CS553 – Cloud Computing. Homework 5

Authors:

Luis Cavanillas (A20474430)
Jorge Cervera (A20474346)
Jorge Gonzalez (A20474413)

Semester: Spring 2021

Problem Statement:

In this assignment, a Hadoop and Spark codes to sort small and large files with different approaches have been created. To that purpose, KVM and QEMU have been used to deploy some Virtual machines in the Chameleon instance. Then, a Network File System has been configured and a HDFS created to be able to successfully run the Hadoop and Spark.

Proposed Solutions:

To tackle the sorting problem, one Hadoop and one Spark sorting Java program will be developed. Results will be compared at the end.

Hadoop

Two of the three components of Hadoop make it a solution for these kinds of problems. The first of them, *Hadoop Distributed File System* (HDFS) is responsible for storing the files. The second component is *MapReduce*, which is responsible for processing the files.

In order to store a file in the system, HDFS breaks it in N different parts, stores them inside Data Nodes and saves the metadata about them inside the Name Node. This is all the task of HDFS.

In order to process a file, MapReduce comes into the picture: applying a certain piece of code on a single file (input file) results in applying it to its different N splitted chunks (input splits) inside the Data Nodes. This process is called *Map*. Therefore, in Hadoop, the number of mappers for an input file are equal to the number of input splits of this input file.

Whenever the input splits have been processed, they have to be merged in order to generate the desired output, the processed file. This function of reducing multiple outputs to a single one is a process called *Reducer*.

Map and *Reduce* are the two different phases of the MapReduce component of Hadoop.

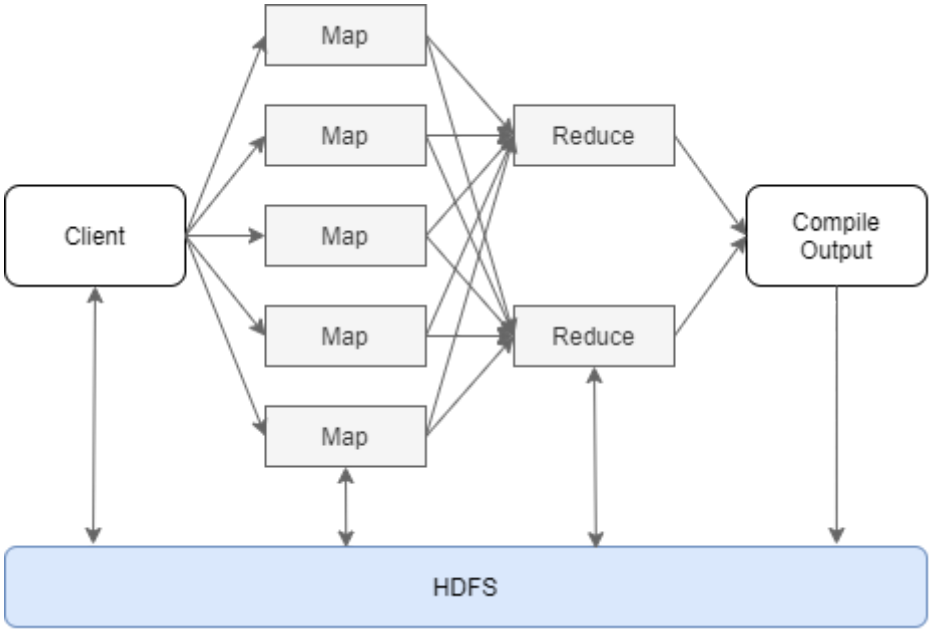


Figure 1. MapReduce component in Hadoop.

Mappers do not run directly on the input chunks of the file. This is because the input splits contain text (bytes) and Mappers work with *key-value* pairs. In Hadoop, each line of the file is called "*record*", and they are converted to key and value using *Record Readers*. By default, a file has a *TextInputFormat*, thus a Record Reader will read one line at a time and convert it to a key-value pair as defined by the user. A Mapper will only run once for each of these pairs.

Map function

As explained above, the map function in the Java code will only run once for every key-value pair on every input subfile. Therefore, the main objective of this method is to format each line of the subfiles for the *shuffling and sorting* automated process.

```
public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {

    tLine.set(value.toString() + "\r\n");
    context.write(tLine, NullWritable.get());
}
```

Figure 2. Map function in the Java code.

As shown in Figure 2, the output of the Mappers will be a key-value pair created as:

- key = line
- value = null

This means that all pairs will be created with a line as the key, and empty value. The reason behind this formatting is explained below.

Shuffling and Sorting

The Mapper provides output corresponding to each key-value pair provided by the Record Reader. This is called the *Intermediate Data* and will be passed to the Reducer after two more stages: *Shuffling and Sorting*.

The Shuffling phase takes place first. Here, all values associated with a unique key are combined.

The Sorting phase will be the last step before the Reducer receives the resultant output. Here, all key-value pairs are sorted automatically BY KEY. This is a huge advantage for the current task as no sorting algorithms are required to be implemented. Hadoop sorting is an automated process.

As shown in Figure 2, all key-value pairs will be sorted by key (a file line) automatically, so no further coding is required.

Reduce function

Finally, all the sorted output key-value pairs are sent to the Reducer. As explained before, this function will produce the final result with the processed data (the sorted file).

```
public void reduce(Text key, Iterable<NullWritable> values, Context context) throws IOException,
InterruptedException {
    context.write(key, result);
}
```

Figure 3. Reduce function in the Java code.

The Reduce function shown in Figure 3 does not require any processing steps as the desired output (sorted pairs) has already been done in the previous step.

Spark

Whereas Hadoop reads and writes files to HDFS, Spark processes data in RAM using a *RDD* (Resilient Distributed Dataset). In this case, Spark is running with the Hadoop cluster serving as the data source.

One of the main advantages of Spark is that, while Hadoop MapReduce only processes data in batch mode, this can process real-time data from real time events.

```
JavaSparkContext sparkctx = new JavaSparkContext(Config);
JavaRDD < String > contents = sparkctx.textFile(args[0], 1);
```

Figure 4. Loading the input file (as the first command line argument) into memory (JavaRDD object) from Java using Spark.

The first step to take on the Java program is to load the input file into a JavaRDD object, this is shown in Figure 4. The file loaded into memory is defined as the first command line argument when calling the resulting binary.

```
JavaPairRDD < String, String > words = contents.mapToPair(new PairFunction < String, String, String > () {
  @Override
  public Tuple2 < String, String > call(String s) throws IOException {
    return new Tuple2 < String, String > (s.substring(0, 10), s.substring(10) + "\r");
  }
});
```

Figure 5. Key-Value pair (Tuple) formatting from the file contents into a JavaPairRDD object.

Once the file is loaded into memory, a similar approach to the Hadoop MapReduce has been taken: The file contents are mapped into value-pairs using JavaPairRDD objects.

Considering the input files are generated using GenSort or TeraSort, one can certainly know that the first 10 characters of each line are the only ones required to be sorted. Therefore, the pairs' keys formatting problem will be tackled the same way as the Hadoop MapReduce: they will be formed by the first 10 characters of each line. Their corresponding values will be the following characters from the line. This is shown in Figure 5.

```
JavaPairRDD < String, String > sorted = words.sortByKey(true, 1);
```

Figure 6. Key-Value pair sorting by key.

After the pairs are generated, they will be sorted by key (following the Hadoop approach). This is shown in Figure 6.

```
JavaRDD < String > result = sorted.map(new Function < Tuple2 < String, String > , String > () {
  @Override
  public String call(Tuple2 < String, String > tuple) throws IOException {
    return tuple._1() + tuple._2();
  }
});
```

Figure 7. Key-Value ordered pair formatting into desired output file (JavaRDD object).

Finally, once the pairs are sorted, the output JavaRDD object has to be generated in order to create the sorted file. This will be done by concatenating the key and value ordered pairs. At the end of this process, the output file will be generated by saving the JavaRDD object data into a text file using the *saveAsTextFile()* function. This process is shown in Figure 7.

System set up

Several virtual machines have been deployed in the instance using Quemu-KVM.

These virtual machines will be the name nodes and datanodes in the executed experiments.

- 1 small instance and 1 large instance
- 1 small instance and 1 large instance
- 1 small instance and 6 small instances

These virtual machines have been configured as follows:

- Small instances: 4-cores, 16gb RAM, 38gb disk.
- Large instances: 24cores, 96GB RAM, 228gb disk.

Considering the previous configurations of the virtual machines, Hadoop and Spark environments have been set up as described before.

To generate the data for the hadoop and spark evaluations, instead of gensort, the command terasort has been used. This way, the data is generated directly in the HDFS system without the need to create it locally and then copy it. The following command has been used:

```
bin/hadoop jar hadoop-examples*.jar teragen 10000000 /home/input/1gb
```

Once the data is generated, a bash script is run to get the Hadoop or the Spark code running while monitoring the time taken, cpu and memory usage with the command psrecord. To that purpose, the following commands are use for Hadoop:

```
bin/hadoop com.sun.tools.javac.Main HadoopSort.java

jar cf HadoopSort.jar HadoopSort*.class

bin/hadoop jar HadoopSort.jar HadoopSort /home/input/32gb /home/output
```

and Spark:

```
javac -cp jars/spark-core_2.12-3.0.0-preview2.jar:jars/scala-library-2.12.10.jar:
jars/scala-compiler-2.12.10.jar SparkSort.java

jar cvf SparkSort.jar SparkSort*.class

bin/spark-submit --class SparkSort --master yarn --deploy-mode cluster
--driver-memory n1g --executor-memory n2g --executor-cores n3 SparkSort.jar
/home/input/xgb /home/output
```

At the same time, the arguments used in the Spark sort have been tuned with the following results.

- Number of cores: 1 (small instance) and 4 (large instance and 6 small instances).
- Memory: 2 (small instance) and 7 (large instance and 6 small instances).

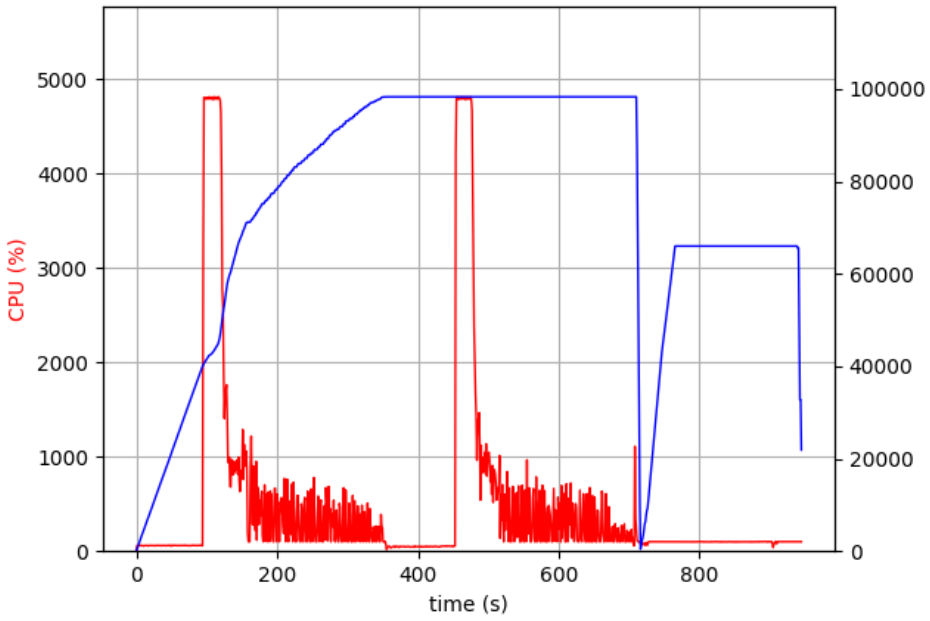
Solutions:

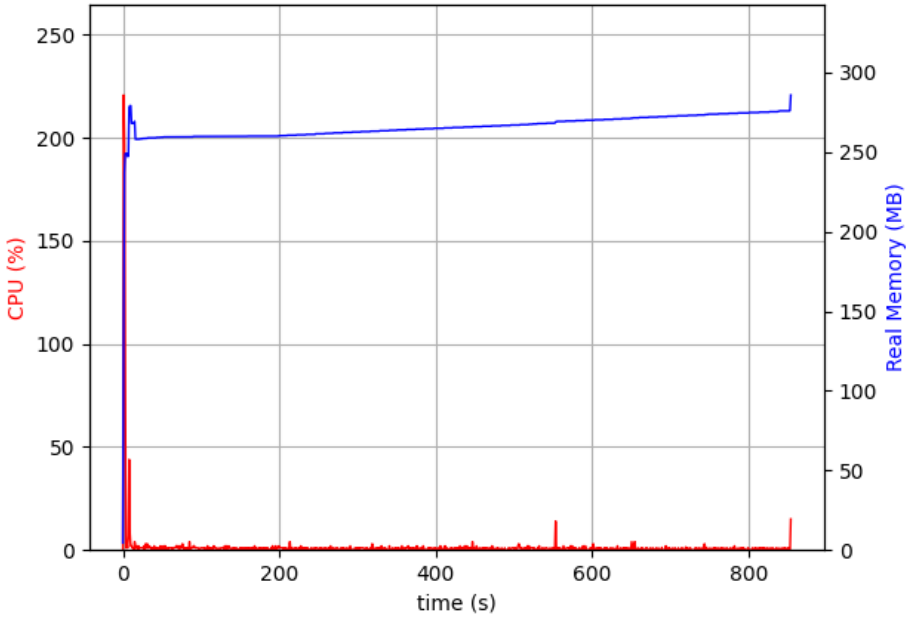
Many issues have been faced when evaluating the different sorts and configurations. The main problem and also the reason why the 64 GB files could not be sorted is that the disk capacity of the instance gets filled up and the virtual machines crash, turning their state to pause and with no chances of turning them back to a running state. Therefore, the Shared Memory and Linux Sorts have been tested with the 64 GB files but the Hadoop and Spark sorts have been tested with 32 GB files to make it work and get some results, although Spark has also crashed with the 32 GB file and the Shared Memory sort with the 64 GB dataset.

Experiment	Shared Memory	Linux	Hadoop	Spark
1 small.instance, 1GB dataset	43906	18039	42613	135254
1 small.instance, 4GB dataset	191110	66112	115601	373658
1 small.instance, 16GB dataset	1307485	418987	509703	1350689
1 large.instance, 1GB dataset	21597	9915	40488	124116
1 large.instance, 4GB dataset	95034	41296	102395	346683
1 large.instance, 16GB dataset	475902	191908	434706	1203708
1 large.instance, 64GB dataset	-	944486	853828*	-
6 small.instance, 1GB dataset	N/A	N/A	31144	88129
6 small.instance, 4GB dataset	N/A	N/A	74414	312612
6 small.instance, 16GB dataset	N/A	N/A	366556	1096518
6 small.instance, 64GB dataset	N/A	N/A	-	-

* 32 GB file instead of 64 GB file

And the following graphs have been obtained for the Linux Sort and Hadoop sort respectively:





What conclusions can you draw?

The Linux sort is the best sorting algorithm out of them all. It reduces the time taken to sort the files, in every scenario, up to two times. At the same time, hadoop performs better than spark but nos as good as the Linux sort.

Which seems to be best at 1 node scale (1 large.instance)?

With just 1 node (1 large.instance) it seems that the best algorithm is the Linux sort.

Is there a difference between 1 small.instance and 1 large.instance?

In the Shared memory and Linux algorithms there is a clear improvement when the instance is large than when it is small. However, for Hadoop and Spark, the differences are not too much as even though the large instance has more resources, they get better splitting the job in many different datanodes to lighten and speed up the job.

How about 6 nodes (6 small.instance)?

In this case, the differences are more noticeable. The time taken to sort the files decreases when the number of instances working as datanodes increase. Especially in the Spark sort.

What speedup do you achieve with strong scaling between 1 to 6 nodes? What speedup do you achieve with weak scaling between 1 to 6 nodes?

Strong scaling speedup: Speedup = t(1) / t(N), where t(1) is the amount of time needed to complete a serial task, and t(N) the amount of time to complete the same unit of work with N processing elements (parallel task).

The average speedup achieved with strong scaling in Hadoop is 1.437420154 and in Spark is 1.320601002.

Weak scaling speedup: Speedup = N * t(1) / t(N), where t(1) is the amount of time needed to complete a serial task, and t(N) the amount of time to complete the same unit of work with N processing elements (parallel task).

The average speedup achieved with weak scaling in Hadoop is 8.624520925 and in Spark is 7.923606014.

Experiment	Shared Memory	Linux	Hadoop	Spark
1 small.instance, 1GB dataset	43906	18039	42613	135254
1 small.instance, 4GB dataset	191110	66112	115601	373658
1 small.instance, 16GB dataset	1307485	418987	509703	1350689
1 large.instance, 1GB dataset	21597	9915	40488	124116
1 large.instance, 4GB dataset	95034	41296	102395	346683
1 large.instance, 16GB dataset	475902	191908	434706	1203708
1 large.instance, 64GB dataset	-	944486	853828*	-
6 small.instance, 1GB dataset	N/A	N/A	31144	88129
6 small.instance, 4GB dataset	N/A	N/A	74414	312612
6 small.instance, 16GB dataset	N/A	N/A	366556	1096518
6 small.instance, 64GB dataset	N/A	N/A	-	-

How many small.instances do you need with Hadoop to achieve the same level of performance as your shared memory sort?

In the previous results we can see that 6 small instances needed 31.144 ms to complete the Hadoop Sort, while shared Memory sort takes 21.597 ms, both

experiments over the 1GB dataset. Assuming that large.instance resources are equal to 6 small instances resources, we can see that when we add 5 small instances more we go from 40.488ms to 366.566 so we will need 11 instances to achieve the 21.597 ms shared memory time.

How about how many small.instances do you need with Spark to achieve the same level of performance as you did with your shared memory sort?

In the previous results we can see that 6 small instances needed 88.129 ms to complete the Spark Sort, while shared Memory sort takes 21.597 ms, both experiments over the 1GB dataset. Assuming that large.instance resources are equal to 6 small instances resources, we can see that when we add 5 small instances more we go from 124.116ms to 366.566 so we will need 15 instances to achieve the 21.597ms shared memory time.

Can you draw any conclusions on the performance of the bare-metal instance performance from HW5 compared to the performance of your sort on a large instance through virtualization?

Our Shared memory sort has failed with the 64 GB file. However, getting the Linux results and analyzing the scenario, the following conclusion has been drawn. The performance is better as the large instance has 96 GB of memory and all the file fits in memory. Therefore, an internal sort can be used without the need of creating small chunks, writing them on disk and reading them again to not exceed the memory capacity (writing and reading on disk are very slow operations).

Can you predict which would be best if you had 100 small.instances? How about 1000?

Considering the reduction in time when the number of datanodes is increased, if 100 small instances are used, the Spark sort will return better results and if there were 1000 of them it will be even better.

Compare your results with those from the Sort Benchmark (http://sortbenchmark.org), specifically the winners in 2013 and 2014 who used Hadoop and Spark. Also, what can you learn from the CloudSort benchmark, a report can be found at (http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf)

As shown in Figure 8, the **Sort Benchmark** results were:

- Spark: 100TB in 1,406 seconds (4.27 TB/min), using 207 nodes with 32vCores each. This means a throughput of **0.67 GB/min per core.**

  Our Spark results over the 16 GB dataset: 16GB within 1,096,518ms with 6 instances of 4 cores each: **0.0365 GB/min per core.**

  Comparing both results we can see that our Spark experiment over the 16GB dataset got 18 times less throughput than Sort Benchmark Spark.

- Hadoop: 102.5TB in 4,328 seconds (1.42 TB/min), using 2100 nodes with 2 cores each. This means a throughput of **0.34 GB/min per core**.

  Our Hadoop results over the 16 GB dataset: 16GB within 366,556ms with 6 instances of 4 cores each: **0.191 GB/min per core.**

  Comparing both results we can see that our Hadoop experiment over the 16GB dataset got 1.78 times less throughput of Sort Benchmark Hadoop.

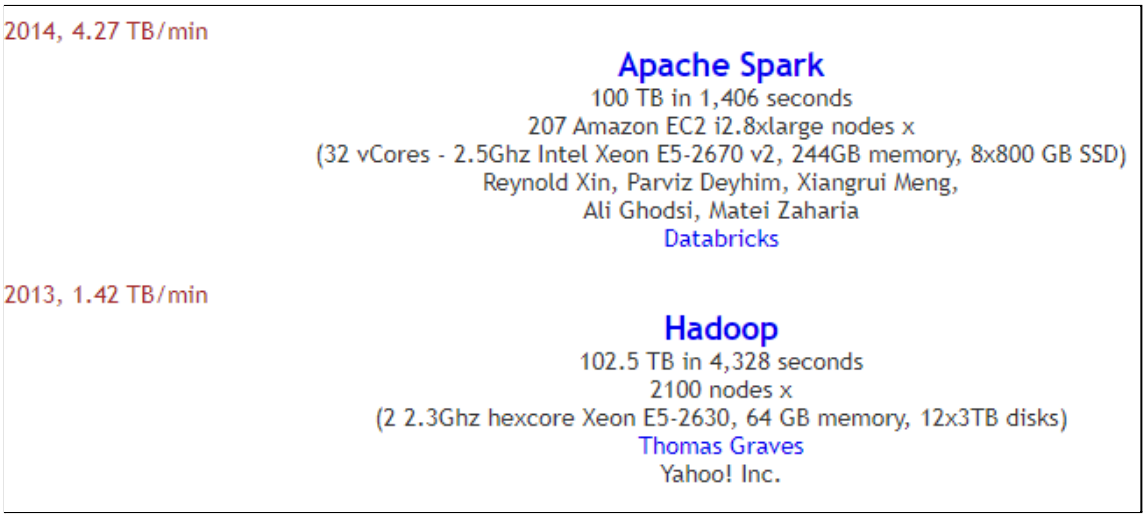


Figure 8. sortbenchmark.org winners for Apache Spark (2014) and Hadoop (2013) sorting