UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
School of Professional & Executive Development

# Big Data Management and Analytics
# Session: Column-Family Key-Value Stores. HBase

Lecturers: Petar Jovanovic and Sergi Nadal

November 4th, 2016

## 1 Required Tools

- SSH and SCP client
- `eclipse` IDE with JDK 7 and Maven plugin installed

## 2 Tasks To Do Before The Session

Before coming to this HBase session, you should have checked the session slides and write down all doubts you might have about them.

## 3 Part A: Examples & Questions (30min)

The first 30 minutes of the session will be spent on answering all the possible questions you might have come up with during the week by preparing the tasks before the session (see above).

## 4 Part B: In-class Practice (2h 30min)

### 4.1 Exercise 1 (30min): Setting up the cluster

Follow the document enclosed to this exercise sheet to start up an 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. **Let the lecturer know when this is finished.**

### 4.2 Exercise 2 (30min): On the HBase basics

Once your HBase cluster is up, open a shell:

```
hbase-1.1.2/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 for each family? What blocksize for each family?

   ```
   describe 'table'
   ```

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'
   ```

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

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

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'
   ```

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'
   ```

   *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 the key-values for row row_1, whereas:*

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

*Will return all the key-values for row row_1 and family cf_1, whereas:*

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

*Will return all the key-values for row row_1, family cf_1 and where the qualifier is empty.*

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 }
```

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 }
```

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'
```

### 4.3 Exercise 3 (45min): On the HBase balancing

In the following exercises, we are going to populate the HBase with little more data and then we will experiment on more advanced 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* with only one family called *all*. Thus, in a HBase shell, run the following:

```
create 'wines', 'all'
```

The next step is to actually populate the wines table. Bulk 150 MB into it (it takes around 4 min). You can do that by running (now in the Linux shell, not in the HBase one) the next command.

```
hadoop-2.5.1/bin/hadoop jar labo2.jar write -hbase -size 0.15 wines
```

Answer the following questions (*labo2.jar* is the JAR built for this session).

1. Explore a bit what is inside the */hbase/data/default/wines* folder in HDFS and its subfolders. Forget about metadata folders in there (e.g., *.tabledesc*, *.tmp*, *.regioninfo*, etc.). Can you relate each subfolder with one of the components from the HBase logical structure? List these relationships.

   ```
   hadoop-2.5.1/bin/hdfs dfs -ls /hbase/data/default/wines
   ```

2. To check the size of a given table in HBase, we need to do that through the HDFS, which is the file system storing all the HBase data. Then, run the following command. What is the size of this table? Does this make sense to you considering that we only inserted 150 MB of data? Discuss this result.

   ```
   hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines
   ```

3. On the web UI you might find other information in a user-friendlier manner. For instance, how many regions were created for the table wines? In what RegionServers are they stored?

4. Then check the size of each region by means of the next command. How much is it? Do you think they are well balanced? Compare your results with the classmate next to you.

   ```
   hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines/*
   ```

### 4.4 Exercise 4 (1h): On HBase balancing improvement

Let us try to make a better balancing. In order to do that, HBase provides DBAs with presplitting, which is a technique to split the table before insertions occur. This way, in the short term, HBase performance should be boosted since all the workload is distributed across all the RegionServers. In the long term, HBase split policy is supposed to uniformly distribute across servers so we shall not worry about it; only performance in the short term is critical. Yes, HBase needs a lot of data to give its best.

In order to perform presplitting, we are going to take advantage of the fact that we know how many rows are in 150 MB of data we inserted. To figure this out, in a HBase shell, run the following:

```
count 'wines', INTERVAL => 100000, CACHE => 10000
```

They should be around 1300000, which we are going to round up to 1500000. Afterwards, we more or less uniformly create a new *wines.1500000* presplit table by setting a key prefix for each region. This should be run in HBase shell.

```
create 'wines.1500000', 'all', SPLITS => ['12', '14', '2', '4', '6', '8']
```

And finally populate it with 1500000 rows. Next command should be run in Linux console.

```
hadoop-2.5.1/bin/hadoop jar labo2.jar write
                            -hbase -instances 1500000 wines.1500000
```

1. With such results at hand, say whether the balancing now is better or not. What about the RegionServers where they are? Compare this with the previous results.

2. Justify how many rows should be in each region due to the key prefixes we chose *'12', '14', '2', '4', '6'* and *'8'*

### 4.5 Exercise 5 (15min): 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 looks for a specific attribute 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.

## 5 Part C: Optional Practice (3h)

Before going ahead in this section, you should first fix how much data you are going to insert in HBase to do it.

***Deliverable: Upload a single zip file to the campus containing all the Java classes you have modified and a document saying the amount of data used and answering the questions below (please specify the question item).***

### 5.1 Exercise 1: Vertical partitioning

Create a table with four families for this exercise:

```
create 'wines_c_1', 'col_1', 'col_2', 'col_3', 'col_4'
```

1. Implement the function **toFamily()** in *MyHBaseWriter_C_1* so that different attributes go into different families as shown in table 1. You will also need to update the *Main* class to load *MyHBaseWriter_C_1* as writer (uncomment the right one). Then compile your code in a new JAR called *labo2_c_1.jar* and insert as much data as you have decided:

```
hadoop-2.5.1/bin/hadoop jar labo2_c_1.jar write -hbase -size SIZE wines_c_1
```

| Attribute | Family |
|---|---|
| *m_acid* | *col_1* |
| *ash* | *col_2* |
| *alc* | *col_1* |
| *alc_ash* | *col_1* |
| *mgn* | *col_1* |
| *t_phenols* | *col_2* |
| *flav* | *col_1* |
| *nonflav_phenols* | *col_1* |
| *proant* | *col_2* |
| *col* | *col_1* |
| *hue* | *col_1* |
| *od280/od315* | *col_2* |
| *proline* | *col_3* |
| *type* | *col_4* |
| *region* | *col_4* |

Table 1: List of attributes and families

2. Use HDFS to check how much disk space is consumed by each family. How much is it? Does it make sense with respect to the number of attributes we inserted in each family?

```
hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines_c_1/*/col_1
hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines_c_1/*/col_2
hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines_c_1/*/col_3
hadoop-2.5.1/bin/hdfs dfs -du -s -h /hbase/data/default/wines_c_1/*/col_4
```

3. Implement the function **scanFamilies()** in *MyHBaseReader_C_1* so that HBase scan is configured to only retrieve families *col_3* and *col_4*. You will also need to update the *Main* class to load *MyHBaseReader_C_1* as reader (uncomment the right one). Then recompile the same JAR *labo2_c_1.jar*. You can check the output by reading the table:

```
hadoop-2.5.1/bin/hadoop jar labo2_c_1.jar read -hbase wines_c_1
```

4. Compare the total time needed to scan the whole by using the old *MyHBaseReader* and your new *MyHBaseReader_C_1*. Discuss the impact of this vertical partitioning on queries that only need *proline* and *region* attributes.

```
time hadoop-2.5.1/bin/hadoop jar labo2.jar read -hbase wines_c_1 > /dev/null
time hadoop-2.5.1/bin/hadoop jar labo2_c_1.jar read -hbase wines_c_1 > /dev/null
```

## 5.2 Exercise 2: Implementing the key design

Recreate the wines table we have been using for this exercise:

```
create 'wines_c_2', 'all'
```

Recall the key discussion in 4.5 and implement it. Imagine queries that retrieve only data for wines of a specific type and region.

1. Implement the function **nextKey()** in *MyHBaseWriter_C_2* to generate row keys based on the key design you have found useful to reduce the number of rows read for this case. You will also need to update the *Main* class to load *MyHBaseWriter_C_2* as writer (uncomment the right one). Then compile your code in a new JAR called *labo2_c_2.jar* and insert as much data as you have decided:

```
hadoop-2.5.1/bin/hadoop jar labo2_c_2.jar write -hbase -size SIZE wines_c_2
```

2. Implement the functions **scanStart()** and **stopScan()** in *MyHBaseReader_C_2* to query for wines of type *type_3* and region *0* without scanning all the table. You will also need to update the *Main* class to load *MyHBaseReader_C_2* as reader (uncomment the right one). Then recompile the same JAR *labo2_c_2.jar*. You can check the output by reading the table:

```
hadoop-2.5.1/bin/hadoop jar labo2_c_2.jar read -hbase wines_c_2
```

3. Compare the total time needed to scan the table by using the old *MyHBaseReader* and your new *MyHBaseReader_C_2*.

```
time hadoop-2.5.1/bin/hadoop jar labo2.jar read -hbase wines_c_2 > /dev/null
time hadoop-2.5.1/bin/hadoop jar labo2_c_2.jar read -hbase wines_c_2 > /dev/null
```

You might need to check the Scan API, which is available at:

`https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Scan.html`