Hive Instructions

The following are steps to interact with Hive for the Estimates generation process. (This file has had project-specific information redacted)

Access to Hive

1. Log into the Prod server, which is an edge node (adjacent) to the Hive cluster, so we can subsquently access Hive.
    1. `ssh <user@server.domain>`
    2. Type `yes` if asked to continue connecting (may not be).

Issuing Commands to Hive

From on the Server

All three of these methods require logging onto the 30 server as mentioned above in order to work.

1. **Inside Hive CLI**:

    1. We can enter the Hive CLI itself directly by using the command `hive`.
    2. Enter `use <dbname>;` to get to the project database.
    3. From here we can enter HQL (Hive Query Language, essentially SQL with Hive-specific aspects when required, like for `CREATE TABLE` where we must also specify a location for Hive to create a table).
    4. Use the command `show tables;` to list all the tables in the database
        - Can use Regex to narrow search: `show tables like "*<table name>*"` will return all tables with *<table name>* in their name, for example.
    5. *[Opinion]*: This requires logging onto the server via the Terminal. This will probably be the clearest approach to do basic data work in Hive for smaller, single query noodling.

2. **Server Command**:

    1. Instead of logging into Hive itself, we can issue queries directly from the Server command line.
    2. `hive -e 'use <dbname>; select * from <table name> limit 10;'`
    3. `hive -e` means "execute the following command in Hive"
    4. `use <dbname>;` tells Hive which database we are using
    5. `select * from <table name> limit 10;` is an example query.
    6. You can string together multiple queries into a single command, seperated by a semi-colon, such as `select * from <table name> limit 10; select * from <table name2> limit 10;`
    7. The output is printed to the console screen by default.
    8. The entire query or chain of queries (everything following the `-e`) must be enclosed in quotes.

3. **Run a Script from the Server**:

1. This will be done for longer SQL queries, like the metrics-generating `<script name>.sql` file, and the `.sh` shell scripts.
2. Upload file to specific location on server with `scp <local file path> <user@server.domain>:/home/dir/<user profile>`
3. Log into 30 server as mentioned above: `ssh <user@server.domain>`
4. Navigate to the file you just uploaded and run it.
5. **NOTE:** If we cannot run a `.sql` file directly from command line on server for Hive, might have to be a `.sh` script. In which case, makes more sense to put the large `.sql` file queries into a single command issued by the "Server Command" method above.
   Update:
   1. [Data Architect] says try `hive -f <filename>.hql` (might have to convert `.sql` to `.hql` for this)
   2. Check this article

From Within a Python Script

Due to the setup of our internal gatekeeping system, we cannot use some packages which make this process easy (such as PyHive). There are a couple modules in Python that allow the user to login to a server and issue commands on the server all from a Python script. I've used `paramiko` for this and have confirmed it works. Example code here. As noted above, issuing a Hive command from the server (instead of inside the Hive CLI) requires the form of `hive -e "<command>"`. Note the command to execute, following the `-e` argument, must be enclosed in quotes.

The benefit of this paradigm is to allow the issuing of commands to the Hive database from within a Python script, ideally keeping the workflow inline and reduce fracturing.

Paramiko can be installed via either `conda` or `pip`:

```
conda install paramiko
# or
pip install paramiko
```

```python
import paramiko

### SSH into the Server first
client = paramiko.SSHClient()
server="<user@server.domain>"
username="<user name>"

## Required to access server permissions
client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(server, username=username, password=None)   ## works w/o PW

table = '<large table name>'     ## Large table for testing limits
limit = 10000
cmd_to_execute = f"hive -e 'use <dbname>; select * from {table} limit
{limit};'"
```

```
ssh_stdin, ssh_stdout, ssh_stderr = client.exec_command(cmd_to_execute)
```

Output to a DataFrame

The key output is the `ssh_stdout` object, which holds the query's results. It is a `file-object` that can be read by built-in Python methods, like `.read()` or `.readlines()`.

The goal is to create a Pandas `DataFrame` from the output of a given query. The output from `.read()` method is a single `bytes` string, and the `.read()` method has a hard row limit based on performance. I've done test runs and plotted the results. Under 100,000 rows most queries are at or under 60 seconds. However, above 100,000 rows the `.read()` method takes on an exponential behavior as the entire *file-object* (the `bytes` return of the Hive query) is pushed into memory all at once. There aren't many queries in the National Estimates generation process that are over 100,000 lines, but this performance limit means another alternative is needed when there are.

A faster way is by using `.readlines()` iteratively in chunks and appending to form a larger DataFrame. This is implemented in the stand-alone module I've written, called `hive_pandas.py`. Its performance doesn't hit the same bottleneck as `.read()`. Up to about 15,000 rows the two methods perform similarly. Past that, the iterative `.readlines()` method is much faster. The largest table used in this process is around 2.3 million rows and growing. The `.read()` method would be projected to take 6+ hours to process this (assuming it even would). `.readlines()` iteration does it in 126 seconds -- two minutes. This means every query to Hive can be handled in the native Python script as was done in Netezza.

Acquiring Data from Hive

Via SFTP

This is a two-step process. It is slower as a result. Ideally this method will not have to be used in the National Estimates monthly generation process since the option via Python was created below. (I do not know about other parts of the project).

1. Save query from Hive to a file on the server.
   The following example (and very ugly) command shows how to save a query to a text file.

   `hive --showHeader=false --outputformat=tsv2 -e "use <dbname>;select * from <table_name> limit 10" | sed 's/[\t]/,/g' |sed -e 's/NULL//g' | sed's/[[:space:]]\+//g' > tmp.txt`

   `sed` is a streaming editor. Its commands take the form of `s/<find this>/<replace with this>/`, where the `<find this>` is a *regular expression*. This specific command replaces tabs with commas and NULLs with spaces.

   Obviously, save `tmp.txt` to the correct directory on the server, such as your user folder.

2. Use SecureCopy to download the saved file from the Hive server to our local machine. This might require that your Public Key is added to Hive server list. (a specialist helped me do this part).

   `scp <server user>@<server.domain>:/home/<server user>/jonpaul <local file path>`

Via Python

This uses the `hive_pandas.py` module I wrote, referenced above. It requires the `ssh_stdout` object returned from the Paramiko `client.exec_command(cmd_to_execute)` method above.

Here is example code showing how to import and use this module. Place the `hive_pandas.py` module where you want on your local machine (by default will be in the `utilities` folder within the project dir). You will reference this directory using the `sys.path.append()` method, which simply adds this directory to the list of places to look for any module you're trying to import -- for this script only.

```
import sys

path_to_hive_pandas_pkg = '<your local path here>'   ## stored in
'utilities' folder in SS
sys.path.append(path_to_hive_pandas_pkg)
import hive_pandas as hp

df = hp.process_hive_df(ssh_stdout)
```

The `process_hive_df()` function attempts some general DataFrame tidying, such as dropping fully NULL / NaN columns and dtype conversions to numeric or categorical, as everything is considered a `str` or `object` when reading from the `ssh_stdout` output. This tidying process is not robust and may require some additional post-hoc cleaning in specific scripts.

The good news is that this allows acquisition of any dataset from Hive within Python.

Getting Data into Hive

Hive has *internal* and *external* table types. The key differences are outlined in this Microsoft Hive FAQ

The primary difference is that Internal ("Managed") tables will have their data removed when they are dropped, while External tables will only have the references (metadata) to them removed.

Per team specialists, we will be using mostly External tables in our project.

External Table Process Overview

1. Copy data from local machine to 30 server
2. `put` the data into the Hadoop File System (HDFS)
3. Make "External" Hive table from the file now in HDFS
4. [optional] Convert to Hive-optimized table format (RC or ORC)

5. [optional] Delete the original "External" table

1. Copy Data from Local to 30 Server

Upload file to the server via SCP -- this example uploads to my user folder. (Remember to execute this command from your local terminal, not on the server)

```
scp <local file> <user>@<server.domain>:/home/<user>/jonpaul
```

2. `put` Data into Hadoop File System

For External tables we will want to put the raw data (e.g. CSV) into a specific "data warehouse" folder in Hive. This gets the data into the HDFS so that later we can name which files from this folder we want use in Hive tables.

Once on the server, issue the following `-put` command to upload data to the Hadoop File System (HDFS).

`put` Syntax

```
hadoop fs -put <file on server to upload> <path in HDFS>
```

Example `put` Command

```
hadoop fs -put jptest_table.csv hdfs://prodserver/user/<server user>/hivetables/jp_tester/data_warehouse
```

Note, this example above assumes you are in the same directory on the server as the data file you want to add to the HDFS. Typical path specification applies if not.

Make Directory if Needed

If the target destination does not exist, create a folder for your upload with the traditional `mkdir` command.

```
hadoop fs -mkdir hdfs://prodserver/user/<server user>/hivetables/jp_tester/data_warehouse
```

Confirm Upload Worked by Listing Tables in the Target Directory:

```
hadoop fs -ls hdfs://prodserver/user/<server user>/hivetables/jp_tester/data_warehouse
```

```
Found 1 items
-rw-rw----+  3 <server user> hadoopprod 296157 2019-05-29 20:14
'hdfs://prodserver/user/<server
user>/hivetables/jp_tester/data_warehouse/jptest_table.csv'
```

3. Create an External Hive Table

Create a Schema to House Tables if Needed

Tables exist inside a schema/database. If one doesn't exist, or you want to make a new one for a specific project, we must create it with the following command.

```
CREATE SCHEMA IF NOT EXISTS jptest_schema;      --Create database (schema)
to house table
```

The *schema* is synonymous with *database* in Hive -- they contain tables. This is shown by the following commands being equivalent:

```
show schemas;
show databases;
```

Create the External Table

The External table in Hive is not optimized for storage or processing, but it *is* queryable. Once uploaded, you can extract data from it using SQL (HQL) commands. But since it is inefficiently stored in Hive, we might want to store the data in a Hive-optimized table ("RC" or "ORC" formats). To do so requires using this External table as a staging table in that process. Note that for a CSV, we must specify the values of

```
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
```

So even though it is a CSV, we are storing it as a "text" file. This is how Hive processes CSVs, TSVs, etc. I believe `TEXTFILE` is the default file format in Hive, so that if you do not specify `STORED AS`, Hive will default to this.

The `LOCATION` keyword tells where in Hive the newly created table will live.

```
CREATE EXTERNAL TABLE IF NOT EXISTS jptest_schema.jptest_table
(
    colA STRING,
    colB STRING,
    colC INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 'hdfs://prodserver/user/<server
user>/hivetables/jp_tester/jptest_table';
```

***Important* - A Note About the Location Paramater**

The `LOCATION` of a Hive table is very important. It can be thought of as a folder in the HDFS system which holds the data and the table schema itself. We create these two parts separately, as we have shown the creation of the schema above. Each folder can have **only one table** in it! This means that each table needs to be created in its **own `LOCATION` (folder)**.

Using the `LOCATION` in the query above, we see that the terminal directory has been named the same as the table itself. This doesn't have to be the case -- you can make the table (`jptest_table`) in any location. But since every table needs to have its own directory, it makes most sense to name the folder after the table it is housing.

```
LOCATION 'hdfs://prodserver/user/<server user>/hivetables/jp_tester/<table name>';
```

The core takeaway is: **one folder, one table**.

Backing up for a moment, when we add a CSV / data to Hive via the `put` command, the location we put it in determines what happens to it. If we put it in a directory that has no tables, it lives happily, like a peaceful child that makes good grades and cleans its room. Then when we later use it to fill a table we've created in Hive, the CSV gets moved to the directory where that table lives. In a sense, it is 'absorbed' by the Hive table. This is fine.

But there's a catch. If you add more than a single CSV file to a table's directory, Hive automatically appends this new CSV data to the existing table. Even if the table schema / structure do not match. Hive will add new columns if needed, or it will create full `NULL` columns if the additional CSV has fewer columns than the existing table.

In effect, whenever we make a table in Hive, Hive absorbs whatever data files are also in that `LOCATION` into the table. So, if we put five different CSVs, each of different table dimensions, into the same directory in Hive and then create a table in that location, without specifying any of the CSVs to use as the data for the newly created table, Hive will automatically fill the newly created table with all the data from each of the five CSVs.

It is because of this that I feel most comfortable with the approach I've laid out above: have a single `data_warehouse` folder to upload all CSVs to. From there, we will distribute them to the desired tables in their corresponding locations. This might not be the most efficient or accepted practice in Hive, but it's what I have seen work most reliably.

Fill the Table with Data

We have now created the skeletal table (schema) in Hive and need to fill it with data.
To do so we specify a file which will be used to populate our table. There are two ways to do this.

From Data Already in Hive

This requires the data file having been loaded into Hive via the `put` command, as shown above.

```
LOAD DATA INPATH 'hdfs://prodserver/user/<server
user>/hivetables/jp_tester/data_warehouse/jptest_table.csv'
INTO TABLE jptest_schema.jptest_table;
```

One important fact is that when we use the `LOAD DATA INPATH ... INTO TABLE` command, the data file (CSV) is **moved** from the starting directory into the target table's directory! So, when I load the `jptest_table.csv` file from the `data_warehouse` directory into the `jptest_table` *table*, the `jptest_table.csv` file will be moved in the process. This means the `data_warehouse` is only a pit stop for the tables and is not a permanent repository.

This behavior might be desirable or undesirable depending on the use case, but it is important to understand as it might not be the expected default behavior.

From a File on the Server -- [Currently Failing]

JP: I did not have success with this method -- it might work only for INTERNAL Hive tables, or it might not work on our setup at all. I spoke with our Data Architect about it and he said to just use the first method, "From Data Already in Hive", above. The advantage of this one if it worked is that you remove one step of having to `put` the data into HDFS first.

`hadoop fs -copyFromLocal /home/jonpaul/jptest_table.csv hdfs://prodserver/user/<server user>/hivetables/jp_tester/jptest_table`

A Note About Specifying Which Database to Use

To avoid having to name the database with every table associated with it, simply use the `use <database name>` command *once logged into Hive* to tell Hive that you want to operate only in a specific database:

```
USE jptest_schema;
```

Then you can simply name the tables desired without prepending the database name.

```
SELECT * FROM jptest_table;
```

A Note About Aliasing Columns and Tables

In Hive, it is **required** to use `AS` when aliasing for consistent results. Failure to alias with `AS` might lead to strange results.

Here is a query with newly created column ("MONTH") and a table ("SGT") that fails to use `AS` when aliasing, which will likely cause problems executing the query.

```
SELECT
    2020 MONTH,              --Will cause a problem because of missing AS
```

```
for alias
    NETWORK_CD,
    SUBSCRIBERS
FROM SOME_GREAT_TABLE SGT   --also missing AS
```

Here is the same query with the correct `AS` usage.

```
SELECT
    2020 AS MONTH,              -- Correctly uses AS
    NETWORK_CD,
    SUBSCRIBERS
FROM SOME_GREAT_TABLE AS SGT
```

Sometimes Hive will execute the query without the use of `AS`, but the resulting column will have potentially an incorrect header as well as data. The key point is that failure to use `AS` does not guarantee a failed query -- it can "silently fail" by successfully returning data; it will just be incorrect data in places.

A Note About Single Quotes

As currently constructed, we *must* use single quotes ("'") in queries with strings when using the `hive_pandas` module.

```
SELECT *
FROM YOUR_TABLE
WHERE NETWORK_CD = 'SC'  -- This is correct
```

```
SELECT *
FROM YOUR_TABLE
WHERE NETWORK_CD in ('SC', 'SB', 'SV')  -- This is correct
```

```
SELECT *
FROM YOUR_TABLE
WHERE NETWORK_CD = "SC"  -- This will fail, with error shown below
(usually)
```

Failure to do so may cause a failed query. This is keeping in line with Netezza and other database preferences.

One of the common errors you'll see if you have forgotten to use single quotes might be: `Invalid table alias or column reference '<variable or column name>'`

A Note About In-Line Comments

Currently, `hive_pandas` requires that no in-line comments be used. Any query submitted through `hive_pandas` that has a comment in it will cause a failure. This will hopefully be addressed in the future. But as of July 2019, all comments must be made outside of the queries themselves.

A Note About Create Table Syntax

```
CREATE EXTERNAL TABLE IF NOT EXISTS jptest_schema.jptest_table
```

The table name is specified by `<database>.<table_name>` syntax.
If the database doesn't exist, table creation will fail, hence why we `CREATE SCHEMA` prior to making the table. If you have specified which database you're going to be working in via the `use <database>` command (discussed above), then you can simply write the table name itself, without the database prefix.

A Note About `STRING` vs. `VARCHAR` in Hive

It is recommended to use the `STRING` data type in Hive when creating tables instead of the, perhaps more common, `VARCHAR` data type in SQL databases.

1. Posts on a forum showed that `EXTERNAL` (externally managed) tables pointing to text files showed no performance differences between `STRING`, `VARCHAR`, or `CHAR`.
2. No character limit for `STRING`, unlike `VARCHAR`. And, more important, if you do exceed the character limit in a `VARCHAR` column, Hive silently fails (does not notify you of this) by default. You will have incomplete data entry but not realize it. This could create a major issue.
3. In Hive versions older than 0.14 (HDP 2.1 and older) `VARCHAR` will *not* use vectorization with your ORC files, whereas `STRING` can use vectorization. Obviously, this can result in substantial differences in query performance.

A Note About GROUP BY and ORDER BY

...use only col numbers...

Verify Table Was Created

This command confirms that our table was created and has been filled with the assigned data.

```
SELECT * FROM jptest_schema.jptest_table LIMIT 50;
```

Create Table from Existing Table

This lets us make a new table from a `SELECT` statement on an existing table. If we `SELECT *` we will copy the entire existing table.

```
CREATE TABLE jptest_schema.new_table AS
SELECT *
FROM jptest_schema.jptest_table;
```

Another way, from Atanu:

```
CREATE EXTERNAL TABLE table2(attribute STRING)
STORED AS TEXTFILE
LOCATION 'HDFSDir';

INSERT OVERWRITE TABLE table2 SELECT * FROM table1;
```

Hive File Formats

Here is a good overview, with code, of primary Hive file/table formats. The purpose of using Hive-specific formats is optimized storage space and potentially performance. According to that article, both `RC` and `ORC` are highly optimized file formats, with a third type, `SEQUENCEFILE`, being used to reduce computational overhead of processing on many smaller tables.

The key issue with these Hive-optimized formats is that we cannot import data directly into them. First we must create a table, as shown in detail above, and then copy the data from the basic table into the optimized tables.

RC (Row Columnar) Table

From the article above:

> `RCFILE` stands of Record Columnar File which is another type of binary file format which offers high compression rate on the top of the rows.
>
> `RCFILE` is used when we want to perform operations on multiple rows at a time.
>
> `RCFILE`s are flat files consisting of binary key/value pairs, which shares many similarities with `SEQUENCEFILE`. `RCFILE` stores columns of a table in form of record in a columnar manner. It first partitions rows horizontally into row splits and then it vertically partitions each row split in a columnar way. `RCFILE` first stores the metadata of a row split, as the key part of a record, and all the data of a row split as the value part. This means that `RCFILE` encourages column oriented storage rather than row oriented storage.
>
> *This column oriented storage is very useful while performing analytics*. It is easy to perform analytics when we "hive" a column oriented storage type.
>
> Facebook uses `RCFILE` as its default file format for storing of data in their data warehouse as they perform different types of analytics using Hive.

Create RC Table

Same `CREATE TABLE` syntax we are used to, though do note the way I've named this example table: with a `_rc` suffix. I don't know if this is standard practice in Hive, but it might be a smart idea to indicate what type of table it is.

```
CREATE TABLE jptest_schema.jptable_rc
(
    col_1 STRING,
    col_2 STRING,
    col_3 INT
)
STORED AS RCFILE;
```

Fill the RC Table With the External Table's Data

Using `INSERT INTO` will append data, I believe, into the `RC` table.

```
INSERT INTO TABLE jptest_schema.jptable_rc
SELECT * FROM jptest_schema.jptest_table;
```

Another option is to use the `OVERWRITE` argument. I believe this overwrites the entire table.

```
INSERT OVERWRITE TABLE jptest_schema.jptable_rc
SELECT * FROM jptest_schema.jptest_table;
```

Delete Raw CSV Table in Hive (Since it is now in RCFILE format).

We can remove the un-optimized External table now that we've copied data from it into our `RC` table. This would be meant to free up space within Hive, which depending on the scale of the project and its data, might have real cost savings associated with it.

`hadoop fs -rm hdfs://prodserver/user/<server user>/hivetables/jp_tester/jptest_table/*`

Access Data From the RC Table

Querying this `RC` table works just the same as any table.

```
SELECT * FROM jptest_schema.jptable_rc;
```

ORC (Optimized Row Columnar) Table

From the article above:

> `ORC` stands for Optimized Row Columnar which means it can store data in an optimized way than the other file formats. `ORC` reduces the size of the original data up to 75% (eg: 100 GB file will become 25 GB). As a result the speed of data processing also increases. `ORC` shows better performance than `Text`, `Sequence` and `RC` file formats. [JP: I don't know how *much* better the performance is than, say, `RC`]
>
> An `ORC` file contains rows data in groups called Stripes along with a file footer. `ORC` format improves the performance when Hive is processing the data.

Create ORC Table

```
CREATE TABLE jptest_schema.jptable_orc
(
    col_1 STRING,
    col_2 STRING,
    col_3 INT
)
STORED AS ORC TBLPROPERTIES ('TRANSACTIONAL'='TRUE');
```

Fill the ORC Table With the External Table's Data

Using `INSERT INTO` will *append* data, I believe, into the `ORC` table.

```
INSERT INTO TABLE jptest_schema.jptable_orc
SELECT * FROM jptest_schema.jptest_table;
```

Another option is to use the `OVERWRITE` argument. I believe this overwrites the entire table and can be used to move data from a non-Transactional table to an ORC table in Hive (nice). See this short example

```
INSERT OVERWRITE TABLE jptest_schema.jptable_orc
SELECT * FROM jptest_schema.jptest_table;
```

Delete Raw CSV Table in Hive (Since it is now in ORC format).

We can remove the un-optimized External table now that we've copied data from it into our `ORC` table. This would be meant to free up space within Hive, which depending on the scale of the project and its data, might have real cost savings associated with it.

`hadoop fs -rm hdfs://prodserver/user/<server user>/hivetables/jp_tester/jptest_table/*`

Access Data From ORC table

Querying this `ORC` table works just the same as any table.

```
SELECT * FROM jptest_schema.jptable_orc;
```

`SEQUENCEFILE`

From the article above:

> We know that Hadoop's performance is drawn out when we work with a small number of files with big size rather than a large number of files with small size. If the size of a file is smaller than the typical block size in Hadoop, we consider it as a small file. Due to this, a number of metadata increases which will become an overhead to the NameNode. To solve this problem sequence files are introduced in Hadoop. Sequence files act as a container to store the small files.
>
> Sequence files are flat files consisting of binary key-value pairs. When Hive converts queries to MapReduce jobs, it decides on the appropriate key-value pairs to be used for a given record. Sequence files are in the binary format which can be split and the main use of these files is to club two or more smaller files and make them as a one sequence file.
>
> In Hive we can create a sequence file by specifying STORED AS `SEQUENCEFILE` in the end of a `CREATE TABLE` statement.
>
> There are three types of sequence files: • Uncompressed key/value records. • Record compressed key/value records – only 'values' are compressed here • Block compressed key/value records – both keys and values are collected in 'blocks' separately and compressed. The size of the 'block' is configurable.
>
> Hive has its own `SEQUENCEFILE` reader and `SEQUENCEFILE` writer libraries for reading and writing through sequence files.

General Hive / Hadoop Commands

List HDFS Folders

`hadoop fs -ls hdfs://prodserver/user/<server user>/hivetables/`

Create Folder

`hadoop fs -mkdir hdfs://prodserver/user/<server user>/hivetables/tmptesttable`

Upload File to HDFS

`hadoop fs -put <your file> hdfs://prodserver/user/<server user>/hivetables/tmptesttable`

Remove One File

```
hadoop fs -rm hdfs://prodserver/user/<server user>/hivetables/tmptesttable/<your file>
```

Remove All Files in a Folder

```
hadoop fs -rm hdfs://prodserver/user/<server user>/hivetables/tmptesttable/*
```

Remove Folder

```
[<server user>@<server> tmp]$ hadoop fs -rm -r hdfs://prodserver/user/<server user>/hivetables/tmptesttable
```

Other Operations

Transfer Table From Hive to Netezza

To transfer table from Hive to Netezza requires one more step, which is to create table in Netezza and use `nzload` command to upload data.

1. Using Hive CLI, download the table into CSV file in LH cluster
2. Create table in Netezza
   - The table definition (data type) will be slightly different between Hive and Netezza
3. In LH cluster, use Netezza CLI (`nzload`) to load data to Netezza.
   - Example: `nzload -df ${fileName} -db ${targetDb} -t ${targetTable} -delim '\t'`

A Direct `os` module command to SSH and Hive in One Line in Python (via Data Architect)

My Python implementation did not use this, but I'm keeping it as reference in case someone else's system works differently and might need a backup option. This would replace the need to use `paramiko`, however the `hive_pandas` module is written to parse the output from the `paramiko` object, so I am not sure if the `os` output below would be valid with it or not.

```python
import os
command = "ssh -t <server user>@<server.domain> 'hive -e \"use <dbname>; SELECT * FROM <table name>\"'"

output = os.popen(command).read()   ## Beware the memory explosion with large queries and .read()
print(output)
```

Reference Material

Links

Internal vs External Tables

ORC Guide

Hive in Python overview that doesn't apply to us b/c of Firewall:

LH Cluster

- `ssh <server user>@<server.domain>` --> LH Cluster
- Use this command to switch user to prod server acct: `sudo su - <server user>`
- `[<server user>@<server> ~]`: you see we are logged into the `<server user>` account on the server, not the `<server user>` account. We want to be on the `<dbname>` account.

Hive Server Info

"The Hive server is running on , on port 10001, in HTTP mode." DB Name: `<dbname>` host address: <server.domain> port: 10001

Examples of how to format the same query for Python

There are three examples below, starting with the least verbose and most familiar and ending with the most verbose.

Note we *must* use double quotes (`""`) in strings, such as `"SC"` instead of `'SC'` for a `network_cd`. **Failure to do so may cause a failed query.**

Preferred Method -- Most Succinct (no sub-query required)

This query requires `set hive.groupby.orderby.position.alias=true;` to utilize `GROUP BY` (and `ORDER BY`) via the column numbers instead of column names. Grouping by column names requires that the column already exist as-named in the table selected from. If you create a new column as part of the `SELECT` statement, even if it is a constant such as `201805` below, then Hive will give the following error: *Expression not in GROUP BY key ''*

Because of this, I've modified `hive_pandas.py` module to include `set hive.groupby.orderby.position.alias=true;` in every query it sends. This prevents having to rewrite every `GROUP BY` with a sub-query, which is both laborious *and* more error prone. This is also the most straight forward and commonly used SQL syntax, and is the most readable.

Note: It IS possible that more complex queries might not execute correctly with this approach in Hive... I have to test them out. For the following query, it works perfectly

```
SELECT
    201805 AS MONTH,
    CASE WHEN NETWORK_CD = "VR" THEN "VC"
        WHEN NETWORK_CD = "C" THEN "AP"
```

```
            ELSE NETWORK_CD
            END AS NETWORK_CD,
        COUNT(*) AS EOP
    FROM FS_MSISDN_201805
    WHERE NETWORK_CD NOT IN ("SA")
    GROUP BY 1,2
    ORDER BY 1,2;
```

Sub-query method that requires setting the GROUP BY alias option to True:

```
SELECT
    monthly.MONTH,
    monthly.NETWORK_CD,
    monthly.EOP
FROM (
    SELECT
        201805 as MONTH,
        CASE WHEN NETWORK_CD = "VR" THEN "VC"
            WHEN NETWORK_CD = "C" THEN "AP"
            ELSE NETWORK_CD
            END AS NETWORK_CD,
        COUNT(*) AS EOP
    FROM FS_MSISDN_201805
    WHERE NETWORK_CD NOT IN ("SA")
    GROUP BY 1,2
    ) as monthly
ORDER BY 1,2;
```

Sub-query method *not* reliant upon setting GROUP BY alias option to True:

```
SELECT
    monthly.MONTH,
    monthly.NETWORK_CD,
    monthly.EOP
FROM (
    SELECT
        raw_cts.MONTH,
        raw_cts.NETWORK_CD,
        COUNT(*) AS EOP
    FROM (
        SELECT
            201805 as MONTH,
            CASE WHEN NETWORK_CD = "VR" THEN "VC"
                WHEN NETWORK_CD = "C" THEN "AP"
                ELSE NETWORK_CD
                END AS NETWORK_CD
            FROM FS_MSISDN_201805
            WHERE NETWORK_CD NOT IN ("SA")
```

```
                ) as raw_cts
        GROUP BY MONTH, NETWORK_CD
        ) as monthly
    ORDER BY MONTH, NETWORK_CD;
```

How to set Hive variables, from this Stack Overflow article.

Use this to trim col names to only column and ignore table name prefix (bug test with hive_pandas.py): `set hive.resultset.use.unique.column.names=false;`

Use this in the create table statement to skip header rows when creating tables from CSV: `tblproperties ("skip.header.line.count"="1");` (here showing skip only first row)

Run the HQL metrics generation files on the server with this command to control variables.

Example:
`hive -hiveconf TIME_PERIOD=201205 -hiveconf PREV_MONTH=201204 -hiveconf SCHEMA=<dbname> -f <script name>.hql`

See if a table is MANAGED TABLE (can use DELETE and UPDATE commands)

`describe formatted <table name>;`

Change Column Name and/or Data Types: `ALTER TABLE table_name CHANGE old_col_name new_col_name new_data_type`

Can keep same name and just change the data type: `ALTER TABLE <TABLE NAME> CHANGE MARKET_CD MARKET_CD STRING`