Hadoop & Spark
Project - 2

[Surabhi Agarwal]

[17203535]

An assessment component submitted in part fulfilment for the module

COMP30770

Programming for Big Data

UCD School of Computer Science

University College Dublin

February 2020

- # Introduction:

- This assignment is catered towards understanding Hadoop and Spark using practical tasks and jobs
- The Spark section of my report focuses towards performing different tasks on the given dataset using SQL and spark.
- The Graph processing section of the report focuses on uploading the dataset into Spark and HDFS and performing data processing jobs in order to comapare the two different systems on the bases of their performace.
- The final section of my report focuses on a research paper. The paper I read is about Google MapReduce and my views about the paper.

- # Spark

1. **Determining the number of records the dataset has in total:**
   Created a variable called records which loads the file in csv format, using blank space as the delimiter. Thereby, using the count function on the variable "records", we can get the total number of records in that dataset as shown below:

```
scala> val records = spark.read.format("com.databricks.spark.csv").option("header", "false").option("delimiter", " ").load("page_counts")
records: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 2 more fields]

scala> records.count()
res0: Long = 5046226
```

2. **Determining the record with the largest page size. If multiple records have the same size, listing all of them:**
   Firstly, I converted the csv file into an SQL table to perform the relevant SQL queries.
   Then applied the SQL query to select the records with largest page size. Then I printed the selected list of the desired records as shown below:

```
scala> val sql_records = records.registerTempTable("sql_records")
warning: there was one deprecation warning; re-run with -deprecation for details
sql_records: Unit = ()

scala> val question2_answer = spark.sql("select * from sql_records where _c3 in (select max(_c3) from sql_records)")
20/03/06 17:05:53 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the schema version 1.2.0
20/03/06 17:05:53 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
20/03/06 17:05:54 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
question2_answer: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 2 more fields]

scala> question2_answer.collect.foreach(println)
[en,Battle_of_al-Q%C4%81disiyyah,3,99999]
[en,DC_Super_Hero_Girls,7,99999]
[en,Old_Masters,1,99999]
[en,Patricia_Roc,5,99999]
[en,User_talk:5_albert_square,7,99999]
[en,Whitney:_The_Greatest_Hits,2,99999]
[ja,%E3%82%AB%E3%83%BC%E3%83%AB%E8%87%AA%E8%B5%B0%E8%87%BC%E7%A0%B2,5,99999]
[ru,%D0%A5%D0%B8%D1%89%D0%BD%D1%8B%D0%B5,3,99999]
```

3. **Computing the total number of page views (hits) for each project (as the schema shows, the first field of each record contains the project code):**

   Firstly, using select operation, I selected the first field (_c0) and the sum of the total page views (sum(_c2) for that particular project. Then I printed the list, note that the whole list was too long so only a part of the output is displayed below:

```
scala> val question3_answer = spark.sql("select _c0, sum(_c2) from sql_records group by _c0")
question3_answer: org.apache.spark.sql.DataFrame = [_c0: string, sum(CAST(_c2 AS DOUBLE)): double]

scala> question3_answer.collect.foreach(println)
[cbk-zam,203.0]
[krc.v,1.0]
[mh.d,2.0]
[ro.b,36.0]
[sco.mw,369.0]
[sd.d,68.0]
[stq.mw,39.0]
[fr.n,520.0]
[kbd,188.0]
[co.b,5.0]
[cs.n,197.0]
[en,5310646.0]
[ja.n,159.0]
[lo.mw,263.0]
[nds.q,1.0]
[pi.d,2.0]
[rmy.mw,12.0]
[uk.v,1.0]
[zh-min-nan.mw,86.0]
[kn.mw,209.0]
[pi,124.0]
[ss,1436.0]
[ie.b,8.0]
[te.mw,248.0]
[cr,98.0]
```

4. **Determining the number of page titles that start with the article ”The”. Finding how many of those page titles are not part of the English project (Pages that are part of the English project have “en” as first field):**

   For the first part, I used the SQL query that returns the number of page titles that start with “The” by using ‘like’ operator. The output is as shown below:

```
scala> val question4_numPages = spark.sql("select count(*) from sql_records where _c1 like 'The%'")
question4_numPages: org.apache.spark.sql.DataFrame = [count(1): bigint]

scala> question4_numPages.collect.foreach(println)
[48684]
```

   For the second half of the question, I use a similar query in which I check that the page title starts which doesn’t have ‘en' using the ‘like’ operator.

```
scala> val question4_notEng = spark.sql("select count(*) from sql_records where _c1 like 'The%' and _c0 not like 'en'")
question4_notEng: org.apache.spark.sql.DataFrame = [count(1): bigint]

scala> question4_notEng.collect.foreach(println)
[11553]
```

5. **Determining the most frequently occurring page title term in this dataset:**

   1. Firstly, selecting the field that contains only the page title and storing it in a variable called ‘question5’.
   2. Then, I modified the variable ‘question5’ to a csv file excludng the header and saving it as ‘temporary.csv’.

3. Further, converted ‘temporary.csv’ into a text file and saved it in ‘text’.
4. The I applied the word count technique MapAndReduce to count the number occurrences of each word in the file.
5. Further, swapped key and value for each pair and saved into variable called swap_wc
6. Then, sorted the keys in swap_wc so that the highest occuring words appeared at the top and saved the result in a variable called high_frequency.
7. Took the first line of high_frequency which has the highest occuring word along with it’s frequency.
8. This gives us the output (118, water) as shown below in the code snippet:

```
scala> val question5 = spark.sql("select _c1 from sql_records")
question5: org.apache.spark.sql.DataFrame = [_c1: string]

scala> question5.write.format("com.databricks.spark.csv").option("header", "false").save("temporary2.csv")

scala> val text = sc.textFile("temporary2.csv")
text: org.apache.spark.rdd.RDD[String] = temporary2.csv MapPartitionsRDD[79] at textFile at <console>:24

scala> val wc = text.flatMap(record => record.split(" ")).map(curr_word => (curr_word,1)).reduceByKey(_ + _)
wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[82] at reduceByKey at <console>:25

scala> val wc = text.flatMap(curr_record => curr_record.split(" ")).map(curr_word => (curr_word,1)).reduceByKey(_ + _)
wc: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[85] at reduceByKey at <console>:25

scala> val swap_wc = wc.map(_.swap)
swap_wc: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[86] at map at <console>:25

scala> val high_frequency = swap_wc.sortByKey(false, 1)
high_frequency: org.apache.spark.rdd.RDD[(Int, String)] = ShuffledRDD[87] at sortByKey at <console>:25

scala> high_frequency.take(1)
res13: Array[(Int, String)] = Array((118,water))
```

- # Graph Processing

1. Uploading the dataset in HDFS:

```
Start one or more stopped containers
(base) Surabhis-MacBook-Air:~ surabhiagarwal$ docker start myserver
myserver
(base) Surabhis-MacBook-Air:~ surabhiagarwal$ docker exec -it namenode bash
root@64be7d9cd5fd:/# ls
4300-0.txt  boot            entrypoint.sh  hadoop-data  lib64  opt   run     srv  usr
KEYS        cit-Patents.txt  etc            home         media  proc  run.sh  sys  var
bin         dev              hadoop         lib          mnt    root  sbin    tmp
root@64be7d9cd5fd:/# wget http://snap.stanford.edu/data/cit-Patents.txt.gz
--2020-03-21 14:04:53--  http://snap.stanford.edu/data/cit-Patents.txt.gz
Resolving snap.stanford.edu (snap.stanford.edu)... 171.64.75.80
Connecting to snap.stanford.edu (snap.stanford.edu)|171.64.75.80|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 85139832 (81M) [application/x-gzip]
Saving to: 'cit-Patents.txt.gz'

cit-Patents.txt.gz       100%[=================================>]  81.20M  2.62MB/s    in 30s

2020-03-21 14:05:23 (2.75 MB/s) - 'cit-Patents.txt.gz' saved [85139832/85139832]

root@64be7d9cd5fd:/# gunzip cit-Patents.txt.gz
root@64be7d9cd5fd:/# hdfs dfs -mkdir /cit-Patents
root@64be7d9cd5fd:/# hdfs dfs -copyFromLocal cit-Patents.txt /cit-Patents
2020-03-21 14:15:36,480 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2020-03-21 14:15:38,240 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
2020-03-21 14:15:40,721 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
```

Th above commands show how I uploaded the dataset into HDFS by mainly logging into the namenode container, then downloading the dataset using wget command, unzipping it using gunzip and then finally copying it into the HDFS using copyFromLocal.

2. **MapReduce: A MapReduce job in Hadoop that computes the significance of patents. Output the top ten patents (with largest significance) and their significance.**

   With the help of the word count example given in the Hadoop documentation and the explanation in the PDF of questions, I was able to solve this task with MapReduce jobs.
   The file Question2_MapReduce.java contains the code solution for this task.

   I have made use of 2 Mapper jobs and 2 Reducer jobs as shown in my code script, the final output I got was from the second reducer.

   In order to make a jar file of my source code and run it in HDFS, I used the following commands:
   //Logging into the namenode container:

- `docker exec -it namenode bash`

   //Exporting the java tools:

- `export HADOOP_CLASSPATH=/usr/lib/jvm/java-1.8.0-openjdk-amd64/lib/tools.jar`

   //Compiling the java file to generate the classes:

- `hadoop com.sun.tools.javac.Main Question2_MapReduce.java`

   //Creating the jar file:

- `jar cf Question2_MapReduce.jar Question2_MapReduce*.class`

   //Executing the MapReduce jobs and saving the final output to output2/directory:

- `hadoop jar Question2_MapReduce.jar MapReduce /input/output1/output2`

   //Now in order to read the top ten patents and their significance, the following command which uses a pipe and passes the output to the sort function which sorts according to the second column in the reverse order. Then we read the first then lines of the sorted text:

- `hdfs dfs -cat /output2/part-r-00000 | sort —k 2 -n —reverse | head -n 10`

```
2020-03-22 15:06:31,245 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
4723129 779
4463359 716
4740796 678
4345262 658
4558333 654
4313124 633
4683195 630
4459600 613
4683202 605
3953566 411
```

3. **GraphX: A GraphX program to compute the significance of patents and output the top ten patents**

- For this task, firstly I started the spark master container using docker start spark-master then I downloaded the dataset into spark using the following commands:

```
bash-5.0# wget http://snap.stanford.edu/data/cit-Patents.txt.gz
Connecting to snap.stanford.edu (171.64.75.80:80)
cit-Patents.txt.gz   100% |********************************************************************************
bash-5.0# gunzip cit-Patents.txt.gz
```

- Since, GraphX is Spark based and runs in memory and the file is too large, I only use the first 1000000 lines of the file for faster performance. If I use the entire file then spark crashes.
  So using the head command, I read the first 1000000 lines of cit-Patents.txt and generate another smaller dataset to work with.
- Then, I started the spark shell

```
bash-5.0# head -n 1000000 cit-Patents.txt > cit-Patents_1m.txt
bash-5.0# /spark/bin/spark-shell
```

- The following program uses GraphX to compute the significance of patents.
- Loading all the important spark libraries
- Loading the graph using GraphLoader and passing the file as input, this loads the graph and creates the edges
- Calculating the number of vertices and the edges
- Finally computing the top ten patents by sorting them according to their citation occurences.

```
scala> import org.apache.spark._
import org.apache.spark._

scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._

scala> val graph = GraphLoader.edgeListFile(sc, "cit-Patents_1m.txt")
graph: org.apache.spark.graphx.Graph[Int,Int] = org.apache.spark.graphx.impl.GraphImpl@4b0393ee

scala> val vertexCount = graph.numVertices
vertexCount: Long = 778090

scala> vertices.count()
res3: Long = 778090

scala> val edgeCount = graph.numEdges
edgeCount: Long = 999996

scala> graph.inDegrees.sortBy(_._2, ascending=false).take(10).foreach(println)
(3658634,57)
(2813048,52)
(3738339,50)
(3760171,48)
(3717609,46)
(3827237,43)
(3610799,41)
(3706549,39)
(3738341,38)
(3773894,37)
```

4. A short paragraph to compare the two systems in terms of performance and usability.

The HDFS approach is based on Apache Hadoop and the GraphX approach is based on Apache Spark in the above question. Both the systems are similar in a way that they both store data distributed across a cluster of servers or nodes.

However, while using GraphX (Spark), data is stored in RAM and extended RDDs are used, therefore this approach is much faster in terms of performance. A downside over here is that the size of the dataset matters a lot because we can be limited in terms of the memory and may have to decrease the size of the dataset as shown in the above task.

On the other hand, Hadoop file systems stores data stores on the data on the disk, therefore taking much longer to complete the MapReduce jobs. However, in this situation, we are not depending on the RAM so the jobs will get completed even for larger datasets which is a plus point for HDFS when it comes to scalability. The paper I've read in the report also throws light on the scalability factor of using MapReduce.

In terms of usuability, it is much easier to use Spark over MapReduce since we don't need to write separate Map and Reduce jobs in order to complete the task, we just need to load and construct the vertices and edges which makes it much simpler.

- # Reflection: Research Report

MapReduce: Simplified data processing on large clusters

Motivation for the research: Problems faced

In recent years, companies and organizations have had to gain an understanding of how to handle enormous amounts of data, and furthermore, how to derive insights from them in a timely manner. Parallelizing the task so that subtasks are handled by other processors or computers is one of the common approaches to deal with this issue. This can be a powerful technique, however, the task of parallelizing can be daunting and time consuming for people who are not trained in programming parallel and/or distributed systems. Also, handling all the implementation details could take more time than the time saved by running calculations on multiple machines or processors.

Solving the problem using MapReduce

In order to solve the above problems, the authors of this paper introduced MapReduce, which is a programming model that is able to automatically handle many of the challenges of parallel and distributed computing, such as data partitioning, inter-machine communication, and failure tolerance while achieving high performance. The MapReduce framework hides the details of parallelizing your workflow, fault-tolerance, distributing data to workers, and load balancing behind the abstractions map and reduce. The user of MapReduce is responsible for writing these map and reduce functions, while the MapReduce library is responsible for executing that program in a distributed environment.
Their implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable. Not only that, a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use as well as hundreds of MapReduce programs have been implemented and about a thousand of MapReduce jobs are executed on Google's clusters every day.

Outlining the Programming model of MapReduce

As discussed above, the computation to be performed is expressed through two functions: map and reduce. The map function takes an input key, value pair and outputs an intermediate key, value pair, while the reduce function accepts the intermediate key and a set of values for that key. The reduce function merges together these values to form a possibly smaller set of values. In effect, the map and reduce functions mimic those of the Lisp programming language where map is responsible for applying a function to a list of elements and reduce is responsible for merging a list of elements together. The difference with the MapReduce framework is it can handle lists that are too large to fit on a single machine.

Scientific discussions: Implementation and Execution of MapReduce Improvements

Many different implementations of the MapReduce interface are possible. However, at Google large clusters of commodity PC's are connected together with switched ethernet. In the Google MapReduce environment, Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine, Commodity networking hardware is used, a cluster consists of hundreds or thousands of ma-chines, and therefore machine failures are common, storage is provided by inexpensive IDE disks attached directly to individual machines and finally users submit jobs to a scheduling system.

In terms of the execution of the program, the framwork partitions the input data into pieces such that it is split to be processed by different machines. Multiple copies of the program is run on different computers in the network, one of which is a special master computer that schedules and assigns map and reduce tasks to idle worker PCs. The workers then compute their map functions and return results partitioned into a user-specified number of partitions. Finally the partitions are sorted on a reduce worker PC

and the results are computed for each of the unique intermediate keys. When all of the results are computed, they are appended to an output file.
The MapReduce framework handles errors by restarting worker machines. Since the results of a map worker are stored on local disk, if that machine goes down the results are lost. Therefore, the master is responsible for scheduling that piece of work on a new worker machine.

Strengths & Contributions of the paper
The main strength of this paper is that it introduced a new method for automatically handling the details behind running computations on a parallel and/or distributed system. As the results show, the system works as intended on Google's compute cluster, and is quite robust to failure, despite their efforts to stress test the system by intentionally killing large numbers of worker processes, etc. The widespread adoption in Google for their computational tasks at the time this paper was published speaks to its success in increasing access of parallel and distributed computing to everyone. Given how widely known and used it has become since its initial introduction, it is safe to say the MapReduce has been an important contribution in its field.

One weakness of the implementation of MapReduce in this paper is how it does not handle cyclic computations, where the data stays in place and is worked on by various operators repeatedly, such as parameter optimization in machine learning. Doing this would require MapReduce to be repeatedly called, potentially reducing performance.

Other related work
There are many systems that have provided restricted programming models and used the restrictions to parallelize the computation automatically. The authors of the paper throw light on some of the approaches similar to MapReduce. There are sveral components of the MapReduce approach which has been compared to different systems and how MapReduce was inspired them.
Bulk Synchronous Programming and some MPI primitives provide higher level abstractions that make it easier for programmers to write parallel programs. The main difference between these sytems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.
The locality optimization of MapReduce draws its inspriation from techniques like active disks, wherein computation is pushed into processing elements that are close to local disks in order to reduce the amount of data sent across I/O subsystems or the network.
The Charlotte System is another example which has a eager scheduling mechanism similar to the backup task mechanism of MapReduce. One shortcoming of the eager system in Charlotte is the failure of the entire computation given that a task causes repeated failures. This is fixed in the MapReduce approch by skipping bad records.
Another example quoted in the paper is the programming model of River where processes communicate with each other by sendig data over distributed queues. Though it is similar to MapReduce in a way that it tries to provide good average case performace even in the presence of non-uniformities introduced by hadrware, it differs from MapReduce in it's approach of doing so. Unlike River, MapReduce restricts the programming model by partitioning the problem into larger number of fine grained tasks.
Other examples such as BAD-FS and TACC have also been mentioned by the authors.

Conclusions & Thoughts

The authors of the paper do a great job in explaining different aspects of Google MapReduce by explaining the data processing in simple steps, the

master/worker architecture as well as addressing challenges of their approach. I also liked the way the authors compared each and every segment of MapReduce with other systems and programming models in the industry.

Overall, I found the paper easy to read and gained insights about what MapReduce is and how it works. I found the structure of the paper very similar to the last paper I read about the Google Bigtable which made me realise that the authors at Google work on real-world, large scale and distribute data processing problems for cutting edge technology. This demonstrates the usefulness and scalability of their products.

• Conclusion

- I found this project an interesting way to learn about Spark, HDFS and GraphX
- All the 3 sections were linked together in a clear way so that we could get an understanding of how things work.
- The research paper also made my understanding of MapReduce more clear.
- Finally, we can conclude that Spark is an upcoming cluser computing software which has faster performace, however MapReduce and HDFS are great tools when it comes to scalability and dealing with larger datasets.