1 Introduction

In any modern business application or scientific effort, the ability to manage large volumes of data is paramount [2, 4]. For this purpose, businesses and scientific communities worldwide developed systems to store and analyse volumes of data that get larger and larger every year. Consequently, data is growing faster than our ability to store or index it [3]. A few examples include 3 Billion Telephone Calls in USA each day, 30 Billion emails daily, 1 Billion SMS, IMs and much more [3]. One particularly important application of Big Data technologies is through scalable text analytics. For instance, every time someone submits a query through Google's search engine, Google's systems have to parse through vast amounts of web pages and count occurrences of different words within the page, or execute other text analytics processes. As such, for companies such as Google, Facebook and Microsoft, it is essential to have a system capable of parsing and analysing large documents efficiently. Nevertheless, accessing data stored on disk (e.g. contents of web pages, or data stored by a database) is very slow compared to the computational power of modern systems, which increases exponentially [5]. This created an impetus on using parallel processing to facilitate disk access procedures, such as reading or writing.

One such system is the MapReduce framework, implemented within Hadoop, a reliable and scalable platform for storage and analysis [3]. MapReduce is a programming framework for parallel and distributed computing in Java [1]. MapReduce is comprised of two components: the mapper and the reducer. The mapper's purpose is to convert the data into a set of key/value pairs, otherwise known as tuples. During the second stage of the process, the reducer reads the key/value pairs produced by the mapper and aims to reduce the number of tuples through the following processes: the shuffle stage and the reduce stage. The main advantage of the MapReduce paradigm is the abstraction that it provides over the Read/Write problem described before. It achieves this by transforming the said problem into a computation over sets of keys and values [3].

One of the most popular platforms that implement the MapReduce framework is Hadoop. Hadoop is comprised of several software utilities that streamline the process of utilising many interconnected computers to handle large volumes of data. One of the main additions that Hadoop introduces to the MapReduce paradigm is the Hadoop Distributed File System (HDFS). Through this approach, Hadoop partitions files into blocks which are then distributed between different nodes in a cluster. Subsequently, Hadoop distributes packaged code (that is designed using the MapReduce paradigm) to the mentioned clusters which are then executed in parallel, thus improving the overall efficiency of the system.

Although Hadoop provides a very efficient approach to the process of parallel computations, the task of configuring the system and developing the required code can often be very tedious and prone to errors. For this purpose, several higher-level systems have been designed, that allow the programmer to build the desired query without writing lower-level Java code through the MapReduce paradigm. One such system is Apache Hive, a data warehouse software mechanism built over Hadoop that provides data query and analysis [6]. One of the primary advantages of using Hive over simple MapReduce code is its SQL-like interface, also known as HiveQL, which is often easier to use as well as less prone to human error. An important thing to note is the usage of Hive on top of Spark, an open source, in-memory execution environment for running analytics applications. One of the primary reasons for using Hive with Spark rather than Hadoop is the execution speed of the submitted jobs. Since Spark is an all-memory database, jobs processed by Spark tend to be executed much faster than those processed through Hadoop.

As described in the following section, for the purpose of this project we have attempted to provide an implementation to two common data query procedures, Top-k Query and Reduce Side Join, which have been implemented in both Java, through the MapReduce with Hadoop system, and Hive with Spark. A more detailed description of the two procedures can be observed in section 2 and a thorough comparison of the two approaches (e.g. performance, implementation difficulty, etc.) can be found in section 3.

2 Implementation

2.1 MapReduce

The first part of the project consisted of designing two queries using a MapReduce programming padagrim. Both programs completing the queries described in the project specification were built using Java and exported into a JAR file using an extension to Eclipse - Eclipse-Hadoop-Plugin. Once exported, the programs can be run on Hadoop by submitting a job using the following command in the terminal:

```
$ hadoop jar u1633084_u1600967.jar task1.TopKStores 10 2451457 2451813 tpcds40G/store_sales.dat
    topk_u1633084_u1600967
$ hadoop jar u1633084_u1600967.jar task2.ReduceSideJoin 10 2451457 2451813 tpcds40G/store_sales.dat
    tpcds40G/store.dat join_u1633084_u1600967
```

Each MapReduce program consists of three main components:

- **Wrapper class** - a class that contains all the necessary configuration for running the MapReduce job. It parses and handles user input (arguments to the program). The input file(s), output path and additional constraints are saved to job's configuration. The configuration is stored in a *Configuration* class instance that can be shared across classes of the program using publicly accessible *Context* object.
- **Mapper classes** - a set of mappers that implement distinct *map()* functions writing key-value pairs to the context. Mappers can be bound per file in the main class using the *MultipleInputs* object.
- **Reducer class** - a class that implements a *reduce()* method parsing the output of all mappers grouped by the key of the tuples. Produces a set of key-value pairs used as the final output of the program.

2.1.1 Top K Stores

The first task of the MapReduce part aimed to design a MapReduce program capable of producing a list of up to k stores with the highest total net profit. The data supplied in *.dat* files is '|' separated and contains missing attributes for some records. Hence, a new parsing method *tokenize()* was implemented (see Fig. 2) to account for incomplete tuples.

The wrapper class of the program is *TopKStores*, which saves in the program configuration the constraints specified by the user: (1) number of stores to display, (2) earliest date and (3) latest date of a sale record to be considered. The class maps a single input file (path from user input) to a *TopKMapper* mapper class and assigns *TopKReducer* as a reducer class.

Each mapper parses subsequent lines of the input file using the *tokenize()* method and determines whether all necessary attributes are present in a tokenized tuple (store key, net profit and date). If present and the data is within the specified time period (retrieved from the context), then a pair with key being the store ID and value being the net profit is written into the context, otherwise ignore it.

The reducer contains one private attribute - a *TreeMap* of total net profit as the key and store ID as the value, which allows for storing these entries in order. Descending order can be imposed by constructing the *TreeMap* object with a *Collections.reverseOrder()* attribute. The *(*reduce()) method sums a set of all mapped values for each key to obtain the total net profit. The result for a given key is then added to the *TreeMap* object, then, if structure's size exceeds the threshold k the entry with the smallest profit (key) is found using *lastKey()* method and removed. Finally, the default *cleanup()* function is overwritten to write into the context each entry stored in the map, which produces the final result.

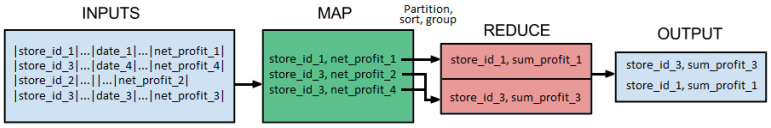

Figure 1: Simplified work flow of the MapReduce program for selecting top k (here $k = 1$) stores with highest total net profit.

```
public ArrayList<String> tokenize(String line, char separator) {
    ArrayList<String> result = new ArrayList<>();
        char[] s = line.toCharArray();
        String val = "";
        for(int i=0; i<s.length; i++) {
                if(s[i] == separator) {
                    result.add(val);
                    val = "";
                    continue;
                }
                val += s[i];
    }
    return result;
}
```

Figure 2: Method in Java allowing to tokenize input String on a given separator.

2.1.2 Reduce-Side Join

The second task in this section was to not only generate a list of top k stores with the highest net profit, but also map each store with their number of employees. The task is a direct extension to task 1 (see subsection 2.1.1).

The wrapper class is *ReduceSideJoin*, it extends the wrapper from task 1 by requiring additional input path from the program arguments. It also maps the two input files to separate mapper classes using the *MultipleInputs* object. Outputs of both mappers are then parsed by a single reducer to produce the final result.

The first mapper *TopKMapper* is a mirror copy of the mapper from task 1. The only difference being that this class writes to the context store IDs as keys and a "*topk*" label concatenated with profit, which allows distinguishing the origin of input in the reducer. The second mapper *NumEmplMapper* parses the second input file containing details on each store. Each line is parsed using *tokenize()* method, and if both store ID and number of employees is present in a given record, then it is written into the context with a value label "*numEmpl*".

Once partitioned, sorted and grouped, the *JoinReducer* reducer class parses outputs of both mappers. The class has two attributes: (1) a *TreeMap* containing sorted pairs of total net profit and store ID (same as *TopKReducer*), (2) a *HashMap* of store IDs and corresponding number of employees. The *reduce()* function iterates values aggregated for a given key and handles the entry depending on its origin (stored in value's label). If the entry has a "*topk*" label then the remainder of the value is the profit and it is added to the store's total profit, otherwise, the second element of the value represents the employee count and it is added to the *HashMap* object. Once iterated, the summary net profit value is put in the *TreeMap* and if the tree size exceeds k then the entry with the smallest profit is removed from the structure. Once all reducers exit, the overwritten *cleanup()* method is called, which creates a temporary *TreeMap* of reverse key-pair order (i.e. store ID is the key and profit is the value). Therefore, the tree can be traversed in ascending order of store ID and together with the employees number retrieved from the *HashMap* can be written into the context.

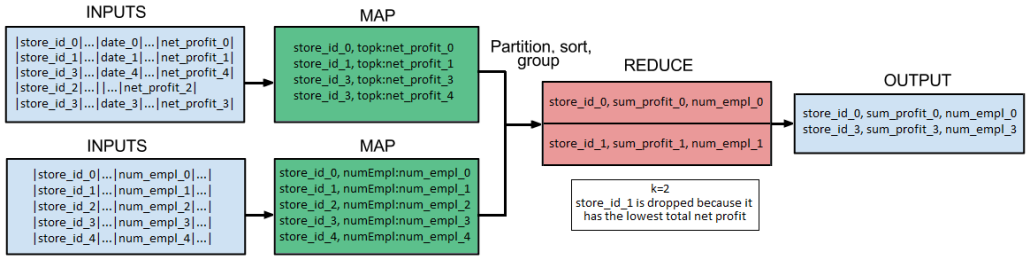

Figure 3: Simplified workflow of the MapReduce program for selecting top k (here $k = 2$) stores with highest total net profit and joining the result with another query in the reducer to append the number of employees.

2.2 Hive and Spark

The second part of the project consisted of reiterating the previous two exercises using Hive and Spark rather than Hadoop and MapReduce. As conferred in the introduction, the advantage of using Hive rather than MapReduce is based on the SQL-like interface that Hive provides, which is often much easier to use and less prone to errors. Moreover, using Hive on top of Spark also offers better performance due to the fact that Spark is an in-memory execution environment that runs different analytics jobs faster than Hadoop.

After installing Hive, we had to configure our system to recognize it by adding the following two lines to the *.bashrc* file:

```
export HIVE_HOME=/path/to/hive
export PATH=$PATH:$HIVE_HOME/bin
```

The configuration required to run Hive on top of Spark has been provided by the department, and as such, the second step was to establish a connection with the server. For this purpose we have used the beeline service:

```
$ bin/beeline
```

After starting the beeline service we had to connect to the actual server using the following command:

```
beeline> !connect jdbc:hive2://137.205.118.65:10000/default;auth=noSasl hiveuser password;
```

After completing the steps outlined above, the connection has been successfully established and we were able to implement our solution to the stated tasks. Occasionally, we would check the job progress using the following link: `http://137.205.118.65:8088`

2.2.1 Top K Stores

As described in section 2.1.1, the purpose of the first task was to design a program, this time using Hive's SQL-like interface, capable of retrieving a list of K stores with the highest net profit. Below is the query that we implemented to achieve this:

```
SELECT ss_store_sk, SUM(ss_net_profit) AS profit
FROM store_sales_1g
WHERE ss_store_sk IS NOT NULL and ss_net_profit IS NOT NULL and ss_sold_date_sk IS NOT NULL and
     ss_sold_date_sk >= 2451457 and ss_sold_date_sk <= 2451813
GROUP BY ss_store_sk
ORDER BY profit DESC
LIMIT 3;
```

The **SELECT** clause reduces the number of projected attributes to only those that we are interested in: the store's key and corresponding total net profit. The net profit has been computed using the *SUM* aggregation function in conjunction with the **GROUP BY** clause, that creates groups of stores with the same key, upon which the aggregation function is applied. The **FROM** clause specifies what table to be used for retrieval. During the implementation stage we have used the 1 Gigabyte version of the table, however, for the final iteration, we have tested our solution on the 40 Gigabytes table as well. The **WHERE** clause is used to reduce the number of tuples considered for selection and aggregation. The first three conditions ensure that some of the critical fields are not null (i.e. the store's key value, the associated profit and the date of the sale). The second two conditions verify that the date of the sale is indeed between the two provided boundaries. The **ORDER BY** clause is used to order the resulting rows in descending order of the total net profit value. Finally, we use the **LIMIT** clause to limit the number of returned records to be equal to the specified K value, which in the above example is equal to 3.

2.2.2 Reduce Side Join

The second task of this assignment was to produce a list of top-k stores based on their net profit, and execute a join operation with another table that stores the number of employees associated with each store. As described in section 2.1.2, this is a direct extension of the previous task. The HiveQL code that we developed for this exercise is as follows:

```
CREATE VIEW topk AS
SELECT ss_store_sk, SUM(ss_net_profit) AS profit
FROM store_sales_1g
WHERE ss_store_sk IS NOT NULL and ss_net_profit IS NOT NULL and ss_sold_date_sk IS NOT NULL and
     ss_sold_date_sk >= 2451457 and ss_sold_date_sk <= 2451813
GROUP BY ss_store_sk
ORDER BY profit DESC
LIMIT 3;
```

```
SELECT topk.ss_store_sk, topk.profit, store.s_number_employees
FROM topk JOIN store_1g AS store ON topk.ss_store_sk = store.s_store_sk
ORDER BY topk.ss_store_sk;
```

```
DROP VIEW topk;
```

The first code snippet presents an updated version of the code used in the first task, which is responsible for selecting the top K stores and their total net profit and order them in descending order of the profit value. The only difference in the context of this task is given by the **CREATE VIEW** clause which stores the returned tuples in a temporary table view, then used as input for the second query thus producing the final output.

The second query is used to implement the join operation between the contents of the produced view and the data stored in the *STORE_1G* table. The view contains a list of ordered stores and their total net profit, whereas the *STORE_1G* contains the number of employees associated with each store, hence the required join operation between the two tables. As before, the **SELECT** clause reduces the number of projected attributes to only those that we are interested in: the store's key, the total net profit, and the number of employees for each store. The **FROM** clause, in conjunction with the **JOIN** clause, specify the tables from which the data will be retrieved. The **ON** clause provides the join condition, which in this case is an equality relation between the store key of the rows selected from the view and those selected from the *STORE_1G* table. The **ORDER BY** clause, reorders the selected tuples, in ascending order, based on the key value of each row.

Finally, the **DROP** clause presented in the final snippet is used to delete the topk view created previously. As a result, when running the query sequence in a future iteration no exceptions will be raised due to the fact that there already is a view with the specified name.

3 Testing

In this section, the obtained result and corresponding time efficiency will be discussed. We will present the outcomes for both MapReduce and Hive approaches for the queries specified correspondingly in sections 2.1 and 2.2 of this report. For the same tasks and equivalent parameters, both Hive and MapReduce yield the same results (see Fig. 4) which further increases the probability of their correctness. The programs were tested by running the same sequence of queries and evaluated in terms of their time efficiency and correctness. Despite there being additional memory usage information supplied from the execution of a Hadoop job, the Spark provides only elapsed time of the HiveQL query, hence, making it the sole performance comparison measure.

Query	Top K stores				Reduce Side Join					
Data size	1G		40G		1G			40G		
Result	4	-7.0288976E7	110	-2.9603104E8	1	-7.1105624E7	245.0	22	-2.9703488E8	228.0
	8	-7.065968E7	37	-2.96715936E8	2	-7.17464E7	236.0	37	-2.96716352E8	244.0
	7	-7.0945344E7	22	-2.97036064E8	4	-7.028908E7	218.0	46	-2.97865216E8	238.0
	1	-7.1105096E7	103	-2.97409088E8	7	-7.0944128E7	297.0	73	-2.97570912E8	234.0
	2	-7.1746784E7	73	-2.97571488E8	8	-7.0659792E7	278.0	79	-2.98477888E8	204.0
	10	-7.1865552E7	112	-2.97744384E8	10	-7.1865616E7	294.0	80	-2.98369984E8	208.0
			46	-2.97867712E8				82	-2.98641952E8	290.0
			80	-2.9836464E8				103	-2.97408864E8	211.0
			79	-2.98476544E8				110	-2.96024672E8	252.0
			82	-2.9864144E8				112	-2.97742528E8	283.0

Figure 4: Results obtained for the two described tasks obtained by both MapReduce program and Hive queries. The full commands used to generate above outcomes are presented in 2.1 and 2.2.

To ensure correctness, before running the queries on the 10G and 40G data set, a smaller subset of the store data was created, which allowed for manual result calculation in Excel and comparison with the outcomes of developed programs. Different user inputs were also tested to verify their valid parsing and obtaining expected results on the smaller data. Due to Hive outputs being more reliable, they were used as the benchmark for the 10G and 40G data sets to evaluate the correctness of the MapReduce programs. In terms of time complexity, the Spark and Hive method proved to be superior. Not only were its execution times faster than those achieved by the MapReduce programs, but also more predictable (see Fig. 5). This predictability arises from smaller variance in the execution time for the same query (i.e. the same task, data set and arguments). Combined with the advantages discussed in sections 2.2 and 4, the latter Spark approach is deemed favourable for this project.

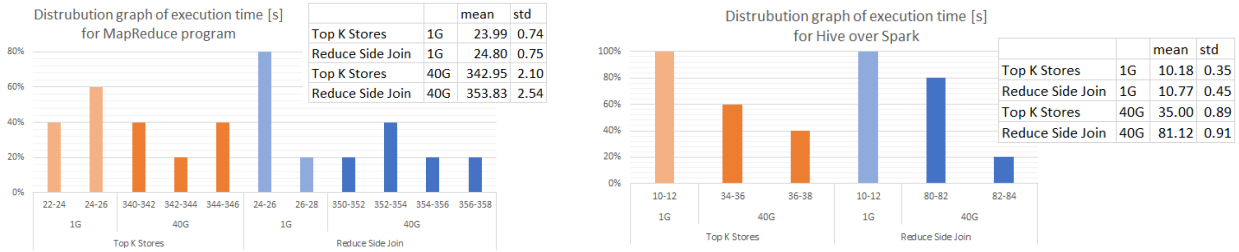

Figure 5: Mean, standard deviation and distribution of execution time for the two tasks (Top K Stores, Reduce Side Join) and two data sets (1G, 40G) on separate plots for both MapReduce and Hive over Spark.

It is important to note that the time efficiency of both programs is hardware dependent, therefore, their performance may vary based on the machine executing the code or workload on the server. To guarantee the most precise measurements, all queries were run on the same machine and it was ensured that no other expensive processes were running. Nonetheless, the aim of presented results is to analyze the performance of the approaches in contrast to the each other, rather than give objective measurements.

4 Conclusion

In conclusion, the central aim of this project was to present the two different approaches to Big Data analytics implemented by MapReduce over Hadoop and Hive over Spark as well as their efficiencies and caveats. We consider that our results clearly present their differences and similarities. On one hand, Hadoop with MapReduce has the potential to be the more efficient approach if the programmed code utilizes ingenious data structures as well as fast and complex algorithms. This is, however, unlikely due to the inherent complexity of the MapReduce framework and the Hadoop platform, that makes writing complex and properly optimized code a rather tedious and difficult task. On the other hand, as it can be observed from the provided source code for both tasks, the Hive implementation of the same query is often much simpler, and therefore less likely to exhibit unwanted behaviour due to human error.

If provided more time, one of the first changes that we would make would be regarding the two solutions' performances. Both the generated MapReduce code, as well as the HiveQL query, could be optimized in various ways. Better data structures and algorithms could be implemented in the context of the MapReduce code, that would improve the program's performance as well as its memory footprint. Similarly, various query optimization techniques (e.g. applying selections and projections before joins) could be employed to speed up the execution time of the query. Additionally, we could improve the verbosity of the produced MapReduce code and adjust it to handle various types of exceptions (e.g. strings where a number would be expected) more gracefully.

References

[1] Mapreduce. https://www.tutorialspoint.com/hadoop/hadoop_mapreduce.htm. [Online; accessed 30-November-2018].

[2] D. Dworin. Data science revealed: A data-driven glimpse into the burgeoning new field, 2011.

[3] H. Ferhatosmanoglu. Course material, November 2018.

[4] D. Laney. 3-d data management: Controlling data volume, velocity and variety, February 2001.

[5] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 19 April 1965.

[6] J. Venner. Pro hadoop. *Apress*.