Project 1 Report – Eoghan McGlinchey 18300093

Spark

Before answering the individual questions, I had to read in the csv file and assign it to a variable called data. As well as that, I had to create a temporary table called "dataTable".

1. Finding the project with the most stars involved a simple SQL query. I first selected the name and the stars columns from the table with the constraint being where the number of stars was equal to the maximum number of stars from the table. I was then able to print the result using .collect.foreach(println). The answer I got for this question was [spark,28798].
2. Computing the total number of stars for each language involved another SQL query> I first selected the language and the sum of the stars, grouping them by the language. I was then able to print the results on new lines using .collect.foreach(println). The answer I got for this question was [JavaScript,25344]
3. [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]

   a. Finding the number of project descriptions that contained the word 'data' involved another SQL query. First, I selected the count of the descriptions from the table with the constraint being that the description contained the word 'data'. I applied this constraint using the REGEXP (regional expression) key word. I was then able to print the results on new lines using .collect.foreach(println). The answer I got for this question was [25].
   b. Finding the number of project descriptions that contained the word 'data' and who had their language value set involved another SQL query. Similar to 3a, I selected the count of the descriptions from the table with a nested constraint where the description contained the word 'data' and the language was not null. I was then able

to print the results on new lines using .collect.foreach(println). The answer I got for this question was [24].

4. Finding the most frequently used word in the project descriptions was a bit more challenging than the previous questions. First, I selected the description from the table using an SQL query and assigned it to a variable. I then moved the variable to a temporary CSV file, ensuring that the header was set to false since the previous select statement would not have returned any headers. I then converted the CSV file to a TXT file and assigned it to a variable. I then used MapReduce to count the number of occurrences of each word in the TXT file. Then I swapped the keys and their corresponding values just so that the word would appear before the number of times it appeared. I then sorted these key value pairs in descending order by setting ascending to false. I then took the first key value pair and converted it to an RDD. I then was able to print the result using .collect.foreach(println). The answer I got for this question was (34, Apache).

Graph Processing

Before answering the individual questions, I had to import the relative GraphX packages as well as the relative GraphX classes.

1. I first had to define the schema for the graph, with both columns containing the names of the various authors (strings). I then wrote a function to parse the input into a class called Coauthorship. Since both columns were strings, this was very straightforward. I then loaded the csv file into an RDD variable and removed the headers by filtering it out. I then parsed the RDD of the csv file into an RDD of the Coauthorship class using the function I made to parse the input into a class. Next, I gave I.Ds to the authors using the .distinct.zipWithIndex function and I swapped the values of this variable so that the tuple would have the I.D first, followed by the author's name using the .map(._swap) function. I then defined the default vertex and assigned it the value "nowhere". I then mapped the author's I.D to the authors. Then, I created an RDD that contained the I.Ds of author1 and author2 using the map and distinct functions. I then created the distance variable to represent the distance between the nodes in each edge and let it equal to 1. Next, I created the edges RDD that contained the I.Ds of both authors and the respective distances between them (1). Finally, I was able to create the graph using the Graph function and the authorWithIdSwap vertices, the edges and the default "nowhere" value.
2. Before I began question 2, I needed to import the ShortestPaths packages so that I could compute the shortest paths. I then got the I.D for Paul Erdös using the authorMap.get("Paul Erdös").get function, the authorMap being the function that mapped the I.Ds to the authors. I then used the ShortestPaths.run function to get the shortest paths between the Paul Erdös node and the rest of the nodes in the graph. I then selected just the distances from the tuples by mapping the second value in each tuple and using the map function. I then selected the highest value from these distances using the max function and assigned it to the maxErdösNumber variable. Finally, I was able to print this value using the print() function. The answer I got for this question was 3.

Project 2 Question 3 – Eoghan McGlinchey 18300093

The research paper I have decided to report on is entitled “Spark: Cluster Computing with Working Sets”. The challenge the paper addressed is that most of the systems in which MapReduce and its variants are implemented were built around an acyclic data flow model that was not suitable for other popular applications. To combat this, the authors proposed a framework called Spark which not only supported these applications, but also retained the scalability and fault tolerance of MapReduce. Spark was capable of doing this by using an abstraction known as resilient distributed datasets (RDDs), which are read-only collections of objects that are partitioned across a set of machines that can be rebuilt if a partition is lost. The paper also claimed that Spark was capable of outperforming Hadoop tenfold when it came to iterative machine learning jobs and was also capable of interactively querying datasets up to 39GB with sub-second response time.

There were three main solutions to which the authors compared themselves to – MapReduce, Dryad and Map-Reduce-Merge. While MapReduce pioneered a model in which data-parallel computations were executed on clusters of unreliable machines by systems that automatically provided locality-aware scheduling, fault tolerance and load balancing, Dryad and Map-Reduce-Merge generalised the types of data flows supported. While these systems allowed the underlying system to manage scheduling and to react to faults without user intervention, these systems were limited to applications that could be expressed efficiently as acyclic data flows. As well as that, Hadoop users had reported that these systems were inefficient when it came to handling iterative jobs as well as interactive analytics.

The solution proposed by the authors involved a new cluster computing framework called Spark that could provide similar scalability and fault tolerance properties to MapReduce while supporting applications with working sets. The main idea behind spark was the RDD which was capable of achieving fault tolerance by maintaining enough information about how it was derived from other RDDs to be able to rebuild any partitions that were lost. RDDs also struck a balance between expressivity and scalability and reliability. Spark would be implemented in Scala and was capable of being used interactively from a modified version of the Spark interpreter and allowed the user to define RDDs and other data types and use them in parallel operations on a cluster. Along with the RDDs, the programming model of Spark consisted of parallel operations which could be performed on RDDs such as reduce, foreach and collect. The programming model of Spark also contained shared variables, of which two were restricted – broadcast variables and accumulators.

The authors’ first experiment involved RDDs. The authors found that when a cached dataset was defined and its elements were counted using map and reduce, the dataset would be stored as a chain of objects capturing the lineage of each RDD. They also found that within each RDD, they all implemented the same interface that consisted of the getPartitions, getInterator and getPreferredLocations operators. The authors also examined the parallel operations invoked on datasets and found that when this happened, Spark created a task to process each partition of the dataset and sent these tasks to the worker nodes. The authors tried to alter this by sending each task to one of its preferred locations using delay scheduling. The authors also found that the RDDs were only different in how they implemented the RDD interface which made faults easy to handle. They also realised that to achieve the requirements for shipping tasks to workers, they relied on the fact that Scala closures are Java objects. However, it was found that this aspect was not ideal since they found cases where closure objects referenced variables in the closure’s outer scope that were

not actually used in its body. The authors initially used HDFS to broadcast variables but at the time were developing a more efficient streaming broadcasting system and had found that when a broadcast variable was created, its value was saved to a file system while the serialised form of the broadcast variable was a path to this file. The authors also found that upon implementing accumulators, each accumulator was given a unique ID and saved; upon saving the accumulator, the serialised form contained its ID and it's "zero" value for its type. They also found that the driver applied updates from each partition of each operation just once to prevent double-counting when tasks were re-executed. Finally, in order to make the interpreter work with Spark, the authors made the interpreter output the classes it defined to a shared filesystem and they also changed the generated code in a way such that the singleton object for each line referenced the singleton objects for previous lines directly instead of going through the static.

With regards to the logistic regression, the authors' found that although the first iteration with spark took 47 seconds longer to run than Hadoop, the subsequent iterations were just six seconds since they reused cached data and allowed the iteration to run up to ten times faster. They also investigated crashing a node while the job was running and found that in a 10-iteration case, the performance was slowed by 50 seconds on average. They also found that the recovery time was higher than expected due to the high HDFS block size. Upon implementing the alternating least squares job to measure the benefit of broadcast variables, the time to resend the ratings matrix on each iteration took up the majority of the job's running time. They also found that even when implementing an application-level multicast system, resending the ratings matrix on each iteration was costly. However, they found that caching the ratings matrix using a broadcast variable improved performance by almost three times. When it came to using interactive spark, the spark interpreter was used to load a 39GB dump of Wikipedia in memory across 15 machines and was queried interactively. Although the first query took 35 seconds, subsequent queries took between 0.5 and 1 seconds.

I found the idea behind this research paper to be very captivating. I really appreciated the overall structure of the paper as the sections were presented in a clear and well-defined manner. I also appreciated the clear and concise manner in which the authors identified the solutions to which they compared themselves to and their limitations, it was clear that a new approach was needed to combat the problems associated with MapReduce and acyclic data flow models.