HBaseBulkLoad

Hive HBase Bulk Load

This page explains how to use Hive to bulk load data into a new (empty) HBase table per HIVE-1295. (If you're not using a build which contains this functionality yet, you'll need to build from source and make sure this patch and HIVE-1321 are both applied.)

Overview

Ideally, bulk load from Hive into HBase would be part of HBaseIntegration, making it as simple as this:

```
CREATE TABLE new_hbase_table(rowkey string, x int, y int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf:x,cf:y");

SET hive.hbase.bulk=true;

INSERT OVERWRITE TABLE new_hbase_table
SELECT rowkey_expression, x, y FROM ...any_hive_query...;
```

However, things aren't *quite* as straightforward as that yet. Instead, a procedure involving a series of SQL commands is required. It should still be a lot easier and more flexible than writing your own map/reduce program, and over time we hope to enhance Hive to move closer to the ideal.

The procedure is based on underlying HBase recommendations, and involves the following steps:

1. Decide how you want the data to look once it has been loaded into HBase.
2. Decide on the number of reducers you're planning to use for parallelizing the sorting and HFile creation. This depends on the size of your data as well as cluster resources available.
3. Run Hive sampling commands which will create a file containing "splitter" keys which will be used for range-partitioning the data during sort.
4. Prepare a staging location in HDFS where the HFiles will be generated.
5. Run Hive commands which will execute the sort and generate the HFiles.
6. (Optional: if HBase and Hive are running in different clusters, distcp the generated files from the Hive cluster to the HBase cluster.)
7. Run HBase script `loadtable.rb` to move the files into a new HBase table.
8. (Optional: register the HBase table as an external table in Hive so you can access it from there.)

The rest of this page explains each step in greater detail.

Decide on Target HBase Schema

Currently there are a number of constraints here:

- The target table must be new (you can't bulk load into an existing table)
- The target table can only have a single column family (HBASE-1861)
- The target table cannot be sparse (every row will have the same set of columns); this should be easy to fix by either allowing a MAP value to be read from Hive, and/or by allowing rows to be read from Hive in pivoted form (one row per HBase cell)

Besides dealing with these constraints, probably the most important work here is deciding on how you want to assign an HBase row key to each row coming from Hive. To avoid inconsistencies between lexical and binary comparators, it is simplest to design a string row key and use it consistently all the way through. If you want to combine multiple columns into the key, use Hive's string concat expression for this purpose. You can use CREATE VIEW to tack on your rowkey logically without having to update any existing data in Hive.

Estimate Resources Needed

TBD: provide some example numbers based on Facebook experiments; also reference Hadoop Terasort

Add necessary JARs

You will need to add a couple jar files to your path. First, put them in DFS:

```
hadoop dfs -put /usr/lib/hive/lib/hbase-VERSION.jar /user/hive/hbase-VERSION.jar
hadoop dfs -put /usr/lib/hive/lib/hive-hbase-handler-VERSION.jar /user/hive/hive-hbase-handler-VERSION.jar
```

Then add them to your hive-site.xml:

```
<property>
  <name>hive.aux.jars.path</name>
  <value>/user/hive/hbase-VERSION.jar,/user/hive/hive-hbase-handler-VERSION.jar</value>
</property>
```

Prepare Range Partitioning

In order to perform a parallel sort on the data, we need to range-partition it. The idea is to divide the space of row keys up into nearly equal-sized ranges, one per reducer which will be used in the parallel sort. The details will vary according to your source data, and you may need to run a number of exploratory Hive queries in order to come up with a good enough set of ranges. Here's one example:

```
add jar lib/hive-contrib-0.7.0.jar;
set mapred.reduce.tasks=1;
create temporary function row_sequence as
'org.apache.hadoop.hive.contrib.udf.UDFRowSequence';
select transaction_id from
(select transaction_id
from transactions
tablesample(bucket 1 out of 10000 on transaction_id) s
order by transaction_id
limit 10000000) x
where (row_sequence() % 910000)=0
order by transaction_id
limit 11;
```

This works by ordering all of the rows in a .01% sample of the table (using a single reducer), and then selecting every nth row (here n=910000). The value of n is chosen by dividing the total number of rows in the sample by the desired number of ranges, e.g. 12 in this case (one more than the number of partitioning keys produced by the LIMIT clause). The assumption here is that the distribution in the sample matches the overall distribution in the table; if this is not the case, the resulting partition keys will lead to skew in the parallel sort.

Once you have your sampling query defined, the next step is to save its results to a properly formatted file which will be used in a subsequent step. To do this, run commands like the following:

```
create external table hb_range_keys(transaction_id_range_start string)
row format serde
'org.apache.hadoop.hive.serde2.binarysortable.BinarySortableSerDe'
stored as
inputformat
'org.apache.hadoop.mapred.TextInputFormat'
outputformat
'org.apache.hadoop.hive.ql.io.HiveNullValueSequenceFileOutputFormat'
location '/tmp/hb_range_keys';

insert overwrite table hb_range_keys
select transaction_id from
(select transaction_id
from transactions
tablesample(bucket 1 out of 10000 on transaction_id) s
order by transaction_id
limit 10000000) x
where (row_sequence() % 910000)=0
order by transaction_id
limit 11;
```

The first command creates an external table defining the format of the file to be created; be sure to set the serde and inputformat/outputformat exactly as specified.

The second command populates it (using the sampling query previously defined). Usage of ORDER BY guarantees that a single file will be produced in directory `/tmp/hb_range_keys`. The filename is unknown, but it is necessary to reference the file by name later, so run a command such as the following to copy it to a specific name:

```
dfs -cp /tmp/hb_range_keys/* /tmp/hb_range_key_list;
```

Prepare Staging Location

The sort is going to produce a lot of data, so make sure you have sufficient space in your HDFS cluster, and choose the location where the files will be staged. We'll use `/tmp/hbsort` in this example.

The directory does not actually need to exist (it will be automatically created in the next step), but if it does exist, it should be empty.

```
dfs -rmr /tmp/hbsort;
dfs -mkdir /tmp/hbsort;
```

Sort Data

Now comes the big step: running a sort over all of the data to be bulk loaded. Make sure that your Hive instance has the HBase jars available on its auxpath.

```
set hive.execution.engine=mr;
set mapred.reduce.tasks=12;
set hive.mapred.partitioner=org.apache.hadoop.mapred.lib.TotalOrderPartitioner;
set total.order.partitioner.path=/tmp/hb_range_key_list;
set hfile.compression=gz;

create table hbsort(transaction_id string, user_name string, amount double, ...)
stored as
INPUTFORMAT 'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.hbase.HiveHFileOutputFormat'
TBLPROPERTIES ('hfile.family.path' = '/tmp/hbsort/cf');

insert overwrite table hbsort
select transaction_id, user_name, amount, ...
from transactions
cluster by transaction_id;
```

The CREATE TABLE creates a dummy table which controls how the output of the sort is written. Note that it uses `HiveHFileOutputFormat` to do this, with the table property `hfile.family.path` used to control the destination directory for the output. Again, be sure to set the inputformat/outputformat exactly as specified. In the example above, we select gzip (gz) compression for the result files; if you don't set the `hfile.compression` parameter, no compression will be performed. (The other method available is lzo, which compresses less aggressively but does not require as much CPU power.)

Note that the number of reduce tasks is one more than the number of partitions - this **must** be true or else you will get a "Wrong number of partitions in keyset" error.

There is a parameter `hbase.hregion.max.filesize` (default 256MB) which affects how HFiles are generated. If the amount of data (pre-compression) produced by a reducer exceeds this limit, more than one HFile will be generated for that reducer. This will lead to unbalanced region files. This will not cause any correctness problems, but if you want to get balanced region files, either use more reducers or set this parameter to a larger value. Note that when compression is enabled, you may see multiple files generated whose sizes are well below the limit; this is because the overflow check is done pre-compression.

The `cf` in the path specifies the name of the column family which will be created in HBase, so the directory name you choose here is important. (Note that we're not actually using an HBase table here; `HiveHFileOutputFormat` writes directly to files.)

The CLUSTER BY clause provides the keys to be used by the partitioner; be sure that it matches the range partitioning that you came up with in the earlier step.

The first column in the SELECT list is interpreted as the rowkey; subsequent columns become cell values (all in a single column family, so their column names are important).

Run HBase Script

Once the sort job completes successfully, one final step is required for importing the result files into HBase. Again, we don't know the name of the file, so we copy it over:

```
dfs -copyToLocal /tmp/hbsort/cf/* /tmp/hbout
```

If Hive and HBase are running in different clusters, use distcp to copy the files from one to the other.

If you are using HBase 0.90.2 or newer, you can use the completebulkload utility to load the data into HBase

```
hadoop jar hbase-VERSION.jar completebulkload [-c /path/to/hbase/config/hbase-site.xml] /tmp/hbout transactions
```

In older versions of HBase, use the `bin/loadtable.rb` script to import them:

```
hbase org.jruby.Main loadtable.rb transactions /tmp/hbout
```

The first argument (`transactions`) specifies the name of the new HBase table. For the second argument, pass the staging directory name, not the the column family child directory.

After this script finishes, you may need to wait a minute or two for the new table to be picked up by the HBase meta scanner. Use the hbase shell to verify that the new table was created correctly, and do some sanity queries to locate individual cells and make sure they can be found.

Map New Table Back Into Hive

Finally, if you'd like to access the HBase table you just created via Hive:

```
CREATE EXTERNAL TABLE hbase_transactions(transaction_id string, user_name string, amount double, ...)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf:user_name,cf:amount,...")
TBLPROPERTIES("hbase.table.name" = "transactions");
```

Followups Needed

- Support sparse tables
- Support loading binary data representations once HIVE-1245 is fixed
- Support assignment of timestamps
- Provide control over file parameters such as compression
- Support multiple column families once HBASE-1861 is implemented
- Support loading into existing tables once HBASE-1923 is implemented
- Wrap it all up into the ideal single-INSERT-with-auto-sampling job...