COMP 47470

Big Data Programming

Project 2: Hadoop & Spark

Conor Murphy

20205251

30/03/2021

1 Hadoop

The first step was to send the data to the HDFS, the data is stored in directory called data. The data was moved from the NameNode to the DataNode using.

hdfs dfs -put shuttle.csv /data

it can be assumed form here that all MapReduce jobs will be run on this folder. For each MapReduce, the output was stored in a directory with the name of the corresponding question so for question 1 the output is sent to /q1 etc. all java Files are included in the Hadoop Folder.

Question 1

The first problem of the data set is that Thursday is represented by the name “thu” and “thur” to overcome this all instances of “thur” have been changed to “thu”, this was a personal preference, and the opposite could also have been done.

As only weekdays were requested the two days of the weekend were filtered, the title column was also filtered out as this was additional unneeded information. To achieve this all of the keys that were not needed were added to an array this was then used to filter out these values. The full code can be found in WeekdayCounter.java.

Answer: fri 99, mon 94, thu 219, tue 105, wed 126.

Question 2

Above is a snippet form the map function of RowCont.java. This was relatively straightforward; all keys were set to the same String in this case “row” as they are all added together in the reduce job this produces the number of rows. The output of the reduce function is. **Answer:** Row 1001

This can be confirmed as the number of input records is displayed after the function is run, the result of this is also. 1001, Then full code can be found in CountRows.java.

Question 3

```
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken().split(",")[3]);

    if(!word.toString().equals("passenger_count"))
        context.write(word, one);
}
```

Here the desired column is selected (4th one) the value of passengers is used as the key and each one is assigned the value of one, the title column is filtered out. The full code can be found in CommonPassangers.java. The most common is 1 passenger.

Answer

0	5
1	608
2	235
3	62
4	45
5	35
6	10

Question 4

This question was significantly harder than the prior ones as firstly the data had to be added to a separate string as more than one column was needed. The Thursday problem was once again encountered, and this was rectified the same as in question 1. The title column was also then removed. The total column which was the 16th position in this case was then used to compare and it was greater if it was then the day was sent to the reduce function. The full code is available in RidesOverTwenty.java.

Answer: fri 22, mon 16, sat 19, sun 59, thu 51, tue 19, wed 28

Question 5

I modified the code from question 1 and instead of displaying the trips per day I added term all to the on key called Trip which is then counted. Here if weekends were also needed, they could easily be added and the key set to "weekend" although the question specified weekdays, so they were excluded. The full code including the Array declaration of days to exclude is viewable in WeekdayTrips.java.

Answer: Trips 643

Question 6

Note answer is including 2pm

```
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken().split(",")[2]);
    if(word.toString().equals("tpep_journey_hr")){
        continue;
    }
    if(Integer.parseInt(word.toString().substring(0,2)) >= 14){
        word.set("Trips");
        context.write(word, one);
    }
}
```

Firstly, I got the third column, then excluded the column name in the if statement, I used the .substring method to get the first 2 numbers from the time field as they are all that was needed for comparison. They were then parsed to an int which allowed for a simple greater than comparison. Here I only printed the total number of trips as it was all the question asked for, but this could easily be changed to group them by day or to group them by the hour of the trip by using the time as the key in the Map function. **Answer**: Trips 413.

Question 7

This question required 2 MapReduce jobs and a bash script to tie it all together. The first Scrip just gets the number of rides per day and is a modified version of Q1 but this time including the weekend. The Second script is a bit more complicated.

```
while (itr.hasMoreTokens()) {
    String line = itr.nextToken();
    word.set(line.split(",")[1]);
    if(line.split(",")[1].equals("thur")){
        word.set("thu");
    }
    if(word.toString().equals("tpep_pickup_dow")){
        continue;
    }
```

```
    IntWritable num = new IntWritable(Integer.parseInt(line.split(",")[10]));
    context.write(word, num);
}
```

Shown above is the while loop in the map job. Here instead of just passing one in as the value alongside the day, the fare amount is passed in, this is achieved by declaring a new IntWritable object and assigning it the desired value in integer form. In the Reduce function instead of adding up all of the ones, it now adds up the total fare amounts.

A bash script was then used to take the two numeric columns and generate the average,

Bash unfortunate does not support floating point arithmetic operations, so the values are all integers even though most of them vary between 4 and 5.

```
#!/bin/bash
hdfs dfs -cat /q7_p1/* >  p1.txt
hdfs dfs -cat /q7_p2/* > p2.txt
X
for i in $(seq 1 7); do
        day=$(cut -f1 p1.txt | head -n $i | tail -n 1)
        trips=$(cut -f2 p1.txt | head -n $i | tail -n 1)
        total=$(cut -f2 p2.txt | head -n $i | tail -n 1)
        avg=$((total/trips))
        echo $day  $avg
done
```

The results of the two MapReduce jobs shown on the left with the final answer on the right .

	Num Rides	total amt		Avg
fri	99	449		4
mon	94	404		4
sat	100	446		4
sun	257	1218		4
thu	219	1025		4
tue	105	458		4
wed	126	637		5

2 Spark

Question 1

```
val inputFile = sc.textFile("asgn/netflix.csv")
inputFile.count
```

the file is loaded into the rdd and .count is called **Answer:** Long = 7788

Question 2

```
val data = spark.read.format("com.databricks.spark.csv").
option("header", "true").
option("inferSchema","true").
load("/asgn2/netflix.csv")

data.groupBy("type").count().show()
```

The Data is read into the dataFrame then grouping by the desired column and using the count method returns the desired output. **Answer:** TV shows 2410, Movies 537, so there are more movies

Question 3

```
data.registerTempTable("dataTable")
val sum = spark.sql("select type, SUM(duration) FROM dataTable Group By type"
)
sum.collect.foreach(println)
```

The same table defined in Q2 Is used here the table dataTable is registered and a simple SQL query with the SUM operator is used. **Answer:** [TV Show,4280] [Movie,533979]

Question 4

```
val oldest = spark.sql("select title, release_year
FROM dataTable
where release_year = (Select MIN(release_year) from dataTable where type = 'M
ovie')")
oldest.collect.foreach(println)
```

A SQL statement with a nested select can be used to get the oldest movie.

Answer: [Prelude to War,1942] [The Battle of Midway,1942]

Interestingly there is an older entry from the 1920's but it is a TV show, two movies are the oldest as they were both released in 1942. After a quick google search I can confirm that Prelude to war is the oldest film.

Question 5

```
val countries = inputFile.map(parseLine)
val filtered = countries.filter(x => x!="country")
val countryCount = filtered.flatMap(line => line.split(";")).
map(c => c.trim).
filter(c => c != "").
map(country => (country, 1)).
reduceByKey(_ + _);
```

There may look like a lot of map functions are going on here but all of them are necessary, the parseLine function is called which gets the desired column and returns the values. the title column is then filtered out, the filter Map method then separates all of the counties from the string. The trim function then removes

the leading whitespace as if was left the likes of “Germany” and “ Germany” would not be combined together. All of the empty gaps are then removed as initially 507 empty string were being included in the reduce. And finally, the county can be added to the tuple. The full code and output are available in q5.sc

Question 6

```
val descriptions = inputFile.map(parseLine)
val descriptionCount = descriptions.flatMap(line => line.split(" ")).
map(removeVals).
map(desc => (desc, 1)).
reduceByKey(_ + _);
```

This is very similar to the last question an essentially the same approach was adopted. Here all non-alpha numeric characters were removed in a separate function called removeVals. This did not need to be separated into a separate function , this is just a personal preference to make the code more readable. The full code is available in Q6.sc the 5 most common words are **Answer**: (of,4705), (and,5600), (to,5659), (the,7193) and (a,10144)

Question 7

A function called get decade is declared which contains an if statement that determines the decade the year is in, after this it is rather straight forward as it then maps and carries out a standard MapReduce job counting the number of each instance. The code is viewable in q7.sc. The top 5 decades are Answer (2010s,5711), (2020s,899), (2000s,728), (1990s,225), (1980s,106)

Question 8

Here a separate function called get Month is declared which split the date field and returns the first word which is the Month. From here it is once again a MapReduce job with the key being the month and the value being 1, the code for this question is available in q8.sc. there are 10 shows which have no date, these have been left in to preserve the total but could be removed just like the column name if it were needed Answer:

December	833
October	785
January	757
November	738
March	669
September	619
August	618
April	601
July	600
May	543
June	542
February	472
	10

Question 3.1

Before discussing the differences between Spark and Hadoop MapReduce it is helpful to look at both of them separately. This will then allow for a clear basis of comparison between the two.

Hadoop MapReduce was created in 2003 by engineers at Google and its main goal Is for large scale analysis using a distributed file system. One of the main features of Hadoop is this distributed file system which is called HDFS (Hadoop Distributed File System). This allows for the storage of files across thousands of machines (nodes) and for operations then to be carried out on these files stored across many machines. Take for example Yahoo! who had over 25000 servers that stored 25 petabytes of data [1].

HDFS is split into two components, these are the NameNode and the DataNode. The DataNodes store all of the data and the NameNode stored the metadata and location of this data. To understand this more it is helpful to know how data is stored in HDFS, when a new file is to be stored in HDFS the NameNode firstly partitions It in to blocks, which are generally 128 Mb's in size, but this can be changed. This block is then replicated three times, this is how HDFS deals with durability and utilises this approach over RAID, which is used by other distributed file systems like Lustre and PVFS [1]. This replication approach is one of the key principles of Hadoop as it Assumes that failures are common. The Name node never stores any of these blocks but instead will keep track of what DataNode they are stored on, this information is stored is in the form of inodes. It is important to note that the NameNode does not directly call DataNodes but instead communicates with it through heartbeats, every DataNode will send heartbeats to the NameNode to confirm that it is still active, if no heartbeat is received by the NameNode in 10 minutes then the NameNode will consider this DataNode to be out of service and will send the command to replicate the data that was stored on that data node elsewhere[1]. When a client wants to access the data it first gets the location of the relevant blocks from the name node and then will contact the DataNode directly and retrieve the relevant info. Hadoop can then retrieve data from the HDFS to perform MapReduce operations on it, the results of the reduce functions will then be written back to the HDFS where they can be viewed from. During Hadoop MapReduce jobs a scheduler will assign tasks as nodes become available for jobs, this node ideally also stores the data required for the task. There are three types of schedulers which are Fair, Capacity and Basic [4]. One final point of importance with Hadoop MapReduce is Speculative execution, this is when is when a job has been running for some time, but is taking longer than expected, a scheduler can then replicate this job on another node and which ever one finishes first will then be used, this can speed up running time by 40% but also may not be effective if the data distribution is skewed [3]

Spark is another application that offers distributed processing on large datasets. Spark was first developed in 2009 in UC Berkeley's AMPLab. Spark was developed to combat some of the main problems of Hadoop. It offers much faster speed than Hadoop with speeds being around 100 times faster. Sparks is very flexible, and programs can be written in Java, Scala Python, R and even SQL. Spark was started as a subproject of Hadoop but is not an extension if it, it can run using Hadoop but can also run entirely on its own. Spark can run on Hadoop in Three ways the first using HDFS for Storage, the second using both HDFS and Yarn and the final is running inside of MapReduce on top of HDFS. To Clarify, it does not need Hadoop to run and instead can use services like AWS's S3 or HBase for storage and run barebone for processing[5]. Spark not only increased speed but also offers advanced analytical capabilities, as it offers sparkSQL allowing for SQL queries to be run on large, distributed datasets. The increased speed from spark comes from the fact that spark stores intermediate processing data in memory allowing for much faster operations. To achieve this Spark uses a data structure know as an RDD.

RDD's (Resilient Distributed Dataset) are data structures that are fault tolerant and allow for intermediate results to be stored in memory which Is what gives Spark its speed. RDD's allow for coarse-grained transformations (apply to all objects at once)[2] like map filter and join. RDD's achieve fault tolerance by

logging the transformations to build a dataset know as its lineage rather the results of the actual transformation, then in the event that a partition of an RDD is lost, the transformation measures can quickly be recomputed resulting in the data being recovered. An RDD is a read only portioned collection of records that can be created using deterministic operations on data in stable storage like text files or on other RDD's[2]. Spark is a lazy evaluator meaning that when an operation is called in Spark it will not execute immediately instead the operation that is being carried out will be maintained using DAG (Directed Acyclic Graph), then only when the action is called the data is loaded and the lineage of operations can be worked through, this means that a lot of time is not wasted on unnecessary read and write operation as it can view all of the operations that will be carried out so the execution can be optimised[6]. The persist function can also be used in spark to signify that an RDD that we want to keep in memory and use again in subsequent operations i.e if you wanted to run multiple queries on the same dataset. The RDD data type clearly is a powerful tool for distributed analysis but does have instances where it may not be suitable, one for example is when an application needs to make asynchronous fine grained updates as RDD's are better at large batch operations to the entire dataset[2].

Clearly Both Spark and Hadoop are extremely useful tools for large scale distributed data analysis. The main differences between the two is how data Is stored in between jobs, Hadoop will write data to and from HDFS, this means that map reduce jobs that require a lot of operations can take longer due to the need to constantly read and write, Spark on the other hand will keep data in the form of RDD's in memory when permitted meaning this information can be used a lot faster as no read operation is required. The two systems also have different approaches to resilience, Hadoop utilises a three way replication across multiple nodes whereas Spark keeps track of what transformations were carried out allowing for lost data to be reproduced quickly.

Spark is generally much faster than Hadoop and has many instances where it should be used over it for example in large scale batch operations like MapReduce is needed then spark is a better choice. Spark also offers advanced stream and graph processing in the form of Spark streaming and Graph X so if theses operations are needed once again spark should be used. Spark also has fantastic support as it supports Scala, Java, Python and more so this flexibility over java with MapReduce might be the deciding factor for a small company deciding between the two. Additionally, the inclusion of spark SQL meaning that SQL queries can be used is also a big benefit as many companies already use SQL heavily so there is not a major learning curve [8].

From the above passage you may think that Hadoop is essential useless now that spark on the scene, this is not the case as there are many instances where Hadoop should still be used over Spark. Spark is a newer technology which means that expertise is not as available as it is for Hadoop which means that getting a sufficient expertise may be harder, the availability of outside consultancy knowledge would also be greater for Hadoop as it is an established technology. The cost of setting up both clusters can also differ as a Hadoop clusters will require more memory and a spark cluster will require more RAM making them more expensive. If Fault tolerance is a primary concern, then Hadoop may also be the better choice as it three way replication is more robust than Sparks RDD approach. If security is a primary concern Hadoop also wins as it is far more developed in this area with more inbuild security features[7].

Conclusion

There are some instances where Hadoop is preferable over Spark but generally spark is the more powerful tool which many companies are now adopting, Spark is the more advance technology with greater capabilities and outperforms Hadoop in most areas.

References

[1] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, USA, 2010, pp. 1-10, doi: 10.1109/MSST.2010.5496972

[2] Zaharia, Matei & Chowdhury, Mosharaf & Das, Tathagata & Dave, Ankur & Ma, Justin & McCauley, Murphy & Franklin, Michael & Shenker, Scott & Stoica, Ion. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. 2-2.

[3] https://data-flair.training/blogs/speculative-execution-in-hadoop-mapreduce/

[4] Lecture Notes – COMP47470 Lecture 6

[5]Lecture Notes – COMP47470 Lecture 8

[6]https://data-flair.training/blogs/apache-spark-lazy-evaluation/

[7]https://www.xplenty.com/blog/apache-spark-vs-hadoop-mapreduce/

[8]https://logz.io/blog/hadoop-vs-spark/

Question 3.2

The following analysis was done on the paper “Spark: Cluster Computing with Working Sets”. By Zaharia et al.

Questions and Background

The paper seeks to address the issues surrounding some of the pitfalls of Hadoop MapReduce in the area of Iterative Jobs and Interactive Analysis. In 2010 when the paper was published Hadoop had been the go to application for large scale data intensive operations. MapReduce introduced the model whereby data operations are carried out on multiple machines (nodes) by systems that provide Fault tolerance, job scheduling and load balancing. These systems are very good at doing a certain type of job, Zaharia claims that Map reduce performs poorly in two areas, these being.

- **Iterative Jobs**: for example, when optimising a machine learning model using logistic , the same function will be called repeatedly, each time this is done it is treated as a separate MapReduce jobs which mean that the data is being read every time, this can make the entire process excessively long.
- **Interactive Analysis**: When large queries are run on distributed dataset Hadoop will load the dataset every time the dataset it is used. Zaharia gives the example of running queries on a SQL interface like Pig or Hive where you are running multiple queries on the data set, every time a query is run the dataset gets reloaded to be used even though the same dataset is being used, this makes the execution time excessively long even in the tens of second’s range.

The purpose of the paper was to introduce a new technology, Spark that directly address the two problem areas stated above.

Solutions

The paper introduces a new application called Spark and introduces the idea of an RDD. Zahira first introduces the Scala language which is a statically typed high level programming language which runs on the java virtual machine. Spark is unique as they believe it to be the first language to allow users to declare RDD’s, variables, classes, and functions and to then run them in in parallel operations on a cluster.

The paper discuss how Spark provides abstractions for high level programming in two ways which are through RDD’s (Resilient Distributed Datasets) or using one of two restricted types of Shared Variables.

RDD’s

A resilient Distributed Dataset is a read only collection of objects that is stored across multiple machines. Zaharia claims the main benefit of an RDD is the fact that the elements of a RDD do not necessarily need to exist in physical storage, instead the steps to produce the data need only exist, this means that RDD’s are fault tolerant as if a RDD fails the information on how it was created can be used to recreate it on another node. This approach differs from a traditional Distributed shared memory system as they typically use checkpoints to achieve fault tolerance, meaning that in the event of a failure it will take time to roll back, this is not the case with RDD’s. The paper discusses the four ways that an RDD can be created which are, from a file, from a Scala collection, by transforming an existing RDD or by changing the persistence of an existing RDD.

Shared Variables

The second way in which spark provides abstractions for parallel programming is through shared variables, the first of which is broadcast variables. This is a large read only piece of data which can be distributed to nodes which will allow the contents to be reused multiple times while ensuring that each worker only gets the data once. The second is accumulators which are variables that can only workers can add to and only the driver can read, these can be used in the application of counters and in MapReduce jobs.

Experiments

After introducing the main concepts of Spark, and RDD's Zaharia et al then ran three experiment to tests Sparks capabilities against Hadoop MapReduce. These experiments were designed to specifically test the efficiency of Spark against the two problems areas mention in the introduction, these being Iterative Jobs and interactive analysis. The Three Experiments were:

- **Logistic Regression**: a 29GB dataset was used to carry out logistical regression, this is used in machine learning to determine an optimal model and it is an iterative approach where different values are plugged in to a model until the best performing value is found.
- **Alternative Least Squares**: This experiment tested sparks performance for predicting user ratings for unseen movies based on movies people have already rated. The Least squares algorithm is very CPU intensive as opposed to being data intensive which is where Spark performs best. In the experiment a broadcast variable was used to send a file, R containing know ratings for users which would then be used to predict future ratings. The experiment was run using 30 AWS EC2 clusters with 5000 movies and 15000 users.
- **Interactive Spark**: this experiment consisted of loading 39GB's of Wikipedia data into 15 EC2 instance, queries were then run on this dataset and the response time measured.

Results

The results of the three experiments were as follows.

- For the Logistic regression experiment Hadoop took 127s per iteration while the first spark iteration took 174 seconds, this means that for only one query Hadoop was considerable quicker. When increasing the number of iterations however, the time per iteration with Hadoop remains the same as data is read from the file every time but as spark can cache data, this resulted in every additional iteration for spark only being 6 seconds, which is up to 10x faster than Hadoop. Meaning for 10 iterations it takes Spark 228 seconds and Hadoop 1270 seconds.
- In the Alternative Least Squares experiment when the file R containing the necessary data for the computation was not stored as a broadcast variable the time to retrieve the data on each query took up the bulk of the time. when it was stored as a broadcast variable the performance was improved 2.8x.
- For the Interactive Spark Experiment The initial query took 35 seconds to run but after that each query took between 0.5 and 1 second, once again demonstrating the power of storing an RDD in memory.

Conclusion

Overall, I thoroughly enjoyed the paper, the objectives were clearly stated at the beginning and the experiments certainly addressed them. The experiments clearly showed the immense benefits of Spark over Hadoop and definitely highlighted the benefits of RDD's being stored in RAM resulting in comparatively faster execution in Iterative jobs and Interactive analysis. This Paper was written in 2010 which was still in the very early days of spark so to even see how some of the terminology has changed was interesting. To also see how this technology has been adapted and even now offers far more features in Spark3 was also pretty intriguing like DataFrames or stream and graph processing. From the paper It is clear that spark is here to stay while Hadoop is continuously looking to be past its prime.