table data and number of partitions are large. A highly suggested safety measure is putting Hive into strict mode, which prohibits queries of partitioned tables without a WHERE clause that filters on partitions. You can set the mode to nonstrict, as in the following session:

Note: Here the partition column is added in the last and included in the select statement in the last.

Insert into table txnrecsbycat partition (category)
select txnno,txndate,custno,amount, product,city,state,spendby,category
from txnrecords;

Additional Partition options:
Adding new partitions

The following command can be used to add new partitions to a table:

ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec [LOCATION 'loc1'] partition_spec [LOCATION 'loc2'] ...;

Where partition_spec:

: (partition_column = partition_column_value, partition_column = partition_column_value, ...)

You can add multiple partitions to a table using the preceding command.

In Linux terminal

cp /home/hduser/hive/data/txns /home/hduser/hive/data/txns_20181215_PADE

hadoop fs -mkdir -p /user/hive/warehouse/retail.db/txnrecsbycatdtreg/datadt=2018-12-15/region=PADE/

hadoop fs -put /home/hduser/hive/data/txns_20181215_PADE
/user/hive/warehouse/retail.db/txnrecsbycatdtreg/datadt=2018-12-15/region=PADE/

Hive Cli

```
alter table txnrecsbycatdtreg add if not exists partition (datadt='2018-12-15',region='PADE')
location '/user/hive/warehouse/retail.db/txnrecsbycatdtreg/datadt=2018-12-15/region=PADE/' ;

show partitions txnrecsbycatdtreg partition(datadt='2018-12-15');
```

Renaming partitions

The following command can be used to rename a partition:

ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION partition_spec;

```
alter table txnrecsbycatdtreg partition (datadt='2018-12-15',region='PADE')
rename to partition (datadt='2018-12-14',region='NJ');
```

Drop Partitions:

ALTER TABLE myTable DROP PARTITION partition_spec, PARTITION partition_spec;

```
ALTER TABLE txnrecsbycatdtreg drop partition (datadt='2018-12-12');
```

Bucketing

Unlike partitioning where each value for the slicer or partition key gets its own space, in clustering a hash is taken for the field value and then distributed across buckets. In the example, we created 10 buckets and are clustering on state. Each bucket then would contain multiple states.

Bucketing is a technique that allows you to decompose your data into more manageable parts, that is, fix the number of buckets. Usually, partitioning provides a way of segregating the data of a Hive table into multiple files or directories. Partitioning is used to increase the performance of queries, but the partitioning technique is efficient only if there is a limited number of partitions. Partitioning doesn't perform well if there is a large number of partitions; for example, we are doing partitioning on a column that has large number of unique values, then there will be a large number of partitions. To overcome the problem of partitioning, Hive provides the concept of bucketing. In bucketing, we specify the fixed number of buckets in which entire data is to be decomposed. Bucketing concept is based on the hashing principle, where same type of keys are always sent to the same bucket. In bucketing, records with the same bucketed columns will always go to the same bucket. When data is inserted into a bucketed table, the following formula is used to derive the bucket into which record should be inserted:

- ✓ As the data files are equal sized parts, map-side joins will be faster on bucketed tables than non-bucketed tables. In Map-side join, a mapper processing a bucket of the left table knows that the matching rows in the right table will be in its corresponding bucket, so it only retrieves that bucket .
- ✓ Similar to partitioning, bucketed tables provide faster query responses than non-bucketed tables.
- ✓ Bucketing concept also provides the flexibility to keep the records in each bucket to be sorted by one or more columns. This makes map-side joins even more efficient

Partition vs Buckets:

Partition	Buckets
Low cardinal columns (date, country etc)	Low or high cardinal columns
Creates folder	Creates files
Improves where clause performance	Improves Joins and where clause performance

Bucket number = hash_function(bucketing_column) mod num_buckets

Once bucketing is enabled, you can create a bucketed table using the following command:

CREATE [EXTERNAL] TABLE [db_name.]table_name
[(col_name data_type [COMMENT col_comment], ...)]
CLUSTERED BY (col_name data_type [COMMENT col_comment], ...)
INTO N BUCKETS;

Create bucketed table over the partitioned table

Use the following command set hive.enforce.bucketing = true; allows the correct number of reducers and the cluster by column to be automatically selected based on the table.

```
set hive.enforce.bucketing = true ;

create table buckettxnrecsByCat(txnno INT, txndate STRING, custno INT, amount DOUBLE, product
STRING, city STRING, state STRING, spendby STRING)
partitioned by (category STRING)
clustered by (state) sorted by (city) INTO 10 buckets
row format delimited fields terminated by ','
stored as textfile
location '/user/hduser/hiveexternaldata/bucketout';

Insert into table buckettxnrecsByCat partition (category)
select txnno,txndate,custno,amount,product,city,state,spendby,category from txnrecords;

select count(1)

from buckettxnrecsByCat

where category='Games' and state='Baltimore';


SELECT txnno,product,state FROM buckettxnrecsByCat TABLESAMPLE (BUCKET 3 OUT OF 10);

SELECT txnno,product,state FROM buckettxnrecsByCat TABLESAMPLE (0.1percent);
```

==========

Remote Metastore configuration

==========

1. MYSQL DB changes:
   ==========

Start the mysql service

---

sudo service mysqld start

2. Login to mysql using root user to create metastore db:

---

mysql -u root -p

Enter password: **root**

create database metastore;

USE metastore;
SOURCE /usr/local/hive/scripts/metastore/upgrade/mysql/hive-schema-0.14.0.mysql.sql;

show tables;

GRANT all on *.* to 'hiveuser'@localhost identified by 'hivepassword';

flush privileges;

quit;

3. HIVE-SITE.XML configuration to connect to remote metastore

==========

Copy the hive-site_rms_transction.xml config file provided to /usr/local/hive/conf

---

cp /home/hduser/hive/hive-site_rms_transction.xml /usr/local/hive/conf/hive-site.xml

Copy the dependent mysql connector jdbc jar to get hive connected with remote metastore

cp -p /home/hduser/install/mysql-connector-java.jar /usr/local/hive/lib/

Note : Below are the additions made in the hive-site.xml

```
<property>
<name>hive.metastore.uris</name>
<value>thrift://localhost:9083</value>
<description>IP address (or fully-qualified domain name) and port of the metastore host</description>
< /property>
<property>
<name>javax.jdo.option.ConnectionURL</name>
 <value>jdbc:mysql://localhost/metastore</value>
<description>the URL of the MySQL database</description>
< /property>
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>com.mysql.jdbc.Driver</value>
</property>
<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>hiveuser</value>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>hivepassword</value>
</property>
<property>
<name>datanucleus.autoCreateSchema</name>
< value>false</value>
</property>
<property>
< name>hive.stats.dbclass</name>
<value>jdbc:mysql</value>
 </property>
```

4. Running the services before starting hive session

==========

Note: Run the below command in a different terminal and leave the terminal as is

```
hive --service metastore
```

Note: Use the below commands to start hive always:

```
sudo service mysqld start
hive --service metastore
```

Starting Hive CLI with remote metastore

```
hive
```

Export hive data to a file:

```
insert overwrite local directory '/home/hduser/exptxnrecords'
row format delimited fields terminated by ','
select txnno,txndate,custno,amount, product,city,state,spendby
from txnrecords where category='Games';
```

Serialization and deserialization formats and data types

Serialization and deserialization formats are popularly known as **SerDes**. Hive allows the framework to read or write data in a particular format. These formats parse the structured, semistructured or unstructured data bytes stored in HDFS in accordance with the schema definition of Hive tables. Hive provides a set of in-built SerDes and also allows the user to create custom SerDes based on their data definition.
These are as follows:

Json Serde
Xml Serde
Lazy Simple Serde
Csv Serde
Regex Serde

Banking xml data for serde:

Copy the serde jars and other dependent jars to hive/lib

```
cp -p /home/hduser/install/hivexmlserde-1.0.5.3.jar /usr/local/hive/lib/

CREATE TABLE xml_bank(customer_id STRING, income BIGINT, demographics map<string,string>,
financial map<string,string>)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
"column.xpath.customer_id"="/record/@customer_id",
"column.xpath.income"="/record/income/text()",
"column.xpath.demographics"="/record/demographics/*",
"column.xpath.financial"="/record/financial/*"
)
STORED AS
INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
location '/user/hduser/xmlserdebank/'
TBLPROPERTIES (
"xmlinput.start"="<record customer",
"xmlinput.end"="</record>"  );
```

Copy the file to HDFS

```
hadoop fs -mkdir /user/hduser/xmlserdebank/

hadoop fs -put /home/hduser/hive/data/bankserde.xml /user/hduser/xmlserdebank/

select * from xml_bank;
```

UDF using Python code:

```
create table customerall(custno string, firstname string, lastname string, age int,profession string)
row format delimited fields terminated by ',';

load data local inpath '/home/hduser/hive/data/custs' into table customerall;

SELECT * FROM customerall limit 10;
```

Create the below python udf:

```
vi  /home/hduser/mask.py

#!/usr/bin/env python
import sys
import string
import hashlib

while True:
  line = sys.stdin.readline()
  if not line:
    break

  line = string.strip(line, "\n ")
  custno, firstname, lastname, age, profession = string.split(line, "\t")
  print "\t".join([custno, firstname,lastname,hashlib.sha256(age).hexdigest(),
hashlib.md5(profession).hexdigest()])

add FILE /home/hduser/mask.py;

select transform(custno,firstname,lastname,age,profession) using 'python /home/hduser/mask.py' as
(custno,firstname,lastname,age,profession) from customerall limit 10;
```

==========
Hive Sqoop Integeration
==========

Create a hive table using sqoop as like mysql table

```
sqoop create-hive-table --connect jdbc:mysql://127.0.0.1/test --username hiveuser --password
hivepassword --table customer \
--hive-table retail.customer;
```

Import data from sqoop directly to hive customer table

```
hadoop fs -rmr /user/hduser/customer
```

```
sqoop import --connect jdbc:mysql://127.0.0.1/test --username hiveuser –password hivepassword --table
customer \
--hive-import -hive-table retail.customer_data -m 1
```