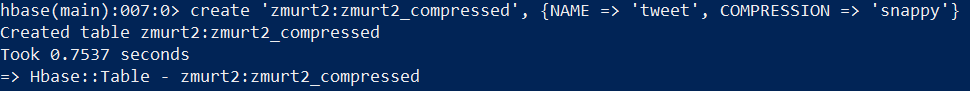
Integrated Hive with HBase to perform HBase compression to optimize disk storage usage and performance.

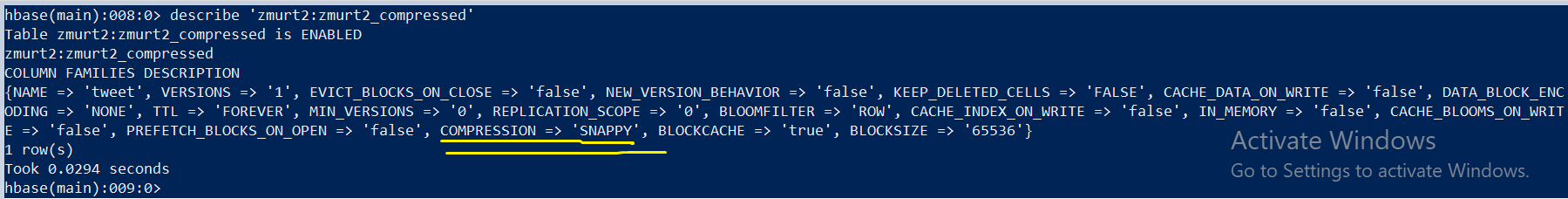
Used a large Twitter dataset was collected by a tweet collection program, yourTwapperKeeper(ytk). It consists of 12 different collections and has about 5 million tweets.

Loaded data into HBase, retrieved data from HBase using HiveQL and evaluated the effectiveness of compression tehniques in HBase. Used Ruby hash to change compression configuration with Snappy compression algorithm.

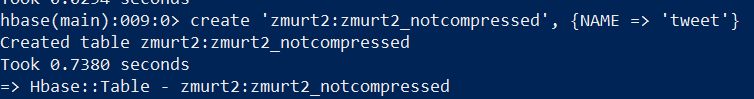
1.

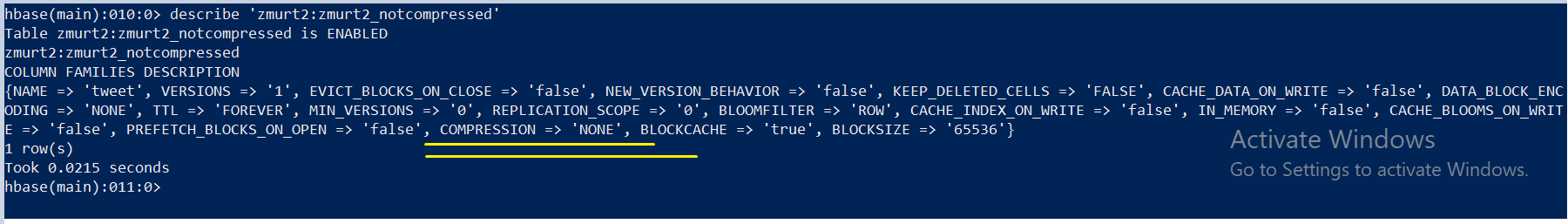
Creating a table **with compression**:

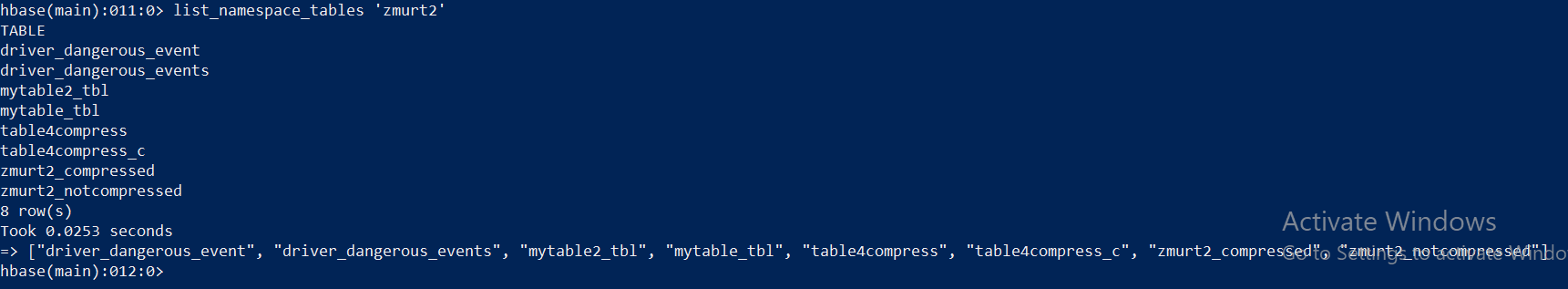


Description of **compressed table:**

Creating a table **without compression:**

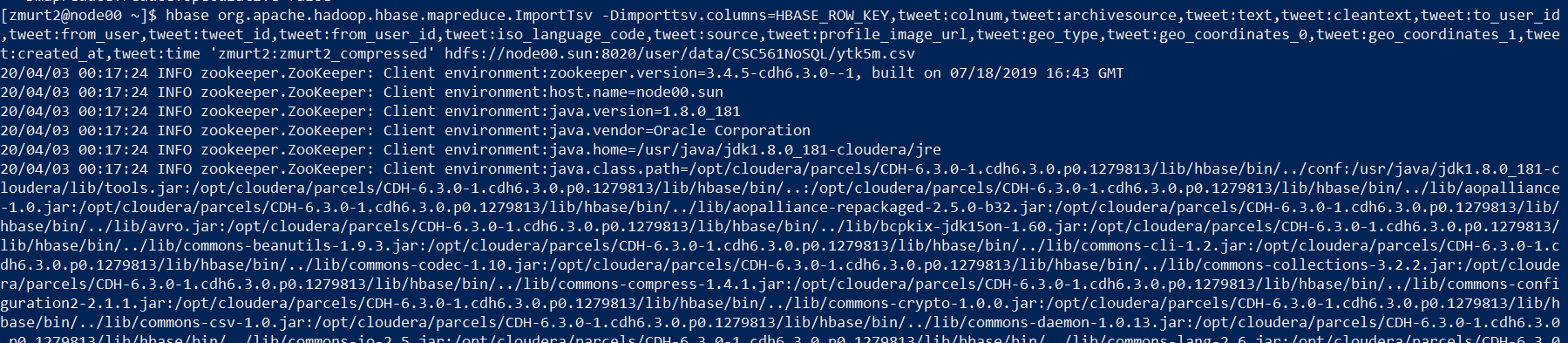


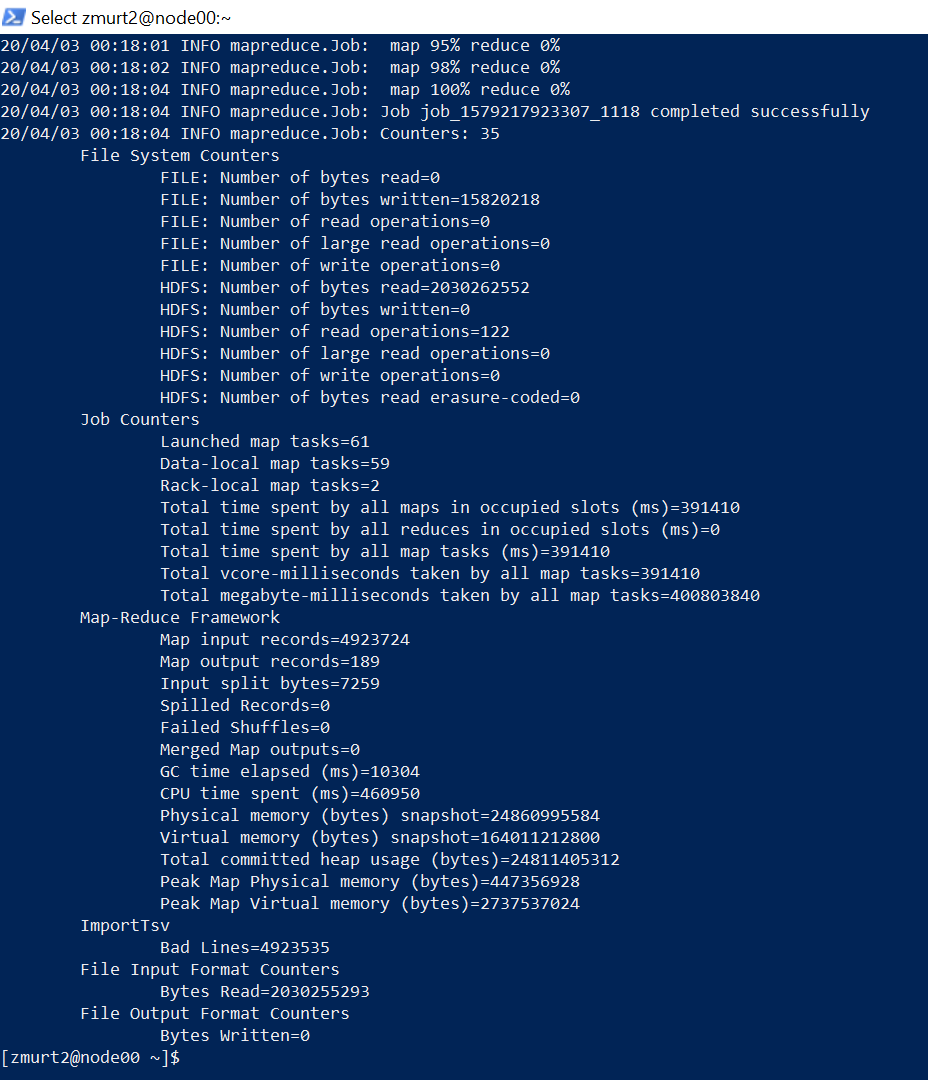
Description of **non-compressed table:**

Listing tables using HBase Shell command:

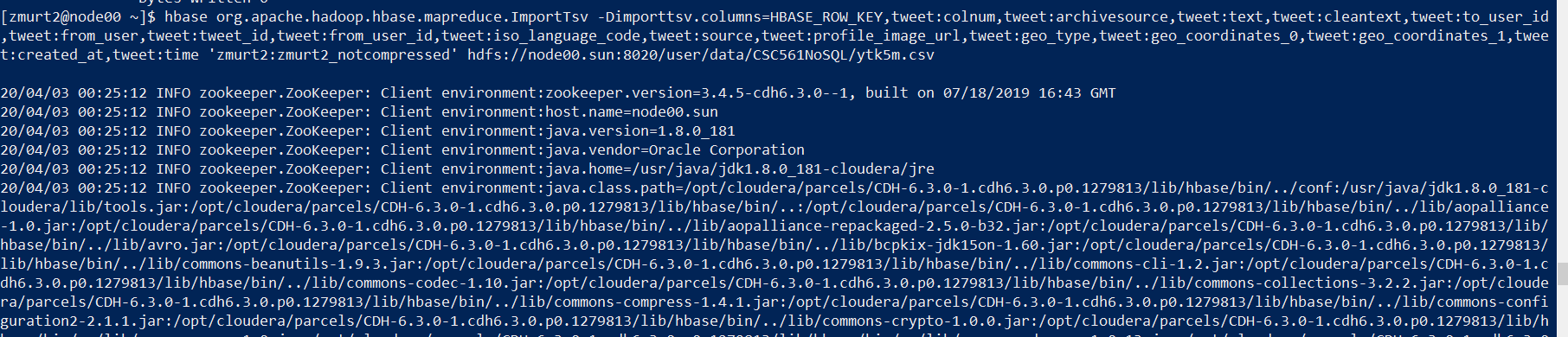
2.

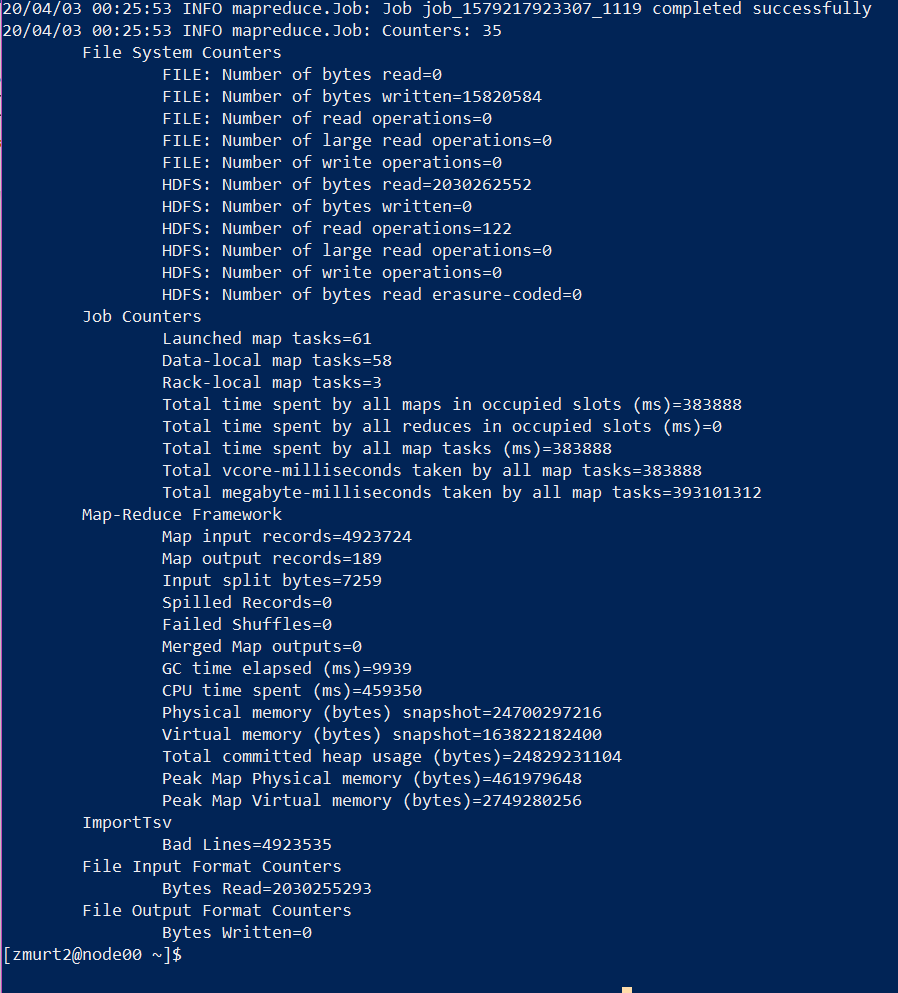
Loading ytk5m.csv into the **compressed table:**



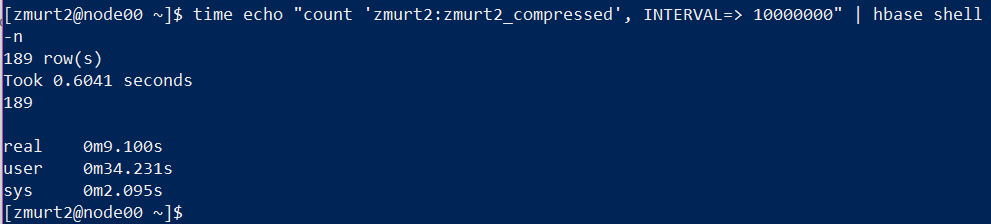


Loading ytk5m.csv into the **uncompressed table:**



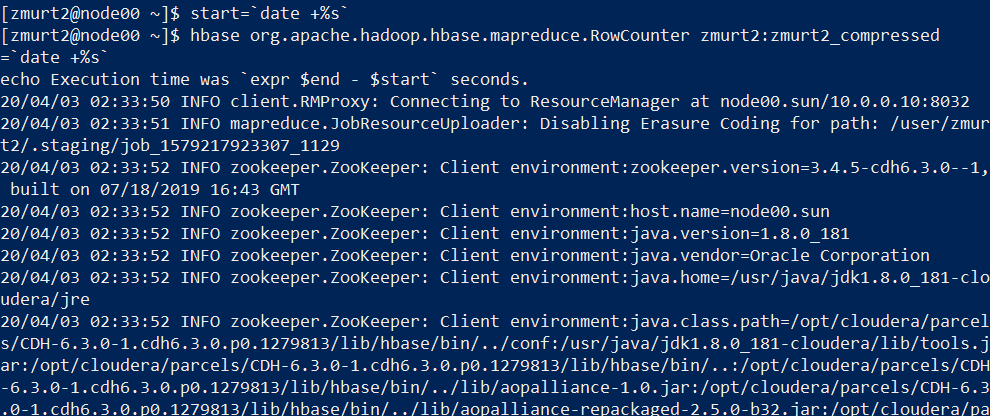


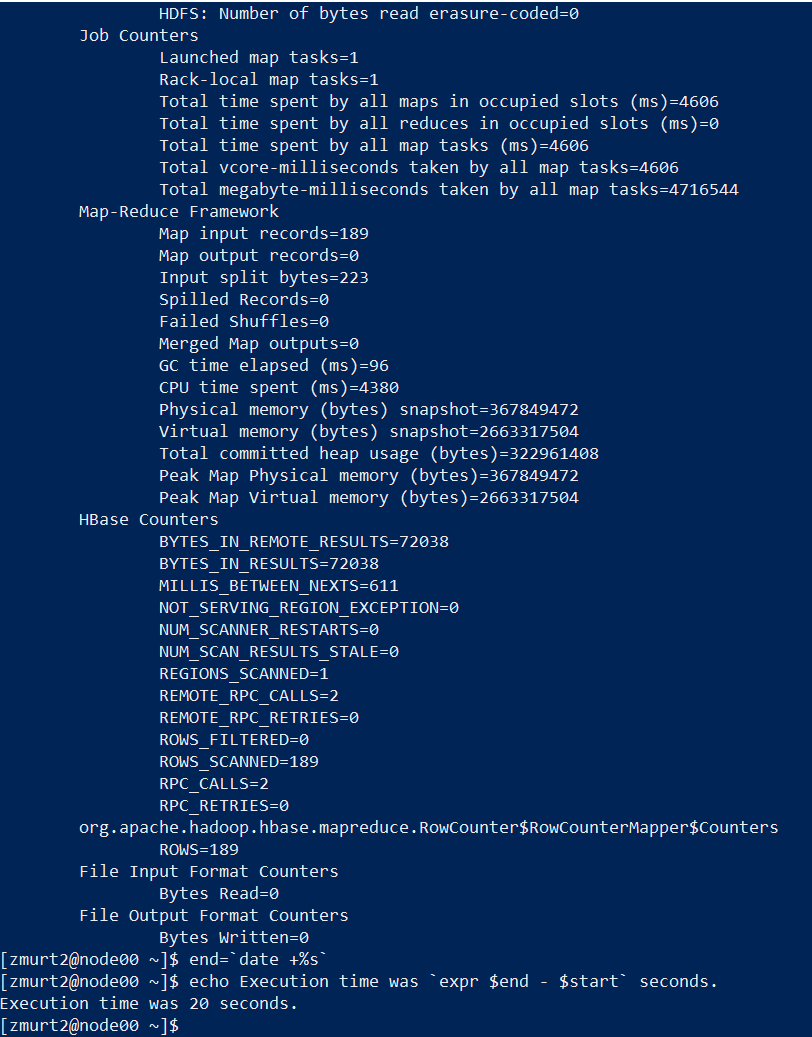
**Counting rows for the compressed table, using ECHO and “time” in Linux shell to determine execution time:**



Execution time: 9.1 seconds.

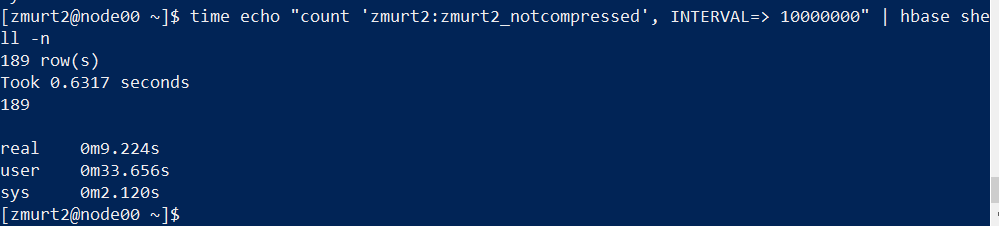
Using the **RowCounter class and start + end date commands in Linux Shell** for the **compressed table:**





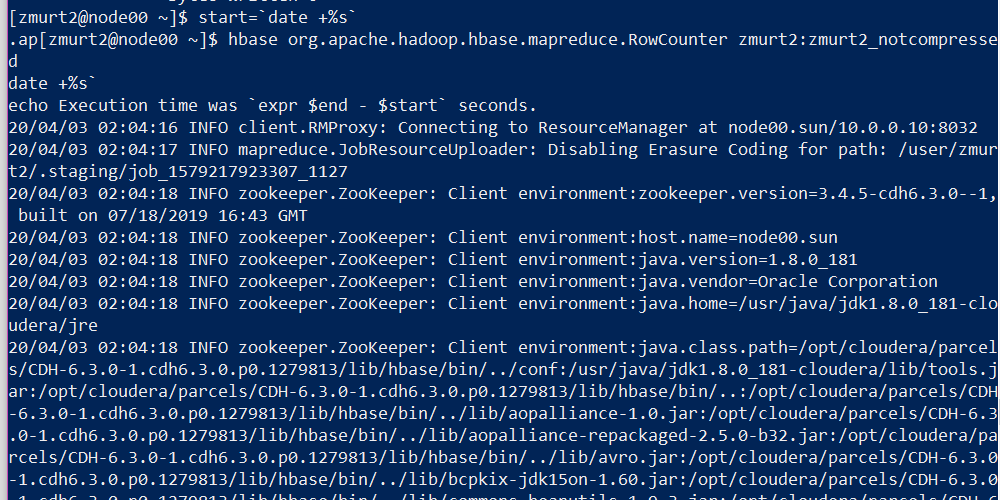
Execution time was 20 seconds.

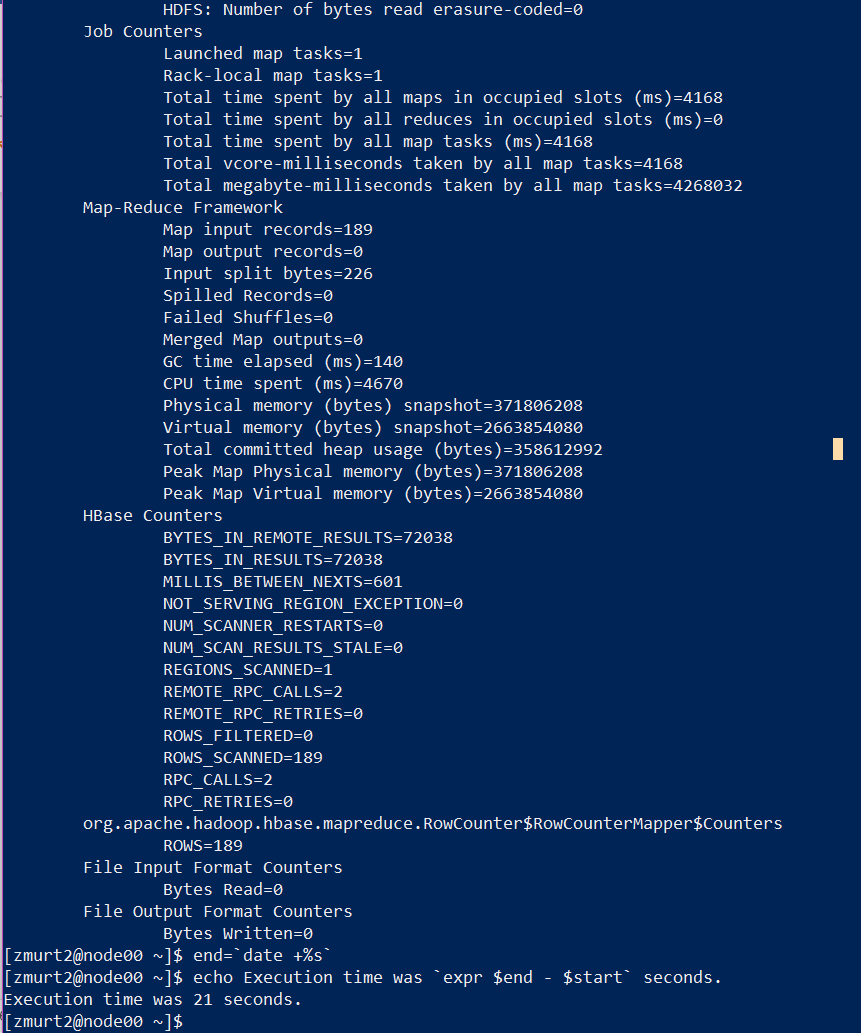
**Counting rows for the uncompressed table:**



Execution Time: 9.224 seconds

Using the **RowCounter class and execution time** for the **uncompressed table:**





Execution Time : 21 seconds

**Explanation of which approach is faster:** The echo approach is much faster. For the compressed and non compressed tables, the total time for execution for each table was around 9 seconds using echo. This is because, the echo command in linux is used to display line of text/string that are passed as an argument .

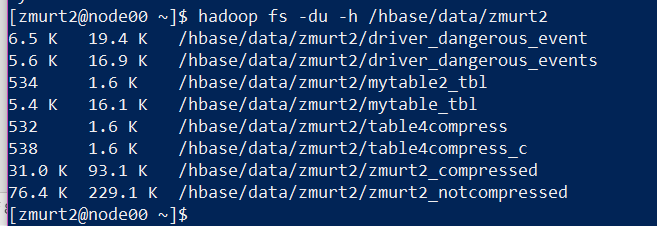
This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file. Here, it displays the output of the command run.

In contrast to that, using the RowCounter class of HBase in Linux Shell took an average of 20 seconds to execute. This is because of the MapReduce job that has to be executed, it takes substantially more time.

RowCounter is a mapreduce job that counts all the rows of a table.

This is a good utility to use as a sanity check to ensure that HBase can read all the blocks of a table if there are any concerns of metadata inconsistency.

Q3.



**Explanation:**

Zmurt2\_compressed is much smaller in size, as it is compressed with the aid of the Snappy algorithm. Zmurt2\_notcompressed, in comparison, is much larger as it was not compressed.

By compressing zmurt2\_compressed, the data has now occupied less space on the disk. In return, data is read on a compressed and smaller data size, which in turn yields an increased performance of reads.

The data that is not compressed results in more space taken, and an increase in CPU load.

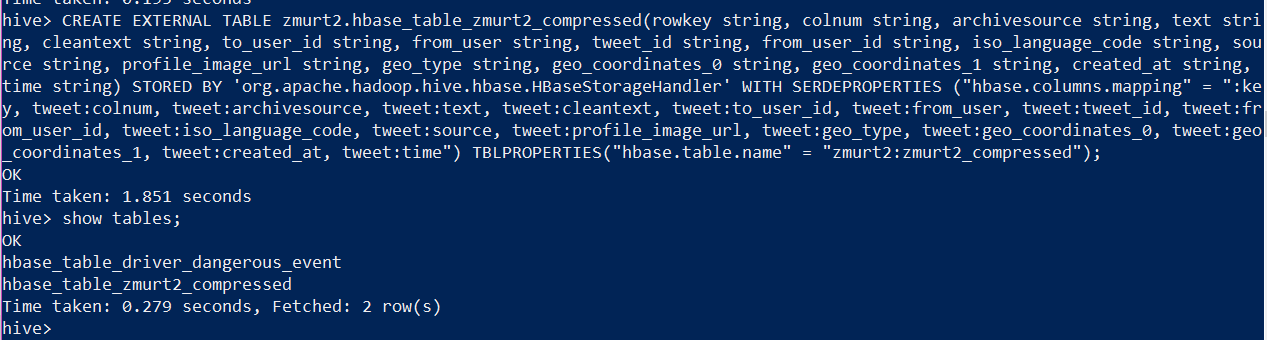
**The first column shows the actual size (raw size) of the files that users have placed in the various HDFS directories. The second column shows the actual space consumed by those files in HDFS.**

Zmurt2\_compressed **actual, raw size** is 31.0 K which is a smaller size than zmurt2\_notcompressed which is 76.4 K. Even the **actual space consumed**, as depicted in the second column is smaller. 93.1K is consumed by zmurt2\_compressed whilst 229.1K is consumed by zmurt2\_notcompressed.

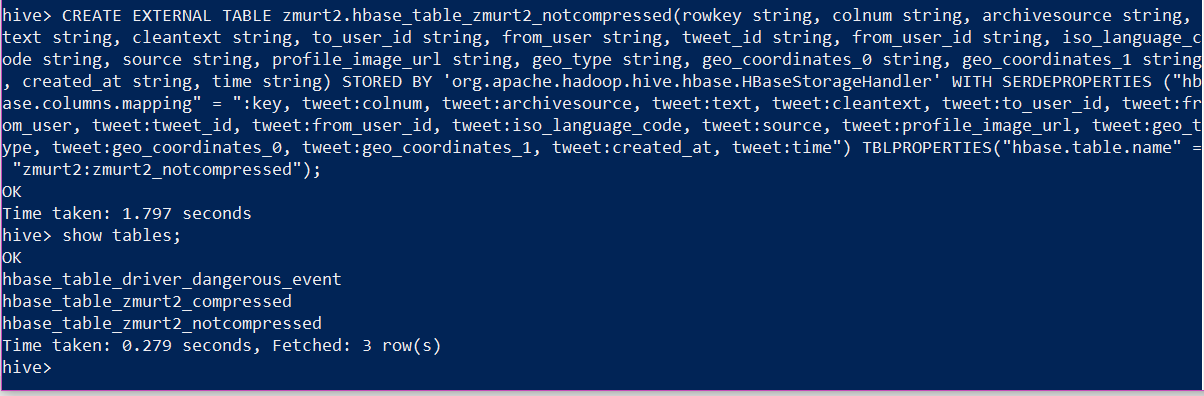
The values shown in the second column are much higher than the values shown in the first column. The reason is that the second column’s value is derived by multiplying the size of each file in a directory by its replication factor, to arrive at the actual space occupied by that file.

4.

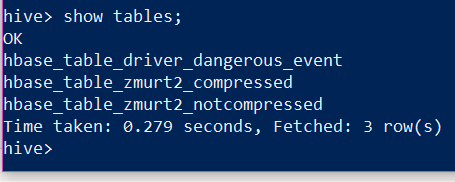
Creating the **compressed version external table** in Hive:

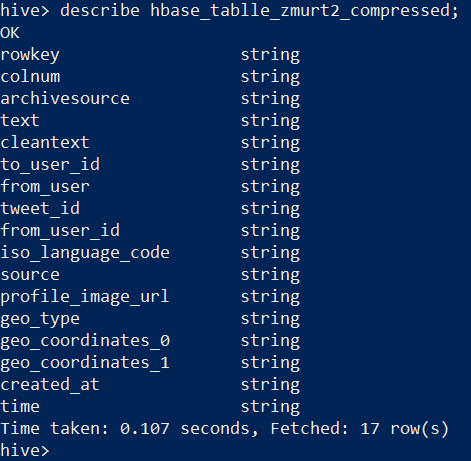


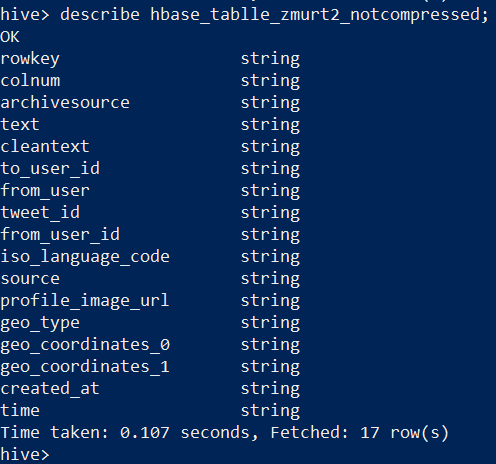
Creating the **uncompressed version external table** in Hive:



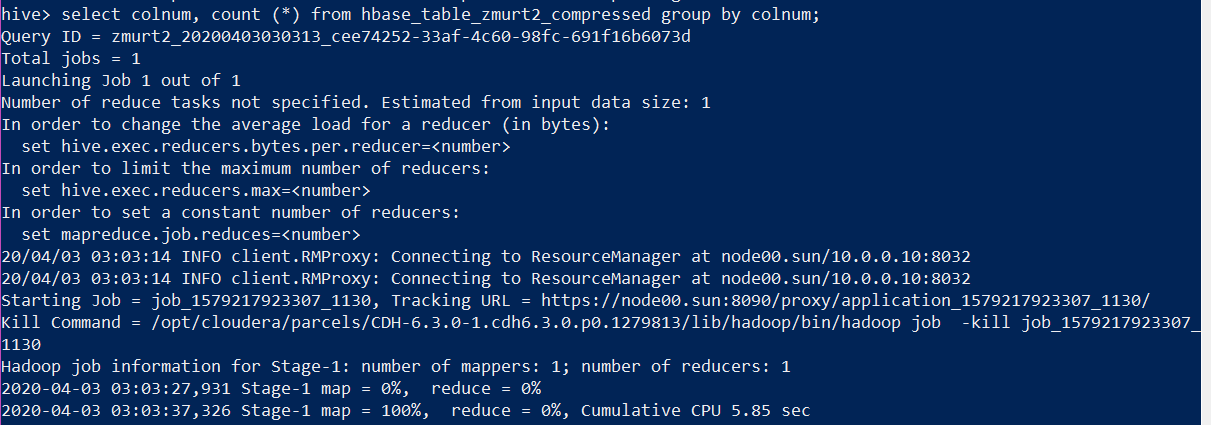
Listing tables:

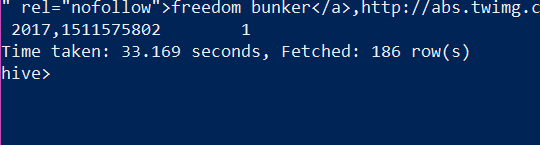




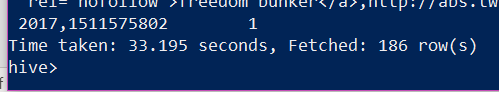


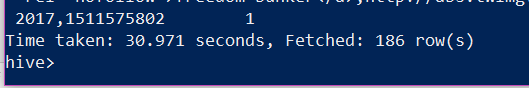
Query performance evaluation with the **compressed table:**





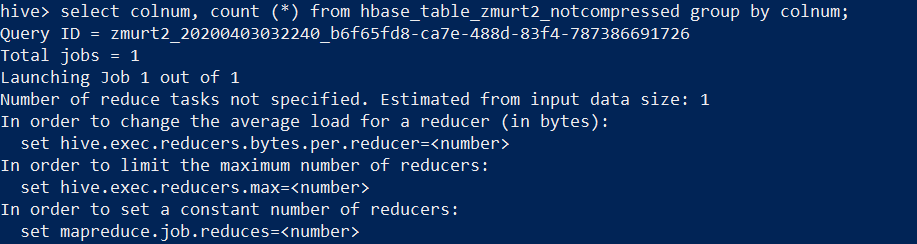
**Running the query twice more:**

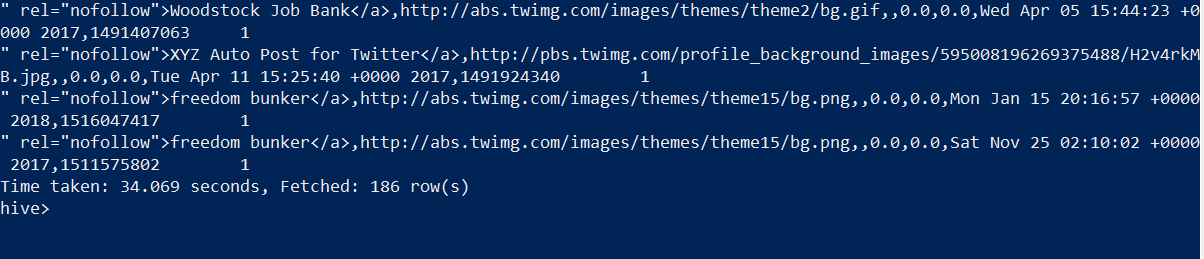




Average elapsed time: **32.445 seconds.**

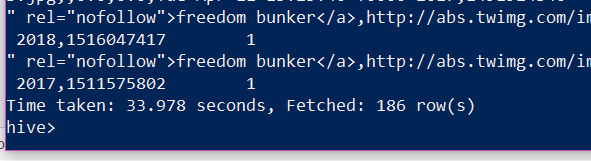
Query Performance Evaluation with the **non-compressed table:**

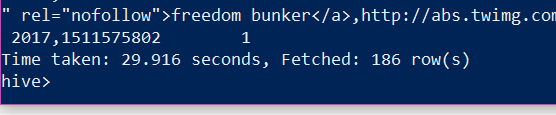




Elapsed time: **34.069 seconds**

**Running the query twice more:**



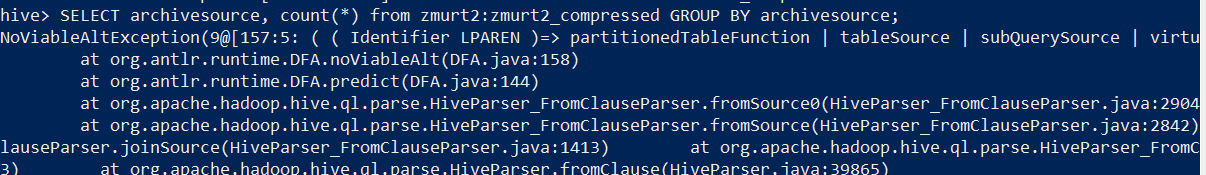


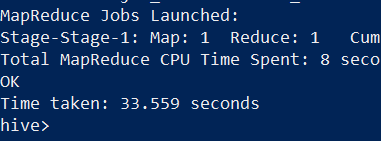
**Average elapsed time: 32.654**

**Since there was no substantial execution time difference whilst working with colnum, I wanted to base my explanation on using archivesource instead.**

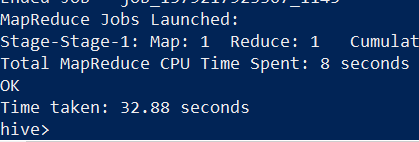
**Explanation:** According to my results with evaluation queries with archivesource, noncompressed runs a little faster because it takes less MapReduce CPU time to execute. MapReduce does not need to decompress the file in order to read with it, it is already decompressed. That is why it takes less time to execute. We have used the Snappy compression algorithm which does not allow splitting. Spltting, in of itself indicates whether the compression format supports splitting, that is, whether you can seek to any point in the stream and start reading from some point further on. The Snappy algorithm is known to be fast for decompression.

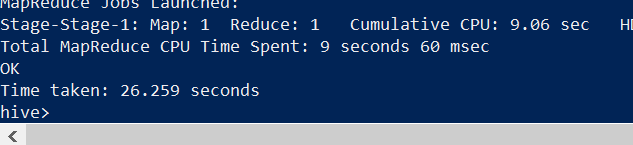
Using another column for evaluation query, **archivesource** for **compressed table:**





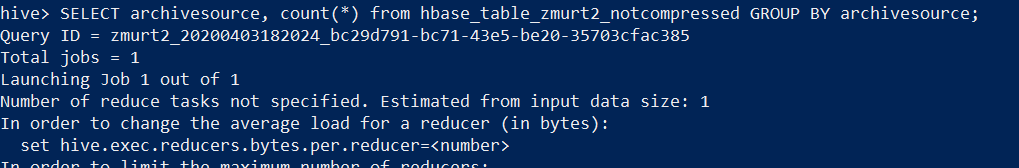
Running the query twice more:

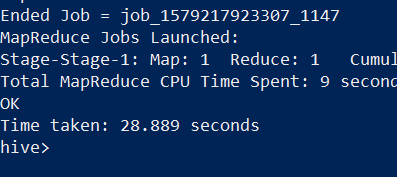




**Average elapsed time is: 30.899**

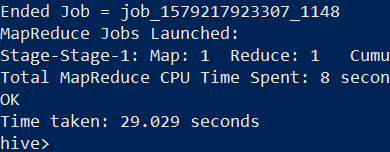
Using the column archivesource for the **non-compressed table:**

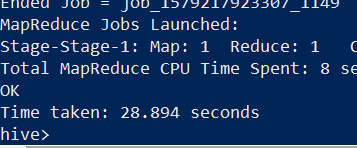




**Time taken: 28.889**

**Running the query twice more:**





**Average time for execution: 28.93**