ECS640U – Big Data Processing

Comparing big data paradigms: MapReduce vs in-memory dataflow programming

Students:
Zain Abbas / 120161603, Dolica Akello-Egwel / 120686298, Mohsin Mahmood / 130118925, Valentin Naumov / 130214319

1. Introduction

1.1. Overview

MapReduce and Spark are both frameworks that allow for parallel processing with large volumes of data. In MapReduce this is accomplished by partitioning input data and distributing it among nodes within a network that carry computations in a map phase and a reducer phase. The map function takes some input data and transforms it to key/value pairs while the reducer receives all the key/value pairs of a given key and merges them to produce a single result.

Spark is the much newer of the two and is viewed by some as an answer to the criticisms many have against MapReduce. Spark consists of a driver program that executes parallel operations on RDDs across a cluster that can be manipulated through transformations. These RDDs serve only as representations of data and are not loaded on the system until a user performs an action that produces some kind of result.

By performing a set of computations in both MapReduce and Spark and varying the size of the input data it should be possible to observe how these two frameworks fair in terms of versatility and scalability.

1.2. Amdahl's Law

Amdahl's Law provides a theoretical speed up in latency of the executed task at a fixed workload that can be expected of a system whose resources are improved, with the consequence being that the amount of speed-up that can be achieved through parallelism is limited by the non-parallel portion of the program. It uses the formula of 1/((1-P)+(P/N) to calculate the speed-up of the task. That sates that in parallelization the proportion of a system (P) or a program which can be made parallel and the proportion that remains serial (1-P), then the maximum speed-up that can be achieved using number of processes (N) is 1/((1-P)+(P/N). Showing that the theoretically the execution of the whole task increases with the improvement of the resources of the system and that regardless of the improvement, the speed-up is always limited by the tasks that cannot be parallelized.

Amdahl's Law only holds when the problem size is fixed, when more computing resources become available, they tend to get used on larger datasets, with the time spent in the parallelizable part usually grows at a faster rate than the inherently serial work.
One of the difference between Amdahl's Law ideal case and an actual case is that not every action done in a program will have the same amount of parallelization with Amdahl's law glossing over it treating each process with equal amount of parallelization. Another limitation

is that it ignores communication cost, with communication intensive applications, Amdahl's law does not capture the additional communication slowdown due to network congestion. As a result, Amdahl's law usually overestimates speed-up that is achievable.

In terms of MapReduce tasks, not every aspect of a MapReduce job will be parallelizable. The Mapper and Reducer jobs will be however functions such as joins and writing to files won't be and this reduces the percentage of parallelizable functions, causing less speed-up.

2. Joins and Counting

Dataset used: MovieLens
Objective: Determine the most controversial movies

This computation is carried out by joining the `ratings.dat` file and `movies.dat` file by their `MovieIDs`. A calculation is then performed to determine which movies received an equal number of negative and positive ratings.

2.1. MapReduce Implementation

2.1.1. Joins

In MapReduce the datasets are joined using an inner replication join. The `movies.dat` file is 0.5MB large and can be copied to each node without taxing the network.

This join is achieved by loading the `movies.dat` dataset in the `RatingsJoin` class. The mapper then uses this file to create a hashtable from the `MovieIDs` contains and their corresponding `Titles`, as shown below:

```
try {
      br.readLine();
      while ((line = br.readLine()) != null) {
            String[] fields = line.split("::");
            movieInfo.put(fields[0],fields[1]);
      }
      br.close();
}
```

From this the `ratings.dat` file can then be split so that the rating is matched with the movie title contained in the `movieInfo` hashtable. This is shown below:

```
String line = value.toString();
String[] parts = line.split("::");
String name = movieInfo.get(parts[1]);
movieRating.set(Double.parseDouble(parts[2]));
if (name!= null){
      movieTitle.set(name);
      context.write(movieTitle,movieRating);
}
```

MapReduce's output statistics revealed that 10000054 records were inputted to the mapper, but only 9973605 were emitted after the join. This basically meant that some ~26000 records

simply went missing unless the code was modified to accommodate movies without a title. After comparing the output of MapReduce and Spark it was discovered that Toy Story was among these "unmatchable" movies. We suspect that this is due to the way the hash table deals with different items that share the same hashcode.

2.1.2. Counting

Counting is carried out by the Reducer. An `Iterable<DoubleWritable>` object is traversed to examine all of the ratings of a given movie. From there the "controversy score" can be determined, which is simply 1 - | (number of positive ratings)/(number of negative ratings) | or 1 - | (number of negative ratings)/(number of positive ratings) | depending on whether the number of positive or negative ratings were greater. Movies that received 0 negative or positive ratings or received less than 10 ratings in total were discarded. When this is complete the statement `context.write(key,controversy)` is used to create the final output of the job, where `controversy` is a `DoubleWritable` object containing the controversy score arrived at through the our formula and `key` is a `Text` object that stores the movie title.

2.2. Spark Implementation

2.2.1. Counting

Counting is done first in Spark because it makes it slightly easier to deal with the RDD. First the system is informed where to find the datasets through calling the `sc.textFile()` method. Once this is carried out the datasets are then mapped to key value pairs as shown below:

```
val IDTitle = moviesFile.map(x => x.split("::")).map(x => (x(0),x(1)))
val IDRating = ratingsFile.map(x => x.split("::")).map(x =>
(x(1),x(2).toDouble))
```

Creating a map from the ratings data and also converting it to a double can be achieved in a single line of code. In contrast, MapReduce/Java requires splitting the file, converting the rating to a double, setting the data in a `TextWritable` and `DoubleWritable` object, and then finally writing them both to the context. The equivalent operation in Spark/Scala requires considerably fewer steps. While the functional programming-based syntax of Spark/Scala is not an easy transition from an OOP language, it allows for the easy manipulation of data in a key/value-format through `map` functions.

Collecting the ratings of the same movie is achieved though calling the `groupByKey()` method. This creates a data structure `ratingGroup` in the form of `[String,Iterable[Double]]`. The counting phase is complete when the object is mapped to the same key, and the return value of the `findControversy` method that functions in a similar way to the Java code.

```
val controversyRatings = ratingGroup.map{case(k,v) => (k,findControversy(v))}
```

The result is a map containing a movieid as a key and the controversy score as a value. Spark does not natively support calling ++ to increment a value but this may be because it provides a `count()` method that allows for something similar without the need of loops.

2.2.2. Joining

Joining in Scala is incredibly simple granted two maps contain a matching set of keys. Once this is in place all that needs to be done is calling the `join()` method. Calling the `count()` method reveals that our join object contains the same number of elements as the original ratings file, meaning that no records have "gone lost."

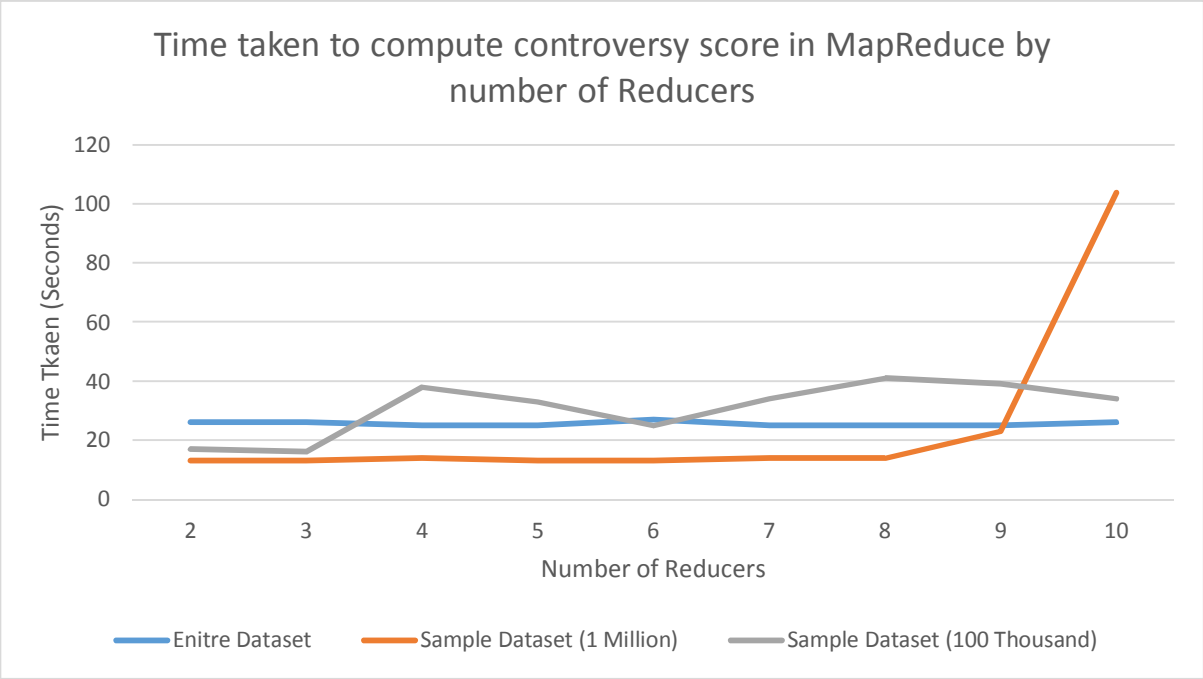
```
val join = IDTitle.join(controversyRatings)
```

Although it seems more intuitive to call `join()` on the much larger `controversyRatings` object and pass the smaller movie file as an argument this actually results in the rating coming first and the title appearing second. Nevertheless, the operation is carried out quite quickly and easily.

Finally the output is saved to the HDFS.

2.3. Results

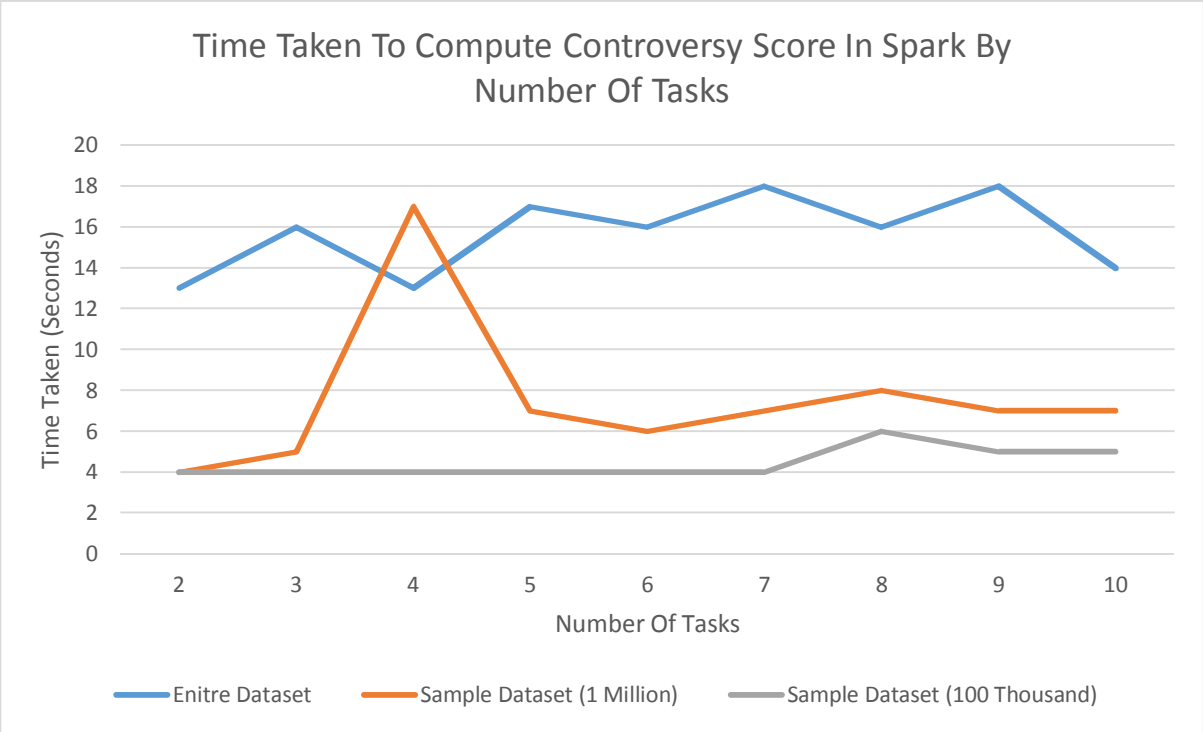
2.2.1. MapReduce



The smallest dataset generally took 40 seconds or less to complete the computation. We can see that 2 or 3 reducers seems to be the ideal number, past which the computation begins to take longer. We can infer that due to the small size of the dataset it seems likely that the non-parallelizable component of the computation very quickly starts to outweigh the parallelizable component and is then multiplied with each additional Reducer. With small datasets increasing the number of reducers quickly ceases to yield a speedup. With the million-record dataset we can see that increasing the reducers up to 8 still produces desirable results.

It is unusual that the smallest dataset had greater times on average than the other datasets which may be due to a lot of activity taking place in the Hadoop culture or might suggest that MapReduce ought not to be used with datasets that are "too small." It would seem that while MapReduce can manage datasets of increasing size, it is still not very scalable.

2.2.2. Spark



While Spark does not rely on Reducers it was possible to alter the number of Tasks used in both the `groupByKey()` and `join()` methods. Although many people advise against using `groupByKey()` to collect values with the same key our Spark program still outperformed MapReduce in terms of speed. It is also quite possible that the `findControversy()` method was not as efficient as it could have been as it did not make use of Spark's `count()` feature and still returned values that received 0 positive or negative ratings and <10 ratings, while the MapReduce/Java code dismisses them. This gave Spark a marginally larger workload because the output contains a greater number of key/value pairs as well as the fact that the movieid information is retained and written to the HDFS but it still carried out every job we tried in under 20 seconds. This is half the time that some MapReduce tasks took.

From these results we can observe that in the case of the smallest dataset increasing tasks did not make the computation any faster, and that from 8 tasks onwards the computation actually became slower. At this point it is reasonable to believe that the parallelizable component of the computation became smaller than the non-parallelizable component.

We observed a noticeable leap when using the largest dataset, but overall Spark's speed was still vastly superior to that of MapReduce. Spark seemed to be the better-performing of the two in terms of its scalability.

3. Iterative Graph-Processing

Dataset used: Web-Google [https://github.com/yuhc/web-dataset/blob/master/web-Google/web-Google.txt]
Objective: Determine how many nodes can be reached from 1234

This computation consists of performing a Breadth First Search on a graph file in the form of node pairs where the first node is the source, and the second is its destination.

3.1 MapReduce Implementation

Graph analysis is performed in Java by using a custom `GoogleNode` data structure to store information on the nodes in the graph. This requires a separate Map-only job that converts the text file to graph nodes and sets their distance from the source node to infinity.

Once this is complete then `GoogleIterateMapper` and `GoogleIterateReducer` carry out the BFS algorithm. The `runJob()` method in the `GoogleIterate` class returns a `Boolean` to indicate if the number of reachable nodes in the map was equal to the number of reachable nodes in the Reducer. This then allows a while-loop to break when the algorithm is complete. This is not a native feature of MapReduce. The user has to create their own custom counters instruct the program to inspect them when a job completes.

Because the process is being carried out iteratively it is also necessary to inform the program to change the directory each time the job is run which is carried out my incrementing an int which is used to determine the input/output directories.

3.2. Spark Implementation

The GraphX library in Spark has an `edgeListFile()` method that quickly converts a file in the format of node pairs to a graph object. The number of nodes the graph file is distributed across can be specified in the argument. Then an initial graph is created that map all of the nodes apart from the source node to a distance value of infinity, thus the preparation phase is completed in a single line of code as opposed to MapReduce/Java's 2 classes.

```
val graph = GraphLoader.edgeListFile(sc,"hdfs://moonshot-ha-nameservice/user/dae30/input/googlegraph/hundredth.txt", numEdgePartitions = tasks).partitionBy(PartitionStrategy.RandomVertexCut)
val root: VertexId = 1234
val initialGraph = graph.mapVertices((id, _) => if (id == root) 0.0 else Double.PositiveInfinity)
```

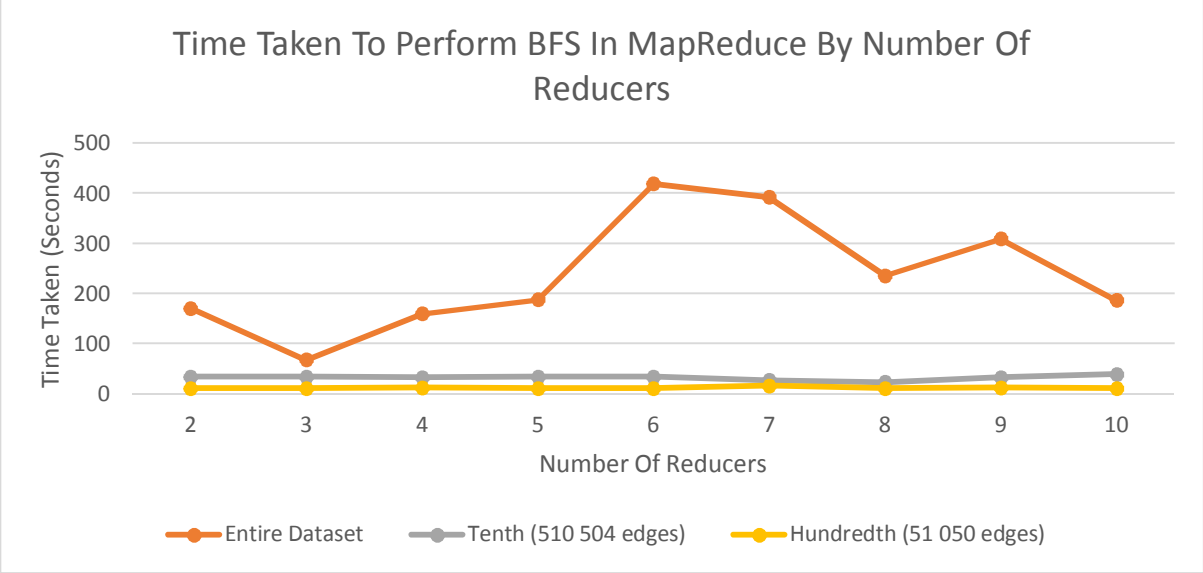
Once this is complete then the BFS algorithm determines the distance of each of the nodes. From this the `subgraph()` method is called to remove all cases where the distance is infinity. Afterwards only the reachable vertices are left, and using the `numVertices` method then completes the computation.

In Spark/Scala this entire procedure requires a single file and 6 lines of code to perform in contrast to MapReduce/Java's 6 files. Even without relying on the GraphX library one could create their own custom Graph class and use 1 or 2 files to carry out this computation. It is also not necessary in Spark/Scala to create files storing partial results during every iteration

of the algorithm because it can be completed in a single execution of the Spark/Scala file. This is a tremendous advantage when dealing with graphs that are several hundred GB in size or greater.

3.3. Results

3.3.1. MapReduce



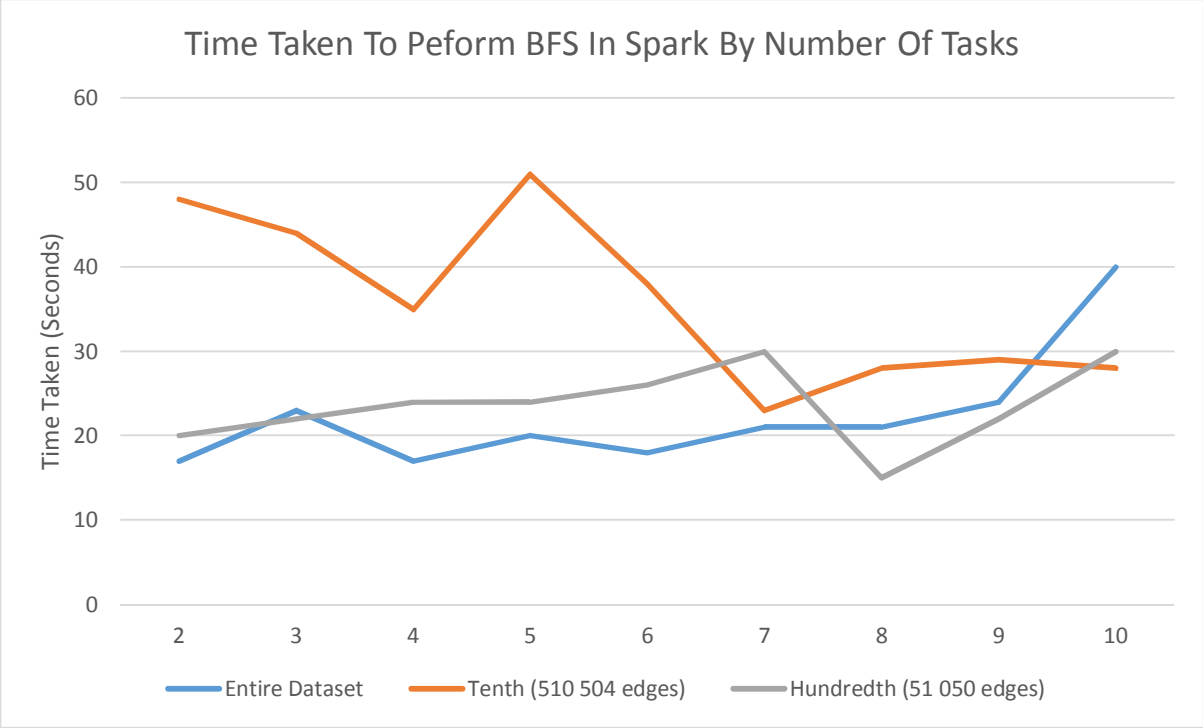
Initially when using the smallest dataset (51050 edges) the time taken to complete the algorithm ranges between 10-15 seconds when increasing the number of reducers from 2 – 10. Contrary to our expectation that the time taken would increase as the number of reducers increase, we observed little to no change which does not conform to Amdahl's Law.

Increasing the dataset to 510504 edges resulted in an increase in time taken to complete the algorithm, with results ranging from 22-38 seconds. Again, this does not seem to be consistent with Amdahl's Law as although the time taken did increase with the number of reducers, it merely increased by a few seconds.

Finally, when using the whole dataset a dramatic increase was witnessed in time taken, with the longest job taking ~7 minutes and the shortest ~1 minute. This led us to conclude that in terms of scalability MapReduce is underachieving, due to a considerable time increase occurring as the data volume increased. It would seem that at least in terms of speed MapReduce is not particularly scalable when handing graph-processing algorithms.

It is also important to make note that only the performance of the BFS algorithm was measured, and the preparation phase was put aside. A number of times the algorithm also appeared to halt prematurely, though this is probably due to our code rather than the result of a defect in MapReduce. A number of times the algorithm completed after two iterations rather than the expected three.

3.3.2. Spark



When using the smallest dataset (51050 edges) the time taken to complete jobs lay in the range of 20-30 seconds, which is a notable increase from the MapReduce job. As the graph suggests there is a slight increase in time taken as the number of reducers increase, this rise is in the region of 10 seconds and follows Amdahl's law.

Thereafter, when using the dataset comprising of 510504 edges, the time taken for completion of jobs increased by a small amount from when using the smallest dataset. This is also similar to the results obtained by the MapReduce job, suggesting that when dealing with small amounts of data either framework could be used in terms of speed. Lastly, when dealing with the entire dataset we observed a huge difference when compared to MapReduce, with time taken in the region between 17-40 seconds. Again, as the number of reducers increase the time taken also increases with the biggest increase of 23 seconds.

Interestingly, we observed some of the fastest speeds with the largest dataset, while it was the performance with the smallest dataset that appeared to struggle.

From the graph we can see that once the number of reducers has reached the area between 7 and 9, there is then a further increase in time taken suggesting the non-parallelizable part outweighs the parallelizable component.

Our time measurements take into account the preparation phase whereby the data is transformed into a graph object. This was not included during MapReduce but we still find that it shows superior performance in terms of speed.

4. Conclusion

In conclusion, Spark appears to me the much faster and more efficient way of handling Big Data jobs in comparison to MapReduce. Overall results for processing our chosen algorithms over datasets of different sizes shows that Spark in all cases was the faster of the two tested methods.

Spark processes data in-memory, whereas MapReduce after processing data, pushes it back to the disk. Therefore, in pretty much all cases, Spark will perform better in comparison to MapReduce.

However, in comparison to MapReduce where files are destroyed in memory when jobs have finished, Spark needs more memory allocation and this is because processes are loaded into system memory and for some time are cached. In situations where data is too big for the system memory or Spark is running alongside resource intensive services like YARN, drops in performance can occur heavily. However, for our project and the cases above this isn't a visible issue.

MapReduce/Java was also a great deal more verbose than Spark/Scala. Tasks that could be achieved in a single line of code required several more lines of code in MapReduce/Java and sometimes even additional classes. It is also important to note that in many cases a less efficient procedure was used in Spark but its results show that it is a much faster approach to our problems in comparison to MapReduce.

The most powerful advantages can be observed in the way Spark/Scala handles graph processing where otherwise complicated MapReduce/Java procedures can be achieved in a way that is far more simple, far more easier to understand, and overall performs a lot faster.

While both Scala and MapReduce showed similar increases in speed when having to deal with larger data, Scala's overall speed was still vastly smaller than MapReduce in the majority of cases.

In addition, in the case of joins, MapReduce's performance simply appears to be defective because certain records failed to match whereas Spark does not suffer from this issue and correctly matches every entry.

References

Cuadrado, F. (2015). 'Week 5: Parallel computing performance',*ECS640U Big Data Processing.* Avaliable at: http://plus.qmul.ac.uk/course/view.php?id=4916 (Accessed: 14 December 2015).

Wolfe, M. (2015). *Compilers and More: Is Amdahl's Law Still Relevant?*. [online] HPCwire. Available at: http://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/ [Accessed 16 Dec. 2015].
Lewis, T., El-Rewini, H. and Kim, I. (1992). *Introduction to parallel computing*. Englewood Cliffs, N.J.: Prentice Hall.

Koivisto, D. (2005). *What Amdahl's Law can tell us about multicores and multiprocessing*. [online] Embedded. Available at: http://www.embedded.com/design/mcus-processors-and-socs/4006486/What-Amdahl-s-Law-can-tell-us-about-multicores-and-multiprocessing [Accessed 16 Dec. 2015].

Apache-spark-user-list.1001560.n3.nabble.com, (2015). Apache Spark User List - BFS implemented. [online] Available at: http://apache-spark-user-list.1001560.n3.nabble.com/BFS-implemented-td4470.html [Accessed 16 Dec. 2015].