**Programming on Cloud - COEN 424** Jafar Abbas (26346650)

**Assignment 2 Report** – Map-Reduce December 2nd 2017

**Introduction**

In this assignment I wrote a Map-Reduce program to compute analytics (max., min., median, standard deviation) on a public data set regarding Airborne Radioactivity. The program is written in Scala and executed in the Apache Spark cluster-computing framework.

**Requirements**

1. Compute Maximum, Minimum, Median and Standard Deviation of the metric “7Be MDC/7Be CMD (mBq/m3)” from the “The Canadian Radiological Monitoring Network – Airborne Radioactivity” data set (csv).
2. Display analytics of the above-mentioned metric for a location in each year.
3. The software is programmed and executed in a Map-Reduce runtime such as Apache Hadoop or AWS EMR. It cannot be a standalone program mimicking map-reduce behavior.
4. The execution environment is either a local computer or on the Cloud. Submission in a single package with source code and executable.

**Specifications**

1. Median is estimated as the 50th percentile using “percentile\_approx”. Standard deviation is based on the “stddev\_samp” (square root of variance of the sample). Min and Max is straightforward.
2. The data set defines location as city, and the date is in MM/DD/YYYY format so we need to extract the year as a substring from the date column. The city and year form the fields on which we group to calculate the aggregates.
3. Programming language used is SCALA and the Map-Reduce framework is Apache Spark.
4. Execution environment is local computer using a standalone Spark cluster, with a master and workers running locally. The results are combined and outputted in “results.csv” for every location/year pair.

**Implementation**

***a) Map-Reduce Algorithm Design***

In the Apache Spark framework there are no explicit “map” and “reduce” functions to define. Hence, I jumped into the implementation:

First, the CSV file is loaded (lazy) within an SQL context using the spark-csv package:

val full\_csv = sqlContext.read

.format("com.databricks.spark.csv")

.option("header", "true") // Use first line of all files as header

.option("inferSchema", "true") // Automatically infer data types (otherwise everything is assumed string)

.load(csvFile)

The next step is to select the required fields for our analytics based on the requirements:

val data = full\_csv.select("Location/Emplacement",

"Collection Start/Debut du prelevement (UTC)",

"7Be MDC/7Be CMD (mBq/m3)")

From the data set, it was clear the “date” is not properly formatted (MM/DD/YYYY) and so the year (YYYY) is extracted from the date and the columns are renamed for the sake of clarity:

val newNames = Seq("location", "date", "mdc")

val df = data.toDF(newNames: \_\*).withColumn("year", substring\_index(col("date"), "/", -1))

The required Analytics can now we performed on this “cleaned” data set. The operations we want are “min”, “max”, “stddev” and “median” of the MDC metric for all locations per year:

val mapping: Map[String, Column => Column] = Map(

"min" -> min, "max" -> max, "mean" -> avg, "stddev" -> stddev)

val groupBy = Seq("location", "year")

val aggregate = Seq("mdc")

val operations = Seq("min", "max", "mean", "stddev")

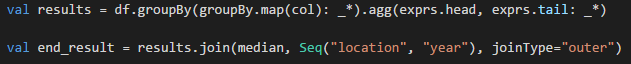
val exprs = aggregate.flatMap(c => operations .map(f => mapping(f)(col(c))))

df.registerTempTable("df ")

var median = sqlContext.sql("select location, year, percentile\_approx(mdc, 0.5) as median from df group by location, year")

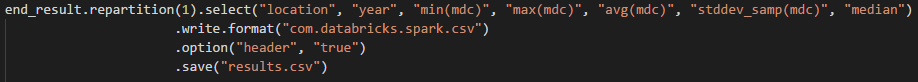
This is the core of our “map-reduce” algorithm design, we group the data by “location” and then by “year” and perform the aggregate operations “min”, “max”, “mean” (extra), “stddev” on “mdc”. These operations are already defined and optimized within the sql.functions package.

Since the median function does not exist, we write the sql query based on the ‘percentile\_approx’ function to approximate the 50th percentile.



Essentially, the “MAP” and “REDUCE” portions of our algorithm are handled internally by the Spark framework. I do not explicitly define them, rather, I make use of pre-defined grouping and functions on the data frame.

To get the results, we execute the above defined operations on our dataset and join the previous results with the median results.



Finally we repartition (merge) the results together into one csv file and save it in the “results.csv” folder.

***b) HOW-TO Run the Program***

**Requirements**:

Java v1.7+

Scala v2.1+

Apache Spark binaries

Hadoop’s winutils.exe tool

Environment variables

Windows 7

**Installation guide** and links for the required binaries can be found here:

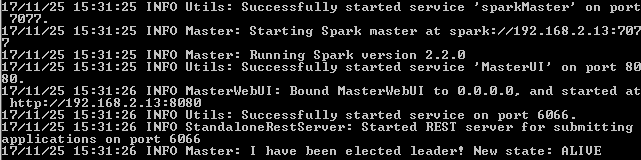
<https://edumine.wordpress.com/2015/06/11/how-to-install-apache-spark-on-a-windows-7-environment/>

**Running the program:**

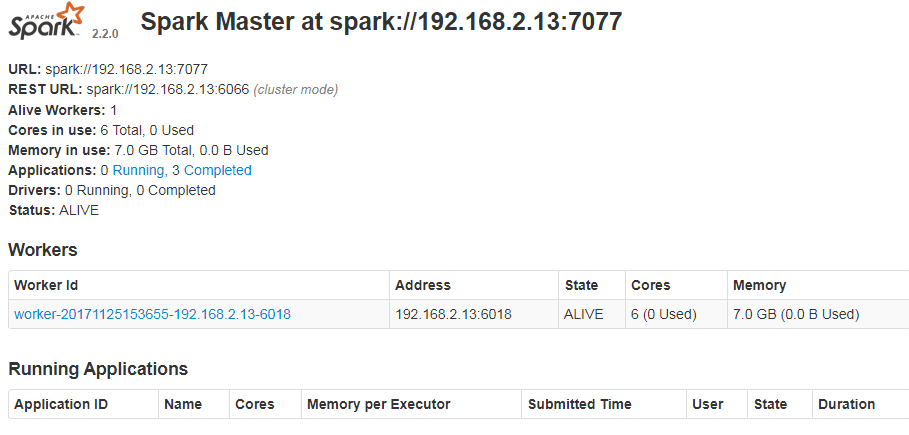
1) Run the Master with following command:



If successful, you should see:



The master also has a default webpage visible at “localhost:8080”:



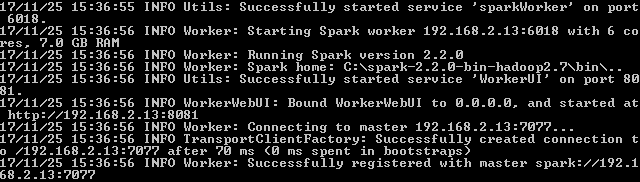
2) Run at-least ONE Worker under the master with the following command:



Notice the worker is targeted to where the Spark master has been ran, i.e. “spark://192.168.2.13:7077”.

In this step, we can also specify the amount of memory and cores the worker is limited to.

If successful, the worker will be registered with the master:



And from the master’s perspective:



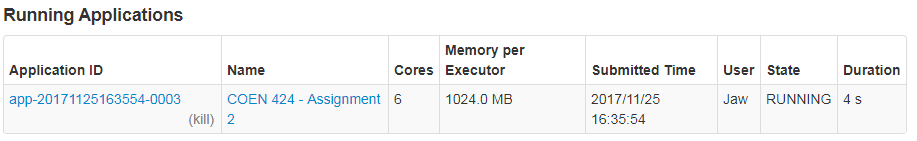
3) Submit the job to the cluster with the following command (within project directory):



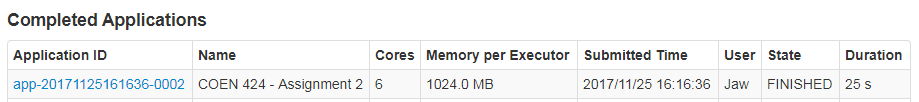
Optional: Recompile the package by using: 

On the master’s UI, you can see the job:

While it’s running:



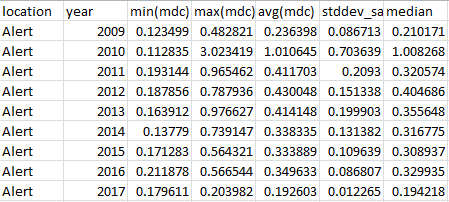
And after completion:



4) View the results inside the folder as:



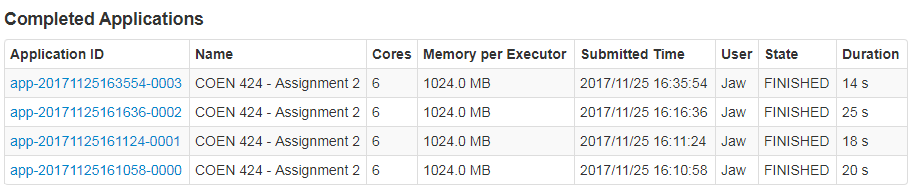
This is the CSV file with all the parts merged. You can now sort it by ‘location’ then ‘year’ to view the results neatly:



***c) Timing of jobs***

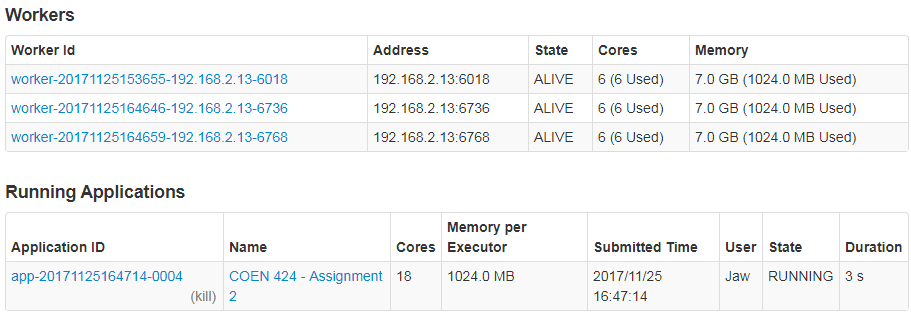
Unfortunately Apache Spark does not provide timings in the logs for the map task and reduce task separately.

However, we have access to the combined time for the job which seems to actually vary. Here’s some sample timings:



***d) Data partition and workload balancing***

I tested having 3 workers in total doing the same job compared to 1 worker:



Each worker gets assigned up to 228 tasks:



With 3 workers it’s about the same number of tasks per worker:

Worker 1: 

Worker 2: 

Worker 3: 

However, **the difference is the size of each task**:

For 1 worker:





For 3 workers:



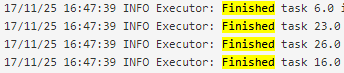


Another thing I noticed is with 3 workers, since the partitions are smaller

Therefore, load balancing is done via data partitioning. Although the numbers of chunks are the same for a different number of workers, each worker actually gets a smaller data chunks the more workers exist.

Furthermore, I noticed the 3 workers more quickly and frequently finishing their tasks since their tasks are actually smaller.

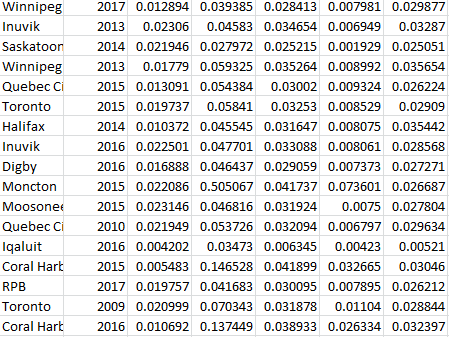
For 3 workers:



Notice the tasks are being quickly finished. For the case with 1 worker, there’s more time between tasks.

This load balancing is also dependent on the resources of each worker. In my case all workers had the same amount of memory and processing power. However, the master would allocate properly sized chunks for a slower worker.

***Sample results:***





*Full results inside the .csv file within the “results.csv” directory.*