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

a) 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).

b) Display analytics of the above-mentioned metric for a location in each year.

c) 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.

d) The execution environment is either a local computer or on the Cloud. Submission in a single package with source code and executable.

Specifications

a) 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.

b) 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.

c) Programming language used is SCALA and the Map-Reduce framework is Apache Spark.

d) 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.

```
val results = df.groupBy(groupBy.map(col): _*).agg(exprs.head, exprs.tail: _*)

val end_result = results.join(median, Seq("location", "year"), joinType="outer")
```

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.

```
end_result.repartition(1).select("location", "year", "min(mdc)", "max(mdc)", "avg(mdc)", "stddev_samp(mdc)", "median")
                      .write.format("com.databricks.spark.csv")
                      .option("header", "true")
                      .save("results.csv")
```

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:

```
spark-class org.apache.spark.deploy.master.Master
```

If successful, you should see:

```
17/11/25 15:31:25 INFO Utils: Successfully started service 'sparkMaster' on port
 7077.
17/11/25 15:31:25 INFO Master: Starting Spark master at spark://192.168.2.13:707
7
17/11/25 15:31:25 INFO Master: Running Spark version 2.2.0
17/11/25 15:31:25 INFO Utils: Successfully started service 'MasterUI' on port 80
80.
17/11/25 15:31:26 INFO MasterWebUI: Bound MasterWebUI to 0.0.0.0, and started at
 http://192.168.2.13:8080
17/11/25 15:31:26 INFO Utils: Successfully started service on port 6066.
17/11/25 15:31:26 INFO StandaloneRestServer: Started REST server for submitting
applications on port 6066
17/11/25 15:31:26 INFO Master: I have been elected leader! New state: ALIVE
```

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

Spark Master at spark://192.168.2.13:7077

Spark 2.2.0

URL: spark://192.168.2.13:7077
REST URL: spark://192.168.2.13:6066 *(cluster mode)*
Alive Workers: 1
Cores in use: 6 Total, 0 Used
Memory in use: 7.0 GB Total, 0.0 B Used
Applications: 0 Running, 3 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20171125153655-192.168.2.13-6018	192.168.2.13:6018	ALIVE	6 (0 Used)	7.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration

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

```
spark-class org.apache.spark.deploy.worker.Worker  spark://192.168.2.13:7077
```

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:

```
17/11/25 15:36:55 INFO Utils: Successfully started service 'sparkWorker' on port
 6018.
17/11/25 15:36:56 INFO Worker: Starting Spark worker 192.168.2.13:6018 with 6 co
res, 7.0 GB RAM
17/11/25 15:36:56 INFO Worker: Running Spark version 2.2.0
17/11/25 15:36:56 INFO Worker: Spark home: C:\spark-2.2.0-bin-hadoop2.7\bin\..
17/11/25 15:36:56 INFO Utils: Successfully started service 'WorkerUI' on port 80
81.
17/11/25 15:36:56 INFO WorkerWebUI: Bound WorkerWebUI to 0.0.0.0, and started at
 http://192.168.2.13:8081
17/11/25 15:36:56 INFO Worker: Connecting to master 192.168.2.13:7077...
17/11/25 15:36:56 INFO TransportClientFactory: Successfully created connection t
o /192.168.2.13:7077 after 70 ms (0 ms spent in bootstraps)
17/11/25 15:36:56 INFO Worker: Successfully registered with master spark://192.1
68.2.13:7077
```

And from the master’s perspective:

```
17/11/25 15:36:56 INFO Master: Registering worker 192.168.2.13:6018 with 6 cores
, 7.0 GB RAM
```

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

```
spark-submit target/scala-2.10/simple-project_2.10-1.0.jar spark://
192.168.2.13:7077 \    --class MapReduce
```

Optional: Recompile the package by using: `sbt package`

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

While it's running:

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20171125163554-0003 (kill)	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:35:54	Jaw	RUNNING	4 s

And after completion:

Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20171125161636-0002	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:16:36	Jaw	FINISHED	25 s

4) View the results inside the results.csv folder as:

part-00000-8775cc8d-a48c-48b5-bf00-24...

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

location	year	min(mdc)	max(mdc)	avg(mdc)	stddev_sa	median
Alert	2009	0.123499	0.482821	0.236398	0.086713	0.210171
Alert	2010	0.112835	3.023419	1.010645	0.703639	1.008268
Alert	2011	0.193144	0.965462	0.411703	0.2093	0.320574
Alert	2012	0.187856	0.787936	0.430048	0.151338	0.404686
Alert	2013	0.163912	0.976627	0.414148	0.199903	0.355648
Alert	2014	0.13779	0.739147	0.338335	0.131382	0.316775
Alert	2015	0.171283	0.564321	0.333889	0.109639	0.308937
Alert	2016	0.211878	0.566544	0.349633	0.086807	0.329935
Alert	2017	0.179611	0.203982	0.192603	0.012265	0.194218

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:

Completed Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20171125163554-0003	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:35:54	Jaw	FINISHED	14 s
app-20171125161636-0002	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:16:36	Jaw	FINISHED	25 s
app-20171125161124-0001	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:11:24	Jaw	FINISHED	18 s
app-20171125161058-0000	COEN 424 - Assignment 2	6	1024.0 MB	2017/11/25 16:10:58	Jaw	FINISHED	20 s

d) Data partition and workload balancing

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

Workers

Worker Id	Address	State	Cores	Memory
worker-20171125153655-192.168.2.13-6018	192.168.2.13:6018	ALIVE	6 (6 Used)	7.0 GB (1024.0 MB Used)
worker-20171125164646-192.168.2.13-6736	192.168.2.13:6736	ALIVE	6 (6 Used)	7.0 GB (1024.0 MB Used)
worker-20171125164659-192.168.2.13-6768	192.168.2.13:6768	ALIVE	6 (6 Used)	7.0 GB (1024.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20171125164714-0004 (kill)	COEN 424 - Assignment 2	18	1024.0 MB	2017/11/25 16:47:14	Jaw	RUNNING	3 s

Each worker gets assigned up to 228 tasks:

```
17/11/25 16:36:08 INFO CoarseGrainedExecutorBackend: Got assigned task 228
```

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

Worker 1: `17/11/25 16:47:45 INFO CoarseGrainedExecutorBackend: Got assigned task 224`
Worker 2: `17/11/25 16:47:45 INFO CoarseGrainedExecutorBackend: Got assigned task 227`
Worker 3: `17/11/25 16:47:45 INFO CoarseGrainedExecutorBackend: Got assigned task 228`

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

For 1 worker:

```
17/11/25 16:36:08 INFO MemoryStore: Block broadcast_11_piece0 stored as bytes in memory (estimated size 25.7 KB, free 364.6 MB)
Finished task 0.0 in stage 9.0 (TID 205). 4013 bytes result sent to driver
```

For 3 workers:

```
17/11/25 16:47:36 INFO MemoryStore: Block broadcast_7_piece0 stored as bytes in memory (estimated size 11.2 KB, free 366.3 MB)
17/11/25 16:47:38 INFO Executor: Finished task 0.0 in stage 3.0 (TID 3). 2883 bytes result sent to driver
```

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:

```
17/11/25 16:47:39 INFO Executor: Finished task 6.0 i
17/11/25 16:47:39 INFO Executor: Finished task 23.0
17/11/25 16:47:39 INFO Executor: Finished task 26.0
17/11/25 16:47:39 INFO Executor: Finished task 16.0
```

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:

Winnipeg	2017	0.012894	0.039385	0.028413	0.007981	0.029877
Inuvik	2013	0.02306	0.04583	0.034654	0.006949	0.03287
Saskatoon	2014	0.021946	0.027972	0.025215	0.001929	0.025051
Winnipeg	2013	0.01779	0.059325	0.035264	0.008992	0.035654
Quebec Ci	2015	0.013091	0.054384	0.03002	0.009324	0.026224
Toronto	2015	0.019737	0.05841	0.03253	0.008529	0.02909
Halifax	2014	0.010372	0.045545	0.031647	0.008075	0.035442
Inuvik	2016	0.022501	0.047701	0.033088	0.008061	0.028568
Digby	2016	0.016888	0.046437	0.029059	0.007373	0.027271
Moncton	2015	0.022086	0.505067	0.041737	0.073601	0.026687
Moosonee	2015	0.023146	0.046816	0.031924	0.0075	0.027804
Quebec Ci	2010	0.021949	0.053726	0.032094	0.006797	0.029634
Iqaluit	2016	0.004202	0.03473	0.006345	0.00423	0.00521
Coral Harb	2015	0.005483	0.146528	0.041899	0.032665	0.03046
RPB	2017	0.019757	0.041683	0.030095	0.007895	0.026212
Toronto	2009	0.020999	0.070343	0.031878	0.01104	0.028844
Coral Harb	2016	0.010692	0.137449	0.038933	0.026334	0.032397
Moncton	2016	0.02256	0.047133	0.031985	0.007679	0.02788
Moncton	2017	0.024259	0.05887	0.038728	0.012527	0.042818
Montreal	2009	0.019071	0.059454	0.031411	0.008417	0.030154
Montreal	2010	0.02038	0.045422	0.031716	0.006705	0.029695
Montreal	2011	0.019943	0.047609	0.030869	0.007409	0.029634
Montreal	2012	0.019037	0.050955	0.031824	0.0078	0.028921
Montreal	2013	0.020138	0.068425	0.032352	0.009191	0.030538
Montreal	2014	0.019677	0.047425	0.030626	0.007324	0.028048
Montreal	2015	0.019873	0.046298	0.031198	0.007465	0.027573
Montreal	2016	0.018379	0.056021	0.032485	0.008759	0.031539
Montreal	2017	0.012971	0.054468	0.034181	0.012614	0.031975
Moosonee	2009	0.019639	0.064352	0.030452	0.009134	0.026125
Moosonee	2010	0.022226	0.046052	0.033568	0.006702	0.032918
Moosonee	2011	0.021824	0.053053	0.03313	0.007738	0.03133
Moosonee	2012	0.02152	0.054256	0.032921	0.008213	0.030138
Moosonee	2013	0.0199	0.147234	0.034571	0.017765	0.029288
Moosonee	2014	0.022184	0.045323	0.032216	0.007174	0.029782
Moosonee	2015	0.023146	0.046816	0.031924	0.0075	0.027804
Moosonee	2016	0.023263	0.070268	0.033363	0.009454	0.029269
Moosonee	2017	0.0249	0.047734	0.034524	0.009585	0.028538
Ottawa	2009	0.019449	0.065883	0.030208	0.008781	0.026945
Ottawa	2010	0.014308	0.044299	0.02984	0.006897	0.028568

Full results inside the .csv file within the "results.csv" directory.