UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Big Data Management (BDMA & MIRI Masters)
Session: HBase (Training)

Alberto Abelló & Sergi Nadal

The purpose of this document is that you get familiar with the environment and tools required for the lab session. This is an optional task, thus no delivery is expected and no grade will be assigned. This document is far from being a fully-fledged tutorial, we refer you to the online manuals of HBase (`https://hbase.apache.org/book.html`) for an in-depth guide.

1 Virtual machine

All tools you need are provided in the BDM virtual machine, accessible via *Virtech* or available to download. Please, refer to the provided manual for details on how to set up the virtual machine.

2 HBase basics

First, start the database (do not forget Hadoop needs to be running) using the provided script:

```
~/BDM_Software/hadoop/sbin/start-dfs.sh
~/BDM_Software/hbase/bin/start-hbase.sh
```

Startup a browser and head to `http://HOST:16010` to make sure HBase is properly running (replace the value `HOST` for the IP assigned to your Virtual Machine). Now you can startup a shell to interact with the database:

```
~/BDM_Software/hbase/bin/hbase shell
```

Now, let's create a table with two families, named *cf_1* and *cf_2*.

```
create 'table', 'cf_1', 'cf_2'
```

Then perform the following steps and answer the questions for each:

1. Perform a description on that table.

   ```
   describe 'table'
   ```

   - **What information is displayed there? How many versions are set for each family? What compression for each family? What blocksize for each family?**

2. Delete the table and recreate it again but this time we want the family *cf_1* to hold up to two different versions.

   ```
   disable 'table'
   drop 'table'
   create 'table', { NAME => 'cf_1', VERSIONS => 2 }, 'cf_2'
   ```

   Now, let's make our first inserts and scan.

```
put 'table', 'row_1', 'cf_1', 'first'
put 'table', 'row_1', 'cf_1:quali', 'second'
scan 'table'
```

- **Where is the qualifier of the first put?**

3. Try to insert the following.

```
put 'table', 'row_2', 'cf_3', 'third'
```

- **Is it possible? Why?**

4. Let's make some more inserts at once, mixing different rows, families and qualifiers and then scan the whole table.

```
put 'table', 'row_2', 'cf_1:qua', 'fourth'
put 'table', 'row_2', 'cf_2:qualifier', 'fifth'
put 'table', 'row_2', 'cf_2:qualif', 'fifth'
scan 'table'
```

5. The next code will run an insertion and a scan on a triple [row, family, qualifier] that already exists.

```
put 'table', 'row_1', 'cf_1', 'sixth'
scan 'table'
```

- **What happened with the old value? Why?**

As mere note, there is a cleaner way to retrieve only one row data from HBase, so scanning the whole table is avoided. This is the get command. For instance:

```
get 'table', 'row_1'
```

This will return all qualifiers for row with ID row_1, whereas:

```
get 'table', 'row_1', 'cf_1'
```

Will return all qualifiers for row with ID row_1 and family cf_1, whereas:

```
get 'table', 'row_1', 'cf_1:'
```

Will return all qualifiers for row with ID row_1, family cf_1 and where the qualifier is empty.

6. Retrieve back the value we overwrote after the last insertion.

```
get 'table', 'row_1', { COLUMN => 'cf_1:', VERSIONS => 2 }
```

- **What is that parameter VERSIONS appearing in the command?**

7. Insert another value in the same triple [row, family, qualifier] and query for it by retrieving three versions this time.

```
put 'table', 'row_1', 'cf_1', 'seventh'
get 'table', 'row_1', { COLUMN => 'cf_1:', VERSIONS => 3 }
```

- **Is the first value still showing? Why?**

You can type "help" to see all the commands in the shell.

3 On the key design

One important thing in HBase is to decide what or how the row keys are going to be defined. Scans, for instance, can benefit from the B+ tree and the clustered index to only read data of interest and to avoid wasting resources on non-relevant data for the current query. As a simple example, imagine a query that uses a specific attribute to filter out the rows quite often and thus we decide row keys have the value of such attribute at the beginning. Queries could then benefit from the lexicographical order from querying through row key prefix.
Now discuss the importance of the key design when writing in HBase. To do so, try to think about using the timestamp at which the insertion happens as row key.

- **Do you think this is a good design?**
- **Justify your answer and, if you think it is not, propose a solution.**

4 Interacting with HBase in Java

In the course's Moodle you will find enclosed a Java project. This project contains a random data generator (using the database schema presented in the assignment) that inserts data to HBase, as well as querying functionalities.

4.1 Project structure

Once you have fetched the code, it can be imported as a *Maven Project* into eclipse. In the *Main* class (in package *bdm.labs.hbase*) you can see that different arguments are expected, and will drive the behavior of the program.

Important note. You will need to change the value `HOST/IP` to the host assigned to your VM in the classes `MyHBaseWriter.java` and `MyHBaseReader.java`. Look for the lines where the property `hbase.zookeeper.quorum` is set.

4.2 Writing data

First, take a look at class `MyHBaseWriter.java`. This class deals with the generation of random data. We mainly rely on the following data structures:

- `int key`, which contains the value (an int) to be used as row key. Of course, this value is expected to be updated from other functions as we will see.
- `HashMap<String,String> data`, which holds a structure where keys will be attribute names and values data for this attributes for a given row. After completely inserting a tuple of data this structure should be updated as we will see.

The class contains the following methods:

- `open`: it creates the connection to HBase and sets a pointer to the target table;
- `nextKey`: it returns the key to be used when inserting a new tuple;
- `toFamily`: given an attribute name, it returns the family name to write the data of this attribute to (in this case it always returns a single family *all*);
- `put`: given an instance of `AdultInfo` (with random data), it populates the `data` map;
- `reset`: it empties the map `data`;

- `flush`: adds as many new rows (using the value of `key` as row key) as pairs of key-value contains the `data` map using the column family given by `toFamily` Here, we update the value of `key` and call `reset`;
- `close`: it closes the connection.

4.3 Reading data

Now take a look at class `MyHBaseReader.java`. This class performs a data scan on an HBase table, which outputs in the terminal. The class contains the following methods:

- `scanStart`: that returns the starting row key to be used when scanning;
- `scanStop`: that returns the ending row key to be used when scanning;
- `scanFamilies`: that returns an `Array` specifying the column families to print;
- `open`: it creates the connection, points it to the target table and defines the `Scan` object (parametrized using the start row key, end row key and families as given by the previous methods);
- `next`: it returns the next row as a `String`.

4.4 Executing the program

The *Main* class expects arguments. As we previously did, the simplest way to define such arguments is by creating a *Run Configuration* of type *Java Application* from the *Run* menu. In the tab menu called *Arguments* you will be able to edit them.

4.4.1 Writing arguments

The expected arguments to write random data to HBase are the following:

1. `-write`, which launches the random data generation process;
2. `-hbase-all`, which sets the program to use the configuration writing data to the `all` column family (in the session you will use different ones);
3. `N`, specifying the number of tuples to write;
4. `tablename`, specifying the target table name.

4.4.2 Reading arguments

The expected arguments to read data from HBase are the following:

1. `-read`, which indicates the program to read data;
2. `-hbase-all`, which sets the program to use the configuration writing data to the `all` column family (in the session you will use different ones);
3. `tablename`, specifying the target table name.