School of Computer Science

COMP30770

Project 2
Spark

Name	Brian Byrne
Student Number	18391933
Date	07-01-2021
Number of Pages	7

This report details the work done for the COMP30770 Project Number 2. In this project, I worked with two datasets, a dataset of the 100 most starred GitHub Repositories which I used in task 1 and then a Big Dataset of authors who co-authored on academic papers, which I used in task2.

This report is structured into three sections. Task 1 focuses on using Scala and SQL to for queries to gain various insights from the dataset. Task 2 involves using object-orientated programming and RDD's with the GarphX package to form interactive graphs based on the co-authorship dataset and to use GraphX's built-in methods to gain more data insights.

1 Spark

As I mentioned earlier Task 1 involved using both Scala and SQL to create queries and derive information from the dataset. These queries are all included with comments in *task1.scala* script included. It had the following four questions:

1) For determining which project had the most stars, I simply created a temporary table using Scala and wrote an enclosed SQL statement called "maxStars" that selected the entire row, where the column stars is equivalent to the highest entry in the column.

2) The goal of this exercise to determine the total number of stars for each language. Using the value "numStars", I selected the columns language and sum(stars) from the same temporary table, using the SQL feature once again that I had created in the previous question and grouped these stars by language using the group by call.

3) Question 3 had two exercises.

   The first exercise consisted of finding the number of project descriptions that contain the word "*data*". Using the value "numDataOccurances", I counted the descriptions from the same temporary table, where the description was like '%data%'. This code displays the number of occurrences of the word. For purposes of this exercise I did not include any occurrences of the word "Data" as only "data" was mentioned in the question.

   The second exercise consisted of finding from data columns we had found earlier, the number of these rows that have their language value set (not empty/null). This simply involved repeating the SQL code from above and adding a Where element that includes only the rows that have one or more characters in its language field by using "CHAR_LENGTH(language)>0". This value is stored in "numDataOccurances".

4) The final exercise of Task 1 involved most frequently used word in the project descriptions. This was a slightly more complex solution than the previous four parts. I used SQL to isolate the description column from the temporary table. I proceeded to convert this value first to a CSV file, and then to a text file. Then I performed a MapReduce job to count all of the words in the text file, and then I swapped the amount of times each word occurred with the word, and preformed a sorting order so the first tuple would be the most occurring word. Now I could simply select the most occurring word tuple as it was the first in the RDD, the I parallelized the value so I could write it to the bash terminal from the script.

Answers to Questions:

Question 1:
[spark,Apache Spark - A unified analytics engine for large-scale data processing,Scala,28798]

Question 2:
[JavaScript,25344]
[Erlang,4820]
[null,30305]
[C++,23899]
[Julia,489]
[Jupyter Notebook,8819]
[C,2038]
[TypeScript,3274]
[HTML,383]
[Scala,59888]
[Clojure,774]
[Go,6558]
[Python,39761]
[Java,102331]

Question 3(a):
[25]

Question 3(b):

[24]

Question 4:
(34,Apache)

2 Graph Processing

Task 2 involves importing the graphX libraries and the second (co-authorship) dataset. It consists of 2 questions. This question is all included with comments in *task2.scala* script included.

1) The first question involves reading in the dataset and building a graph to represent the relationships between the authors in the dataset. After importing the various relevant packages, I created an object “Coauthorship” consisting of two authors names as strings. Then I created a function to parse the inputted file into the Coauthorship class, using str.split to divide the data by the comma in the CSV. Next, I read the csv file as an RDD and removed the header row from the data. I cached the current RDD into memory, before creating tuples and assigning an index to each individual author using flatMap. I generated the author ids using “.distinct.zipWithIndex”, so each author had a unique ID. Next I created a small map that maps each author name to its relevant id to retrieve author ids later on. I created links between the authors using a value called “routes”, making tuples of two authors names who are co-authors. I used the authorMap I had made earlier to linke the id tuples to these author’s names. I went onto assign a universal value “distance” equal to one for forming the graphx edges, so that they’ll all be equidistant, and a value nobody to act as the default vertex. I swapped the authors id and name tuples for the purpose of creating the edges, as the graphx edge format takes only longs as the positions of the vertexes. The tuples changed from format <name,id> to <id,name>. Now I was able to create the graphx edge by mapping the two ids in the authors by using the .get call to retrieve their ids. I then finally constructed the graph using the authors ID numbers as vertices, the edges as edges and the nobody value as the default vertices. I wrote a few sting conversion calls then, to simply display the first four vertices and edges to ensure my graph had been successfully constructed.

2) The second part of task 2 involved calculating the longest shortest part of the graph or the maximum Erdös number of authors. For this exercise I chose Paul Erdös as to use as the node ID in my shortest path call. I retrieved all the shortest paths in graph format from the graph using the imported package “ShortestPaths” . I then mapped the shortest paths lengths to “values” and

MapReduced the map to contain only the number of the longest shortest distance between all nodes in the graph and simply printed the output.

Answers to Questions:

Question 2.1:

(First 4 vertices ,(41234,Helge A. Tverberg) , (28730,Kim B. Bruce) , (23776,Andrew M. Marshall) , (5354,Kaleigh Smith))

(First 4 edges ,Edge(0,11176,1) , Edge(1,5871,1) , Edge(1,9102,1) , Edge(1,9710,1))

Question 2.2:

Maximum Erdos number of authors:

3

3 Reflection

"*Cluster Computing with Working Sets"* is a joined publication by Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker and Ion Stoica, all from the University of Berkley in California. The aforementioned paper investigates methods to improve the functionality and abilities of traditional MapReduce frameworks by using RDD's (Spark). The paper conveys this message by demonstrating various applications of Spark such as *"text search"* and *"alternating least squares"* to show the effectiveness of this framework in real time, whilst displaying it's Scala syntax and how it is interpreted by the Java Virtual Machine.

In the introduction, we are informed of deficiencies that were uncovered in the Hadoop and it's iterative MapReduce methodologies, especially the issue of memory efficiency whereby data is loaded iteratively from the disk each time, often causing the same data to be read in multiple times. Furthermore, separate MapReduce queries are run independently for each job, incurring significant time penalties. The purpose of this Spark, the subject of this paper is to combat these existing issues using resilient distributed datasets (RDD) to ultimately improve how MapReduce preforms on big datasets, while not sacrificing the scalability and fault tolerance linked to Hadoop.

The thinking behind Spark is to provide the user with the ability to cache RDDs in memory to reuse the data, as the data is now more quickly accessible as there is no need to constantly reload the data into memory like in Hadoop. Spark maintains the fault tolerance of Hadoop by caching the RDD to memory across multiple machines so that if some of the RDD is lost, then it can be reconstructed from the rest of the existing RDDs. This excellent scalability and fault tolerant framework is implemented in Scala, a high- level programming language built for creating and defining RDDs on the Java Virtual Machine.

The researchers clearly used the scientific method to develop this paper. To begin, the problem at hand and the proposed solution are immediately unveiled in the form of issues with MapReduce using Hadoop and then formulating the Spark and more specifically RDD's as a solution to these issues. The paper further moves on to explain how Spark operates in detail, reinforcing the explanation with three well descripted examples that act as tests for the purpose of this experiment, including Scala implementations. This is followed by even more analysis of how the backend of Spark. Finally, the paper discusses the results of it's tests and compares these results to the results of the same process on Hadoop. It is quite evident here that Spark is far more efficient than Hadoop when there is a greater number of iterations, and the headed sections enter into more detail on the results of these tests. Finally, the paper includes other related work and details the future work and main points to take away from this experiment.

The three tests consisted of a Scala version of logistic regression algorithm that finds a hyperplane that best separates two points, an interactive Spark interpreter which was given the task of loading 39GB of data, and alternating least squares (ALS), a filtering algorithm that predicts movie ratings from users based on their movie-rating history. For the error identification test. From the logistic regression test, we can see that whilst Hadoop is more efficient than Spark initially, as the number of iterations increase, the running time of Hadoop increases exponentially, while at the same time the running time of Spark becomes nearly immediately quicker than Hadoop, as Spark reuses the RDD cached in memory. From this specific test we can see that Spark allows the job to run up to 10 times faster than Hadoop. The ALS test showed us that RDDs take up relatively large amounts of memory to help speed up jobs as the RDD is cached in memory. The interactive Spark interpreter resulted in similar results to the ALS test, where as Hadoop would be initially faster, but for further iterations through the data, Spark returned the subsequent queries instantaneously.

The paper also offers several alternative existing solutions including more restricted but efficient methods like lineage and distributed shared memory, and more fault tolerant methods like Cluster Computing Frameworks, that all have MapReduce properties, but from my comprehension of this paper, I believe that Spark is the most

effective method for MapReducing big Datasets, due to the fact it doesn't sacrifice fault tolerance for efficiency and builds on the ideas of Hadoop, and established MapReduce framework. Furthermore, the fact the authors are future focussed about the development of Spark, with ideas like diversifying its applications and improving its interfaces means that Spark is only going to get bigger and better.

"Furthermore, we believe that the core idea behind RDDs, of a dataset handle that has enough information to (re)construct the dataset from data available in reliable storage, may prove useful in developing other abstractions for programming clusters."