Large Scale Data Management Course

Lamprakis Panagiotis (03003142)

PhD Candidate – AILS

Contents

[Introduction 3](#_Toc169117838)

[Work 6](#_Toc169117839)

[Query 1 6](#_Toc169117840)

[Query 2 8](#_Toc169117841)

[Query 3 8](#_Toc169117842)

[Query 4 9](#_Toc169117843)

[Appendix 10](#_Toc169117844)

# Introduction

In scope of the Large-Scale Data Management course, I have set up a mini cluster of two nodes using VirtualBox[1]. I have created a master and a worker node and installed Spark and Hadoop on them. The images (1 & 2) below verify the successful installation of Spart and Hadoop on both instances. Also, in the images below, it’s noticeable that passwordless ssh between master & worker has been established.

The code for this homework can be found in [PanosLam/DSML-NTUA-Big-Data](https://github.com/PanosLam/DSML-NTUA-Big-Data) GitHub repo.

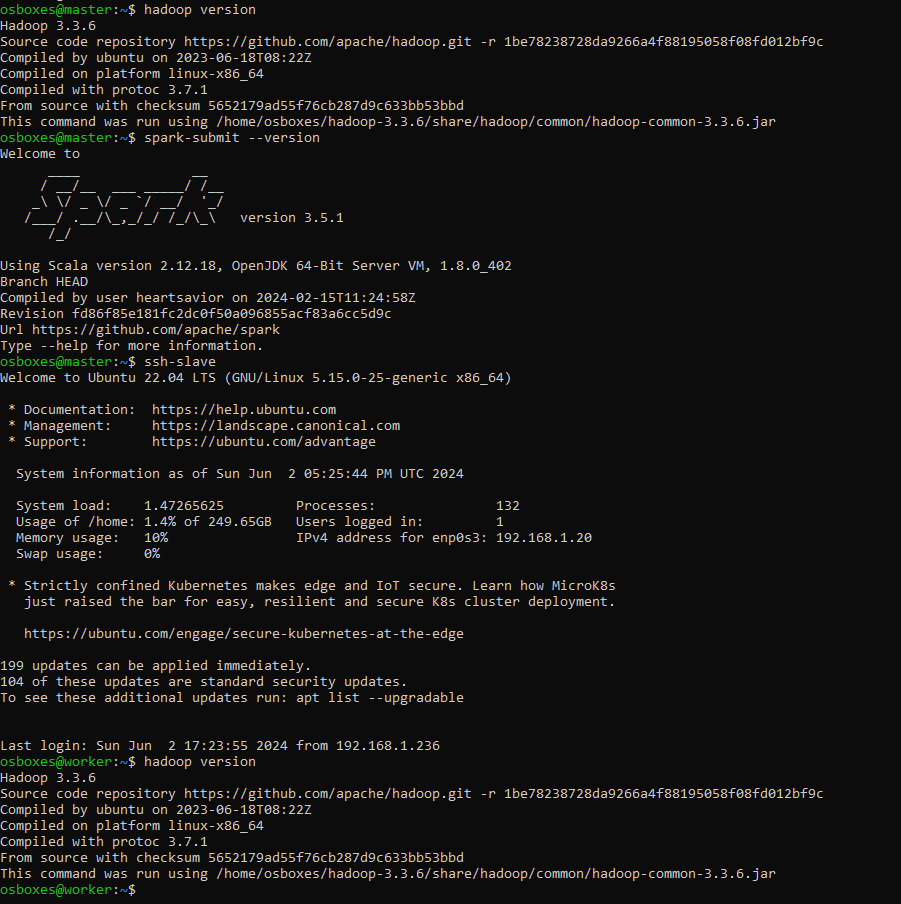


Image 1 Hadoop & Spark versions in Master Node

Master’s private IP is *192.168.1.236* (verified via *ifconfig* command), while worker’s private IP is *192.168.1.20*.



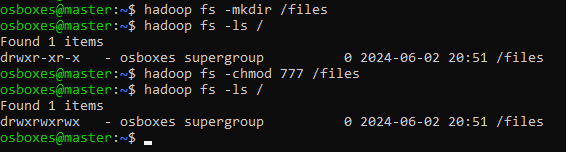
Image 2 Hadoop & Spark versions in Worker Node

Hadoop web interface can be accessed via: <http://192.168.1.236:9870/dfshealth.html#tab-datanode> hyperlink [2].

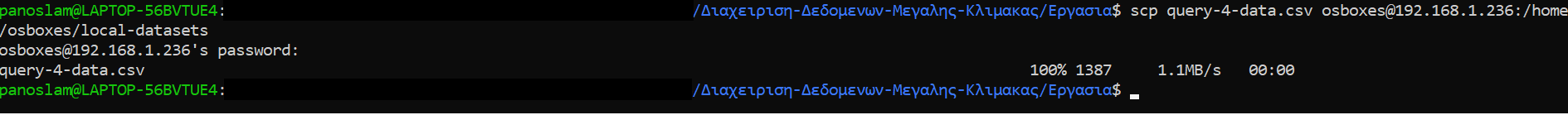
Spark web UI for Master node can be accessed via: <http://192.168.1.236:8080/> hyperlink, and via <http://192.168.1.20:8081/> for the Worker Node.

In order to verify the code, I have setup Spark locally in my PC and run the experiments over a small but representative amount of data. The final results come by executing the code in the cluster.

Creating a folder on HDFS.



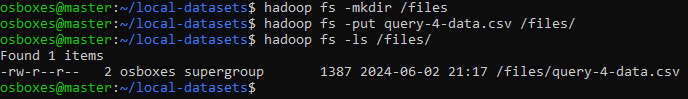
Copying the files from local (PC) into the Cloud (VirtualBox – Master instance)



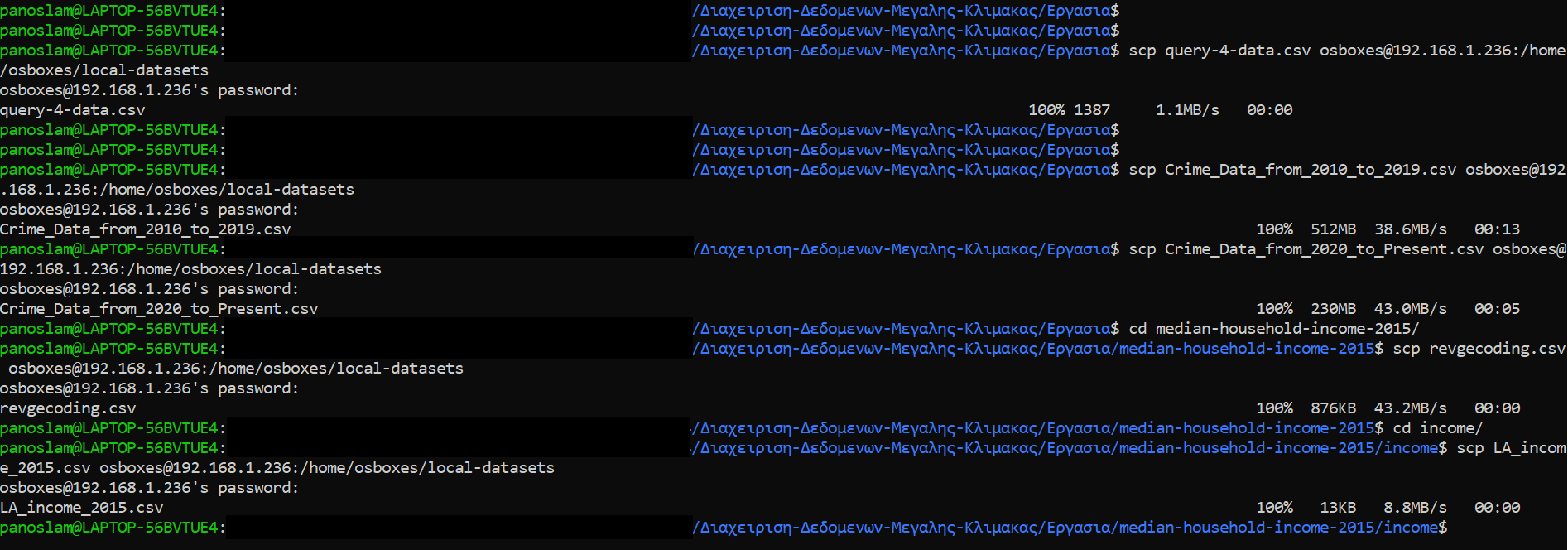
In case the DataNode is not up and running we need to perform the following steps:

1. Run *stop-dfs.sh*
2. Run *mv /home/osboxes/hdfsdata /home/osboxes/hdfsdata\_backup* (remove DataNode directories and backup them)
3. Run *start-dfs.sh*

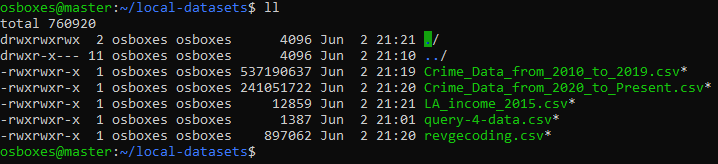
Make a directory in HDFS and upload a file from Master local file system into HDFS.



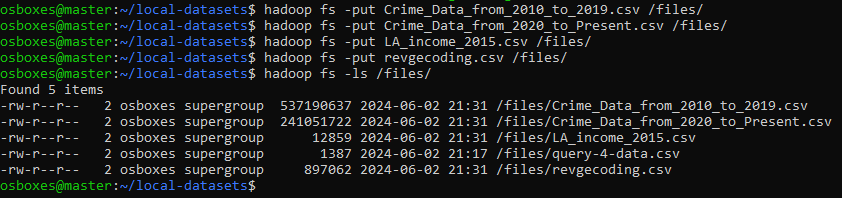
The image below shows that all necessary datasets have been uploaded from local PC to Master FS.



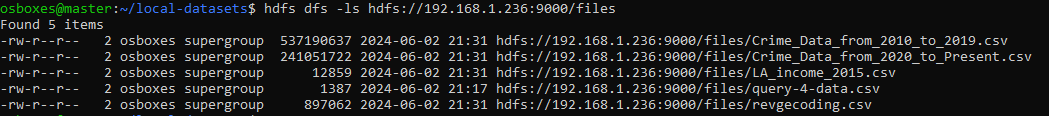
The following image shows the datasets in Master FS.



Finally, the image below verifies that datasets have been uploaded successfully to HDFS.



Verify HDFS uri as created in python code to be used for saving parquet files in HDFS.



# Work

## Query 1

We implement the first query in using both SQL API & DataFrame and we read the data from the CSV file and the parquet file. The table below shows the total execution times for all possible combinations.

|  |  |  |
| --- | --- | --- |
|  | CSV | Parquet |
| SQL API | 55.259 sec | 26.472 sec |
| DataFrame | 54.378 sec | 25.521 sec |

We observe that in general, using the parquet format results in 52-53% execution time cut compared to the CSV cases. This happens for the following reasons:

1. Parquet file formats tend to give better read times because of their columnar storage and their included schema information as opposed to CSV files which are row-based storage and need schema inference. This means that when Spark needs to read specific columns, it’s faster for parquet cases compared to csv, in which Spark needs to read whole rows.
2. Parquet files are compressed, leading to smaller data sizes being transmitted over the network compared to the csv case. This gives an improvement on the network I/O latency.
3. Computations held over parquet data are more efficient, since batches of data can be processed in parallel (vectorized processing).

It is evident that a parquet file format ends up with more benefits.

|  |  |
| --- | --- |
| Image 3 DataFrame results | Image 4 SQL API results |

## Query 2

|  |  |
| --- | --- |
| Image 5 DataFrame results | Image 6 RDD results |

|  |  |  |
| --- | --- | --- |
|  | RDD | DataFrame |
| Execution time (sec) | 74.997 | 45.779 |

It is easy to observe that using DataFrames over RDDs we achieve an improvement of almost 40% (reduction) in execution time. This is because RDDs offer a low-level API to interact with the data and it doesn’t come with optimizations, compared to DataFrames which use the Spark Catalyst optimizer. Moreover, RDDs store the data in Java objects, which can be inefficient, while DataFrames store the data in binary format, which reduces serialization/deserialization costs, thus improving the execution time.

## Query 3

Due to limited time, I was not able to experiment with the different join implementations offered by Spark API. However, I will attempt to guess the best join implementation based on the nature of the different Spark implementations and the dataset under analysis.

The **broadcast** join broadcasts the small dataset to the worker node(s) and the join is performed locally on each node. This join type is effective when merging a small with a large dataset.

The **merge** join sorts both data sets by key and then merges them. This join type is efficient for large datasets that are already sorted or can be sorted efficiently.

The **shuffle hash** join shuffles the data to the worker node(s) and performs the join on the shuffled data. This join type is a good option when the join keys fit into memory and the dataset is evenly distributed.

The **shuffle and replicate NL** join takes each row of one dataset and compares it to every row in the other dataset to perform the join.

Given the above, I believe that **broadcast** join would be the most appropriate one, since we do have a small data set (produced by merging revgeocoding over the zip codes) which is joined with a large one (2015 crime dataset). Also, the **shuffle hash** join would work too, since the keys can fit into worker node memory.

I would expect **Merge** & **shuffle and replicate NL** joins to end up with worse execution times compared to the other two strategies.

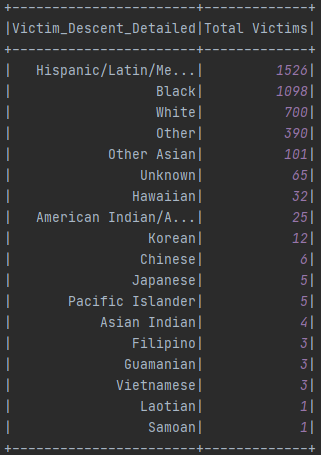


Image 7 DataFrame results

## Query 4

No time to implement the broadcast & repartition join algorithms. However, below there are the results by leveraging the join strategy selected by Catalyst Optimizer for both DataFrame and RDD implementations.

|  |  |
| --- | --- |
| Image 8 DataFrame results | Image 9 RDD results |

What differs between the results of the two implementations here is the numbers in the average distance (not rounded for the case of RDD). I must have done a mistake and I may have confused the longitude and latitude coordinations for the RDD case. However, due to time limitations I’m not in position of fixing it right now.

# Appendix

[1] Using the instructions provided in ***01\_lab1-virtualbox*** & ***0\_Preparatory\_lab\_virtual-box***.

[2] I have faced lots of problems by setting up the cluster in VirtualBox. The private IPs of main and worker nodes were constantly changing everytime I was powering off the machines. Additionally, out of nowhere, Hadoop starting prompting me an error about mismatched cluster ids.