UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
School of Professional & Executive Development

Big Data Management and Analytics
Session: Distributed Storage: Key-Value (HBase) I

Lecturers: Petar Jovanovic and Josep Berbegal

1 Tasks To Do Before The Session

It is important that you: (1) carefully read the instruction sheet for this lab session, (2) introduce yourself to the lab's main objectives, (3) understand the theoretical background, and (4) get familiar with the tools being used.

2 Part A: Objectives & Questions (15min)

In the first 15 minutes, we will first clarify the main objectives of this lab. We will then learn about the role of HBase in the Big Data stack and introduce its main components. We will then overview the internals of HBase and learn how the data are accessed.

3 Part B: In-class Practice (2h 45min)

3.1 Exercise 1 (1h): Setting up HBase

Follow the document enclosed to this exercise sheet to set up HBase cluster, compile the new Java code as you did in previous sessions, make a JAR out of it and upload it to the master node.

Note: *Inside pom.xml of the Java project, you must configure the absolute path of toolsjar property to the location of the lib folder of Java JDK on your PC.*

3.2 Exercise 2 (45min): On the HBase basics

Once your HBase cluster is up, open a shell:

```
hbase-1.3.1/bin/hbase shell
```

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

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

Then do the following exercises.

1. Perform a description on that table. What information is displayed there? How many versions are set for each family? What compression is set for each family? What is the blocksize for each family?

```
describe 'table'
```

Answer: Information regarding the table and details about its corresponding families.
The version set is 1

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'
```

3. Let's make our first inserts and scan. Where is the qualifier of the first put?

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

Answer: In the first put the qualifier empty.

4. Try to insert the following. Is it possible? Say why.

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

Answer: No, it isn't.
ERROR: Unknown column family! Valid column names: cf_1:* cf_2:*

5. Let's make some more inserts at once, mixing different rows, families and qualifiers and then scan the whole table. At what level is the real schemaless property of HBase?

```
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'
```

Answer: It is at qualifier level.

6. Next command to run is an insertion and a scan on a triple [row, family, qualifier] that already exists. What happened with the old value? Why?

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

Answer: When doing the command scan 'table', the old value 'first' is not displayed. Because it only displays the latest inserted value, if the version isn't specified.

Note that 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.

Command to retrieve all the key-values for row row_1:

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

Command to retrieve all the key-values for row row_1 and family cf_1:

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

Command to retrieve all the key-values for row row_1, family cf_1 and where the qualifier is empty.

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

7. Retrieve back the value we overwrote after the last insertion. What is that parameter VERSIONS appearing in the command?

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

Answer: VERSIONS is the number of versions we want to display for the key-value we retrieved.

8. Insert another value in the same triple [row, family, qualifier] and query for it by retrieving three versions this time. Is the first value still showing? Why?

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

Answer: No, because we defined this family to hold two versions, consequently when a third value is inserted, the oldest one is overwritten.

9. Finally, we are going to insert few more rows randomly and then scan the whole table once again. Look at the row order of the result. Why do you think the order of appearance is that one? Feel free to insert more rows if you need them to double check your answer.

```
put 'table', 'row_10', 'cf_1', 'eigth'
put 'table', 'row_AA', 'cf_1', 'ninth'
scan 'table'
```

Answer: Because rows are lexicographically ordered on the row key.

3.3 Exercise 3 (1h): HBase - data access

Now, we will see how the stored data can be accessed in HBase.

In the following exercises, we are going to create a new table in HBase and populate it with some data to experiment on more advanced data access features. To do so, we will need the data generator you should have compiled at the beginning of the session. Then, create a new table called wines.100000 with only one family called 'all'. Thus, in HBase shell, run the following:

```
create 'wines.100000', 'all'
```

The next step is to actually populate the wines.100000 table. Load 100000 rows into it. You can do that by running (now in the Linux shell, not in the HBase one) the next command.

```
hadoop-2.7.4/bin/hadoop jar labo2.jar write -hbase -instances 100000 wines.100000
```

1. **Accessing a single row with row key.** To access a single row with row key, we need to provide, *table name*, *row key*, and *family name*. For example, let's access a row with row key '1234' from the previously created table.

   ```
   get 'wines.100000', '1234', 'all'
   ```

   Execute the given command five times and record the average response time:

2. **Accessing a single value with row key.** To access a single value with row key, we need to provide, *table name*, *row key*, *family name*, and *column qualifier*. For example, let's access a value of the column 'col', from the row with row key '1234' from the previously created table.

   ```
   get 'wines.100000', '1234', 'all:col'
   ```

   Execute the given command five times and record the average response time:

3. **Accessing a row with value filtering.** To access a row with value filtering, we use the *SingleColumnValueFilter* function. We need to provide, *table name*, *family name*, and *column qualifier*, *comparison operator*, and a *value*. For example, let's access a row with column 'alc' (alcohol degree) equal to 12.368596, from the previously created table.

   ```
   scan 'wines.100000', {FILTER=>
          "SingleColumnValueFilter('all','alc',=,'binary:12.368596')"}
   ```

   Execute the given command five times and record the average response time:

4. **Accessing a range of rows with row keys.** Let's access a range of rows from the previously created table.

   Given the *scan* command below, how many rows do you expect to be retrieved?

   Now, execute the command (record the response time):

   ```
   scan 'wines.100000', {STARTROW => '5', ENDROW => '6'}
   ```

   How many rows are actually retrieved?

   Does it coincide with what you previously guessed? How do you think the given command defines the range of rows?

   Answer:

   Execute the above command four more times and record the average response time for five executions:

5. **Access time.** Analyze the average response times above, and answer the following questions.

   - Compare the average times for accessing a single row with row key and with value filtering. Are these two times different? How much? Why?

     Answer:

   - Compare the average times for accessing a single row and range of rows with row keys. Are these times proportional to the amount of rows retrieved in both cases. If not, explain why?

     Answer:

Additional comments: