Lab 11 – HBase

CC5212-1

August 11, 2015

We previously played around with Cassandra, which is a column-family (aka. tabular) store. Now we will play around with another column-family store: HBase. What are the differences between the two?

Well they both have different origins. HBase is essentially a faithful reproduction of Google's BigTable and tends towards write consistency, (loosely speaking, CP in the CAP theorem). On the other hand, Cassandra borrowed a lot of ideas (and experience) from Amazon's Dynamo, extending it to do more of what BigTable offers. In terms of consistency, Cassandra originally leaned towards AP in the CAP theorem, and being available for writes, but later added support to let users choose the level of consistency they wanted for specific operations (for example, you can choose to read from ONE, from ALL, from a QUOROM).

Likewise, HBase was originally designed to be tightly coupled with HDFS/Hadoop, meaning that one could read/write to/from HBase tables (stored on HDFS) using Hadoop. For example one can analyse a huge HBase table with Hadoop, or join tables, or transform one table to another. Though we won't go into detail on this quite now, I'll link to further material for reading at the end. But as Cassandra has developed, it too has added support for Hadoop, Pig, etc. Traditionally HBase still has some performance benefits for scans (reading the whole table) while Cassandra has some benefits for lookups.

But to be honest, both stores have changed a lot over the years, where Cassandra has added the features that HBase had previously (e.g., integration with Hadoop, optional stronger consistency guarantees, etc.) and HBase has added features that Cassandra originally supported (e.g., namespaces/keyspaces), etc. So the answer is that I'm not really sure what to tell you about the differences between them or when you might want to use one versus the other. It seems different people have different opinions, with fan-boys(/-girls) on either side and clear arguments hard to discern. The most general answer seems to be "it depends".

In any case, in this class we'll go through a similar lab to the previous one, but for HBase; this will give you a feel for HBase and illustrate some superficial differences between Cassandra and HBase. You again just need an SSH client onto the cluster (username `uhadoop`, password in the forum).[1]

- First we are going to run the HBase shell, which is similar to the CQL shell for Cassandra:
  ```
  cd /data/hadoop/hbase
  ./bin/hbase shell
  ```
- Second, let's get oriented
  - Who are you?
    `whoami`
    You can configure users and user privileges for interacting with different namespaces/tables.
  - Is the HBase instance up and running?
    `status`
    Hopefully everything looks okay when you run it.
  - So what can we do in here?
    `help`
    ... gives a raw list of commands.

[1] For those who want to read more about HBase, see `http://hbase.apache.org/book.html`

- Third, let's see how to create/alter/remove namespaces (aka. key-spaces in Cassandra) and tables. As before, please replace my details with your own.
  - Let's create a namespace and list the namespaces available:
    `create_namespace 'ahogan'`
    `list_namespace`
  - Namespaces are associated with some options. Likewise user privileges can be set at the namespace level. Let's see what options are set right now and change a setting to say the namespace can contain a maximum of one table:
    `describe_namespace 'ahogan'`
    `alter_namespace 'ahogan', {METHOD => 'set', 'hbase.namespace.quota.maxtables'=>'1'}`
    `describe_namespace 'ahogan'`
  - Let's create a table under that namespace:
    `create 'ahogan:sometable'`
    Didn't work ... need at least one column family (a group of related columns). Here we create a table with two column families called (imaginatively) `f1` and `f2`:
    `create 'ahogan:sometable', 'f1', 'f2'`
  - We can set options on a column family level. Above we just set some defaults; let's see:
    `describe 'ahogan:sometable'`
    Okay, let's change the table to keep the last three versions of values in the `f1` column family, a time-to-live of 36000 on the second column family, and to add a third (new) column family:
    `alter 'ahogan:sometable', {NAME => 'f1', VERSIONS => 5}, {NAME => 'f2', TTL => 36000}, 'f3'`
    `describe 'ahogan:sometable'`
    If we want, we can create the table with these settings directly. Let's try create a second table:
    `create 'ahogan:sometable2', {NAME => 'f1', VERSIONS => 5, TTL => 1000, COMPRESSION => 'GZ'}`
    `describe 'ahogan:sometable2'`
    But wait, something is up. Earlier we set the namespace to have a maximum of one table and now we have two! (Despite checking through the documentation, I don't know myself why it allows this. Not so important perhaps.)
- Okay, so let's remove the tables and the namespace you just created:
  - First let's list all tables: `list`.
  - Let's try drop that pesky second table that should not exist:
    `drop 'ahogan:sometable2'`
    Hmm ... didn't work. Before we remove a table, we have to disable reads to it first:
    `disable 'ahogan:sometable2'`
    `drop 'ahogan:sometable2'`
    Success.
  - Now let's try drop the namespace:
    `drop_namespace 'ahogan'`
    Doesn't work. You can't drop a namespace while there's a table in it. Drop the remaining table and then drop the namespace.
- Okay, so now we move to using the common namespace.
  - Let's see how the table is configured: `describe 'students:cc5212'` We see two column families: one for `bio` (biographical details), another for `likes` (interests). Note the version numbers supported: `bio` supports three versions, `likes` supports one.

  - Let's see what's already in there; maybe you'll be the first besides me to see the table:
    `scan 'students:cc5212'`
  - Now start filling in some details following my example. First we start by adding a row with your name:
    `put 'students:cc5212', 'ahogan', 'name', 'A. Hogan'`
    Ah, but it doesn't work if you don't add the column family to the column. Let's try again:
    `put 'students:cc5212', 'ahogan', 'bio:name', 'A. Hogan'`
    The row key (not named) is `ahogan`. The command says add the value `A. Hogan` to the row `ahogan` under column `name` in the `bio` namespace. Unfortunately, to the best of my knowledge, you have to update each cell individually. Let's add an age:
    `put 'students:cc5212', 'ahogan', 'bio:age', 30`
    Scan the table again to see your row's data.
  - Recall that there's three versions supported in the `bio` column family. To help see that in action later, overwrite your previous initialed name with your full name:
    `put 'students:cc5212', 'ahogan', 'bio:name', 'Aidan Hogan'`
    Later we'll see how to recall multiple versions for a cell.
  - Fill in the rest of your details following the columns used for me.

- Okay, so now we'll try some lookups:
  - First we can get a count of the rows in the table:
    `count 'students:cc5212'`
  - Next we can get all the cells for a given row:
    `get 'students:cc5212', 'ahogan'`
  - We can get the value for a specific column on a specific row:
    `get 'students:cc5212', 'ahogan', 'bio:name'`
  - We can also get, for example, multiple versions of a cell:
    `get 'students:cc5212', 'ahogan', {COLUMN=>'bio:name',VERSIONS=>2}`
  - We can also get multiple cells from a row at the same time:
    `get 'students:cc5212', 'ahogan', 'bio:name', 'bio:age'`
  - HBase is a sorted map: keys are stored in lexicographical order. For this reason, we can also do efficient range lookups on keys; for example, to get all rows from a–g, we can do:
    `scan 'students:cc5212', STARTROW => 'a', ENDROW => 'h'`
    The ranges are exclusive.

- In the previous lookups, we saw that the row key must be given. This is similar to Cassandra. However, in Cassandra, the CQL language supported syntax for adding secondary indexes. The HBase shell does not support this directly (to the best of my knowledge at least). Instead, you would need to manually create secondary indexes using a MapReduce job, for example.

The HBase shell is really just a convenience for playing around with the instance. Serious heavy lifting would have to be done programatically, typically through MapReduce, where HBase tables are just another input/output format for Hadoop. Again you can find some details and examples here: `http://hbase.apache.org/book.html#_hbase_as_a_mapreduce_job_data_source_and_data_sink` (and the sections below).

In any case, thanks for having a look at the lab, good luck with your exams!