COMP30770 Programming for Big Data
Project 2

Dervla Scully 18329511

Question 1: Spark

Download data
wget http://csserver.ucd.ie/~thomas/github-big-data.csv

Start spark shell
/spark/bin/spark-shell

Upload the dataset from the file, deducing schema from header:

```
scala> val githubData = spark.read.format("com.databricks.spark.csv").
option("header", "true").option("inferSchema", "true"). load("github-big-data.csv")
```

1. Determine which project has the most stars. If multiple projects have the same number of stars, list all of them.

```
scala> githubData.orderBy(desc("stars")).show()
```

This shows all rows in descending order, so we can clearly see the rows with the most stars.

To *just* show the rows with the most stars (and show all if multiple) we can instead do the following:

Create an sql table
```
scala> githubData.registerTempTable("githubTable")
```

Select the rows from the table where the stars = max stars
To computer max stars, select max stars from the table

```
scala> val maxStars = spark.sql("select * from githubTable where stars = (select max(stars) from githubTable)")
```

Print
```
scala> maxStars.collect.foreach(println)
```

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

2. Compute the total number of stars for each language.

Group by language, select the language and the sum of stars for each group

```
scala> val sumStars = spark.sql("select language, sum(stars) from githubTable group by language")
```

Print
```
scala> sumStars.collect.foreach(println)
```

3. (a) Determine the number of project descriptions that contain the word “data”.

Count all row where the description contains the substring “data”
The * represents 0 or more characters
The ‘like’ operator searches for the pattern “[0 or more characters]data[0 or more characters]”

scala> **val countContainsData = spark.sql("select count(*) from githubTable where description like '%data%'")**

Print
scala> **countContainsData.collect.foreach(println)**

Output: 25

(b) Among those, how many have their language value set (not empty/null)?

Count all row where the description contains the substring “data” and the language is not null
The * represents 0 or more characters
The ‘like’ operator searches for the pattern “[0 or more characters]data[0 or more characters]”
To check if the language is not null we can simply do “language not null”
We use ‘and’ to filter on both the language not null and contains “data” conditions

scala> **val countContainsDataNotNull = spark.sql("select count(*) from githubTable where description like '%data%' and language is not null")**

Print
scala> **countContainsDataNotNull.collect.foreach(println)**

Output: 24

4. Determine the most frequently used word in the project descriptions.

//Outside of spark shell

For simplicity, cut the the description column into a separate file and remove the title (first row)
cut -d"," -f2 github-big-data.csv | tail -n +2 > descriptions.txt

// Back in spark shell
Load descriptions.txt into an RDD inputFile

scala> **val inputFile = sc.textFile(“descriptions.txt”)**

Get the word count
Split by whitespace to get the individual words
Map each word to (word, 1)
For reduce, compute the sum of the values for the key (word)

scala> **val wordcount = inputFile.flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(_ + _)**

Swap key, value to value, key to sort by word frequency

scala> **val wordcount_swap = wordcount.map(_.swap)**

Sort keys by ascending=false (descending)
Get the highest (1st element)

scala> **val highestFrequency = wordcount_swap.sortByKey(false,1).take(1)**

Print

scala> **print(highestFrequency.mkString(""))**

Output: (32,Apache)

Question 2: Graph Processing

Imports:

```
import org.apache.spark._
import org.apache.spark.rdd.RDD
import org.apache.spark.util.IntParam
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
import org.apache.spark.graphx.lib.ShortestPaths // for question 2
```

1. Write a Spark GraphX program to read the dataset and build a graph that represents it.

Define the DBLP Schema

scala> **case class DBLP(author1:String, author2:String)**

Function that parses a line from the data file into the DBLP class

```
def parseDBLP(str: String): DBLP = {
        val line = str.split(",")              // split line of csv by the comma
        DBLP(line(0), line(1))
}
```

Load the DBLP co-authorship data into an RDD variable

scala> **var dblpRDD = sc.textFile("/dblp_coauthorship.csv")**

Get the header of the CSV file into a value

scala> **val header = dblpRDD.first()**

Filter out the header (as we cannot parse this to a DBLP)

scala> **dblpRDD = dblpRDD.filter(row => row != header)**

Parse the RDD of csv lines into an RDD of DBLP classes using the parseDBLP function

scala> **val DBLPrdd = dblpRDD.map(parseDBLP).cache()**

Create author RDD, use zipWithIndex to add the index in the DBLPrdd rdd as unique identifier identifier

scala> **val authors = DBLPrdd.flatMap(author => Seq((author.author1), (author.author2))).distinct.zipWithIndex**

[This gives Array((Ivan Kalas,0), (Zbigniew J. Czech,1), (Lijuan Wang,2), ……..)]

Map author id -> author name, and author name -> author id

scala> **val mapIdToAuthor = authors.map { case ((author), id) =>(id ->author) }.collect.toMap**

scala> **val mapAuthorToId = authors.map { case ((author), id) =>(author ->id) }.collect.toMap**

So mapIdToAuthor(Ivan Kalas) gives 0, and mapAuthorToId(0) gives Ivan Kalas

Swap the order in authors to have (id, name) instead of (name, id)

scala> **val authors1 = authors.map{ case (author, id) =>(id, author) }**

Define a default vertex called default

scala> **val default = "none"**

Create a coauthorship RDD. Use mapAuthorToId to get the Id for each author name

scala> **val coauthorship = DBLPrdd.map(author => (mapAuthorToId(author.author1), mapAuthorToId(author.author2))).distinct**

Create edges RDD with author1, author2 as Edge objects

scala> **val edges = coauthorship.map { case (author1, author2) =>Edge(author1, author2, "coauthor") }**

Create property graph with Vertex RDD authors1, EdgeRDD edges and Default vertex none

scala> **val graph = Graph(authors1, edges, none)**

2. **Write a GraphX program to find what is the maximum Erdös number of authors in the dataset, i.e. the maximum value of the minimum distance between an author and Erdös in the Graph.**

Get id of Erdos:

scala> **val erdos = mapAuthorToId("Paul Erdös")**

Output: erdos: Long = 1449

Use ShortestPaths to get (Vertex, Map to minimum distance from Erdos to Vertex) for every vertex:

scala> **val shortestPathsToErdos = ShortestPaths.run(graph, Seq(erdos))**

Get the distance from each Vertex to Erdos by giving erdos to the returned map:

scala> **val allDistances = shortestPathsToErdos.vertices.map{ case (id, map) =>(id, map.get(erdos)) }**

Swap (id, erdos number) yo (erdos number, id)

scala> **val erdosDistances = allDistances.map{ case (id, distance) => (distance, id) }**

Sort descending and take the first

scala> **val highestErdos = erdosDistances.sortByKey(false,1).take(1)**

This gives Array((Some(3), 41234)) so the maximum value of the minimum distance between an author and Erdos in the Graph is 3.

To print the maximum value:

```
scala> println(highestErdos(0)._1.get)
```

3. Reflection

The research paper that I chose was "**Spark: Cluster Computing with Working Sets**", written by Zaharia, Chowdhury, Franklin, Shenker and Stoica at the University of Berkley. I found this to be an extremely interesting and enjoyable read. The paper introduces a framework called Spark which aims to overcome some of the problems associated with Hadoop MapReduce. It outlines the motivation behind Spark, Spark's programming model, presents example tasks and their implementation in Spark, discusses some of the early results, and examines related work. I chose this paper as I found the lectures and the implementation of Spark in the labs particularly interesting. Upon studying this paper I hoped to further my understanding of Spark and examine in more exhaustive detail the motivation behind Spark, as well as the cases in which it has shown to be more efficient than Hadoop MapReduce.

Motivation

Performing efficient computation on large datasets is a crucial task that has substantial implications in today's era of big data. Many methods have been introduced to tackle this task, including the widely-used MapReduce method. MapReduce is a powerful framework for distributed computing which is prominent in many modern cluster-based data applications. It is capable of parallelising and distributing computational tasks while achieving scalability, and fault tolerance, and is useful for a large class of applications. However, it also posses some key limitations. **MapReduce is built around an acyclic data flow model** which is not suitable for applications that reuse a working set of data across multiple parallel operations.

The paper focuses on two use cases in particular in which Hadoop users have reported that MapReduce is suboptimal: **iterative jobs** and **interactive analytics**.

Many common machine learning algorithms apply a function repeatedly to the same dataset to optimise a parameter, for example performing gradient descent in logistic regression to find and improve a hyperplane that best separates two sets of points with every iteration. While we can express each iteration as a MapReduce job, each job must **reload the data from disk**. This incurs a significant performance penalty and so is a key limitation of MapReduce which Spark aims to overcome.

Furthermore, MapReduce has shown a deficiency for data exploration and analytics jobs on large datasets, which are interactive, iterative processes based on a series of ad-hoc exploratory queries. Significant latency again incurs when using Hadoop for such tasks, as each query runs as a separate MapReduce job, and so the data must be **read from the disk for each query**. An ideal solution to this limitation would be to allow the user to load the dataset into memory across multiple machines and query it repeatedly. This is a key idea and motivation behind Spark.

Solution: Spark

The paper introduces Spark, a framework that supports these aforementioned **applications, which reuse a working set of data across multiple parallel operations**, while retaining the **scalability and fault tolerance** of MapReduce. The key insight of this paper is that rather than

pushing all intermediate results back to disk as in MapReduce, Spark allows users to cache the data **in memory** for reuse. This aims to reduce the number of read/write operations to disk, thereby improving latency and performance.

The Spark framework consists of a driver program, which implements the high-level control flow of the application and launches various operations in parallel, and two main abstractions for parallel programming: **resilient distributed datasets** and **parallel operations** on these datasets.

Resilient distributed datasets (RDDs) represent a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark allows programmers to construct RDDs in four ways: by referencing a file in a shared file system such as HDFS, by parallelising an existing collection in the driver program, by transforming an existing RDD, or by changing the persistence of an existing RDD.

RDDs are lazy and ephemeral by default, meaning that partitions of a dataset are materialised on demand when they are used in parallel operations and discarded from memory after use. However, a user can cache an RDD (by changing the persistence) which hints that it should be kept **in memory** after the first time that it is computed so that it can be reused. This caching of RDDs in memory for reuse in multiple parallel operations allows Spark to achieve the **scalability of MapReduce**.

RDDs achieve **fault tolerance through lineage**. When we apply parallel operations to an RDD, for example defining a cached dataset and applying map and reduce operations, each of these datasets will be stored as a chain of objects capturing the lineage of each RDD. Each dataset object contains a pointer to its parent and information about how the parent was transformed. Consequently, in the case of a partition failure, Spark has enough information from other RDDs about how the failure partition was derived to **rebuild** it.

In addition, Spark supports two restricted types of shared variables that can be used in functions running on the cluster: **broadcast variables and accumulators**. Broadcast variables allow for large read-only pieces of data which are used in multiple parallel operations, such as look up tables, to be distributed to worker nodes only once.

Examples & Results

The paper provides a number of example Spark programs. The first is a **text search** program, and is based on counting the lines containing errors in a large log file stored in HDFS. In this example, the log file is transformed into an RDD of lines which contain errors. Each line is then mapped to a 1 and added using reduce, which closely emulates MapReduce. The key difference, however, is that these intermediate datasets can persist in memory across operations so that they can be easily reused for further computation. This greatly speeds up subsequent operations in comparison to MapReduce.

The paper also discusses a **Logistic Regression** example, in which a hyperplane that separates two sets of points is initially assigned a random value, and then on each iteration the program sums a function of the hyperplane over the data in an attempt to move the hyperplane in a direction that improves it. This program benefits greatly from caching the data in memory across iterations. When implemented, this was shown to **run 10 times faster** than the equivalent MapReduce job.

The final example discussed is an Alternating Least Squares program, which unlike the other examples is CPU-intensive rather than data-intensive. The program performs a number of steps repeatedly, all of which use a matrix R. Resending the matrix R to the worker nodes on each iteration was shown to dominate the job's running time. Instead, one can make R a broadcast variable so that it only gets distributed to the worker nodes once. Upon implementation, this was shown to **improve performance by a factor of 2.8**.

Related Work

Spark differs from other distributed shared memory (DSM) models in its fault tolerance. Many DSM systems achieve fault tolerance through checkpointing, which allows applications to roll back to a recent checkpoint rather than restarting. As discussed previously, RDDs use lineage information to reconstruct lost partitions, meaning that only the lost partitions need to be reconstructed. The paper makes comparisons between Spark and other systems which have also attempted to restrict the DSM model to improve performance, such as Munin, Linda and Thor.

The paper also compares Spark to other clustering computing frameworks. Comparisons are made between Spark and Twister, a system which also recognised the need to extend MapReduce to support iterative jobs. Twister allows long-lived map tasks to keep static data in memory between jobs, however, it does not currently implement fault tolerance.

The paper touches briefly on Spark's interpreter integration, and focuses only on how Spark has been integrated into the Scala interpreter. It then proceeds to compare Spark's language integration to that of DryadLINQ. The main difference here once again being that Spark allows RDDs to persist in-memory across parallel operations while DryadLINQ does not. Spark also further enriches the language integration model by supporting shared variables.

Conclusion

My overall interpretation of Spark based on the implementation and results discussed in this paper is that it is a successful model for tackling the problems faced in MapReduce relating to applications that reuse a working set of data across multiple parallel operations. The main advantages of Spark for these applications is speed, due to sparks in-memory caching which reduces disk read/write.

Nonetheless, there are also limitations to Spark which are discussed in the paper. Firstly, Spark's in-memory caching is expensive as it requires a lot of RAM. The Spark model also does not have its own file management system, so it relies on other platforms such as Hadoop. Furthermore, Spark does not yet support grouped reduce operations as in MapReduce. Overall, however, my impression is that Spark can be very effective and efficient for the specific class of applications discussed in the paper.

I found the layout and language in this paper very simple and easy to understand, and the comparisons to MapReduce were very clear and comprehensive. However, the paper only compares Spark to Hadoop MapReduce, with only very brief comparisons to other systems near the end of the paper. The results may be more interesting and convincing if compared to other on-going distributed systems. Furthermore, the evaluation and results provided were very limited. For each experiment described, only one trial is mentioned, and there is little mention of how the performance varies as the size of the dataset or the hardware configuration changes. It also doesn't discuss in much detail how Spark performs under memory limitations. It mentions

briefly how if there is not enough memory for caching Spark will recompute the data when it is needed, however there is no mention of specific performance results in such a scenario.

With the exception of these drawbacks, I found the paper to be extremely enjoyable and thought-provoking, and it greatly expanded my understanding of Spark, as well as the limitations of MapReduce.