CS346 Project Report

Group 33
Andreas Petrou, Marios Vouryias, Amar Aziz

March 23, 2022

1 Introduction

This report outlines a comparison between the Hadoop MapReduce and Apache Hive frameworks based on the TPCDS [1] dataset. In particular, the following sections describe any Exploratory Data Analysis performed to learn more about the data, and how we implemented the requested queries using both methods. In addition, performance analysis conducted using several runs of both our MapReduce programs and HiveQL queries is explained, which was used to compare the two approaches and decide when each should be used.

2 Exploratory Data Analysis

For our Exploratory Data Analysis (EDA), we used only attributes which were requested by the queries. These attributes include: *Date, Item* and *Store*.

2.1 Dates

To analyse data on the Dates, we used 10% of the records, 20% of the records, 50% of the records, 70% of the records and 100% of the records. We did that to check if the distribution of dates changes depending on the portion of data we use. To plot the data, we used a histogram with 100 bins. After some testing, 100 seemed a reasonable amount, as it is large enough to provide a detailed graph but is not too large at the point where analysing the data is cumbersome. The following plots display the histograms obtained by using the date field from the dataset:

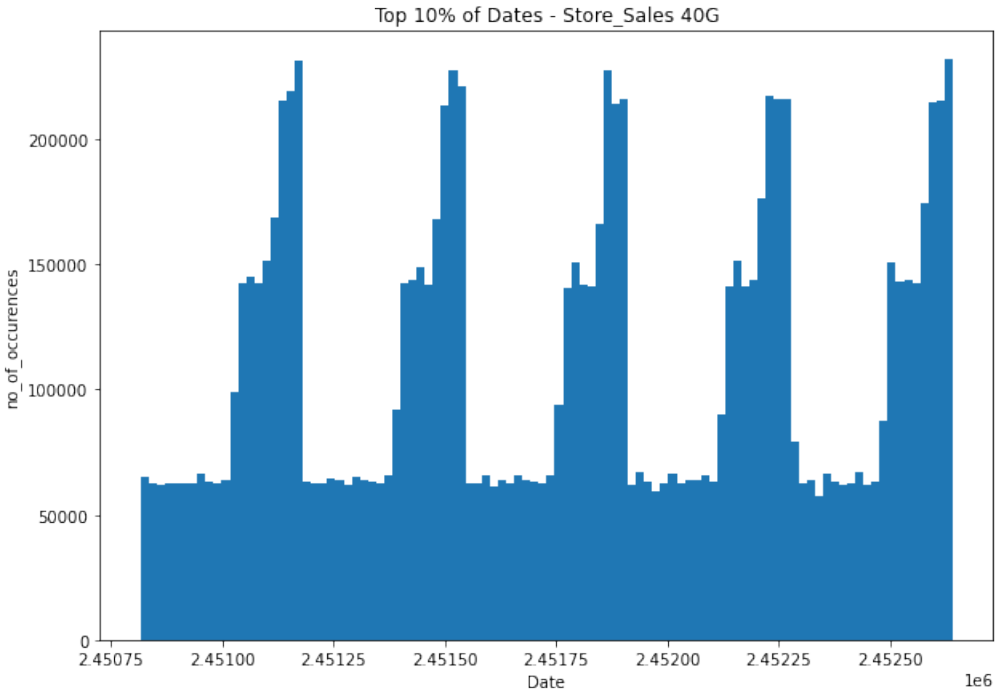

Figure 1: Histogram for dates using 10% of data

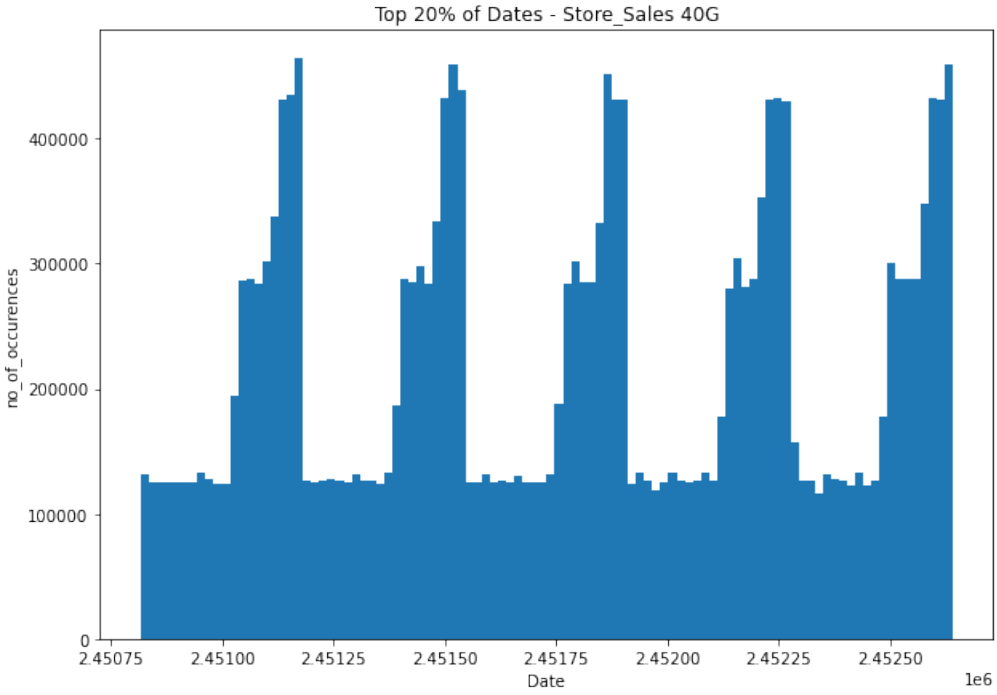

Figure 2: Histogram for dates using 20% of data

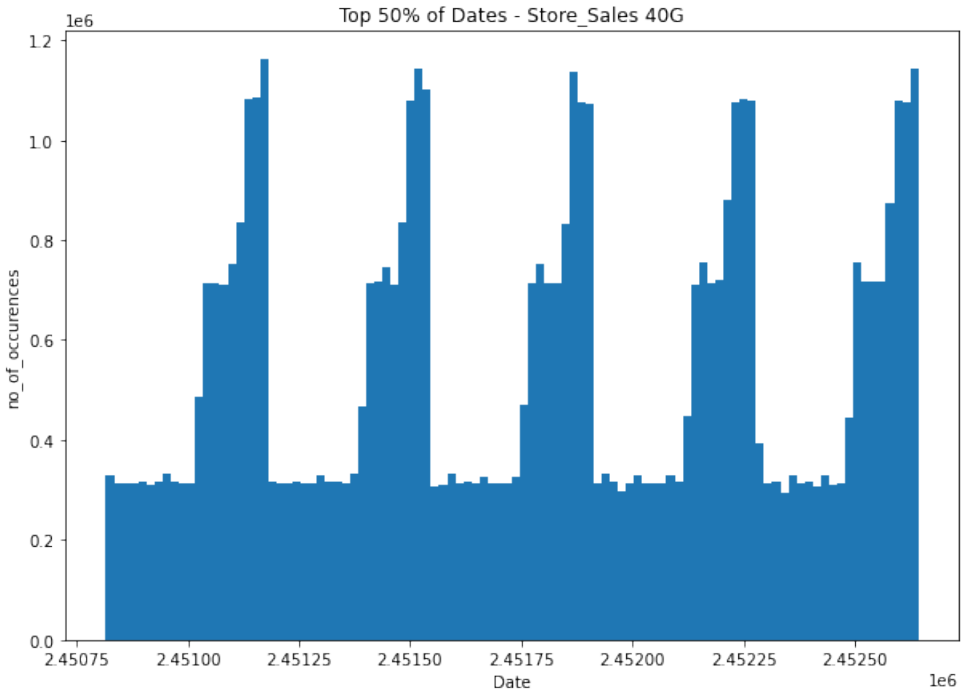

Figure 3: Histogram for dates using 50% of data

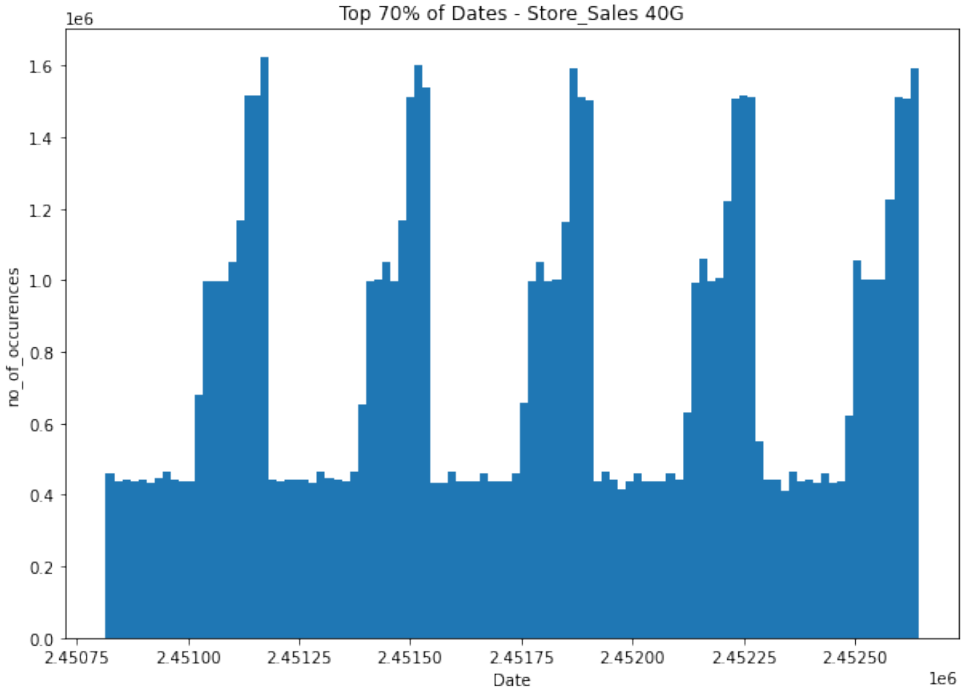

Figure 4: Histogram for dates using 70% of data

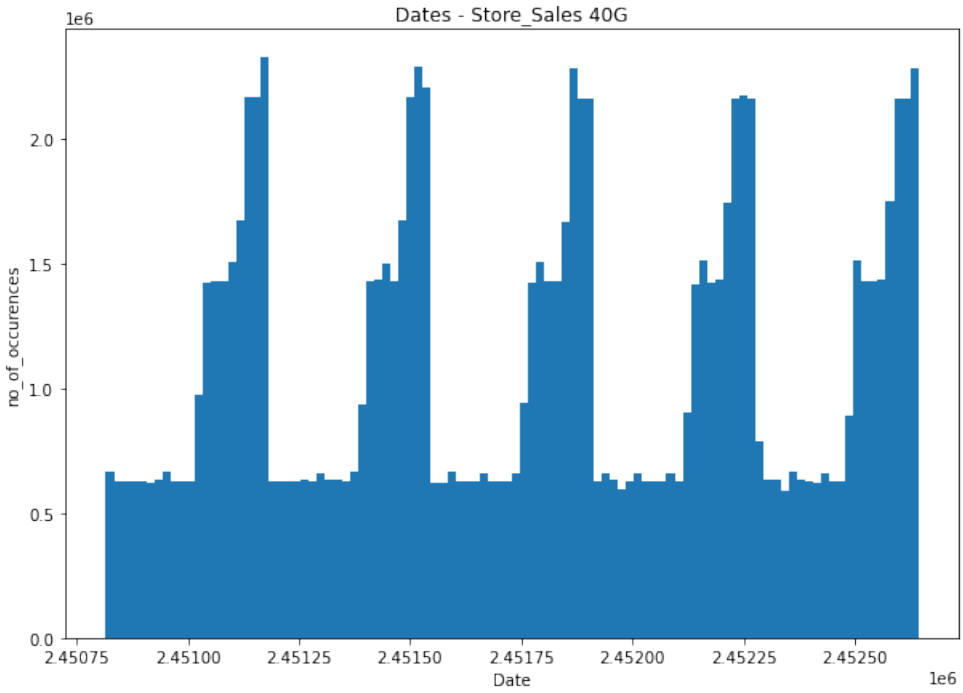

Figure 5: Histogram for dates using 100% of data

Viewing the graphs above, it can be noticed that the distribution of dates hardly changes, even when different portions of the dataset are used. Further analysis of dates shows:

```
minimum date is:  2450816 (1998-01-02)
maximum date is:  2452642 (2003-01-02)
range of days:  1826
```

From these results, it can be observed that 1826 unique dates are used. Therefore, each bin of the histogram used 182.6 days, which rounds down to 182 days (100 bins used). One can either use one or multiple bins to run queries and see how this affects runtime. Overall, each figure follows a pattern where the number of occurrences for each bin hardly changes until it suddenly starts to peak. As a result, it can be concluded that the season may affect the number of sales the stores make.

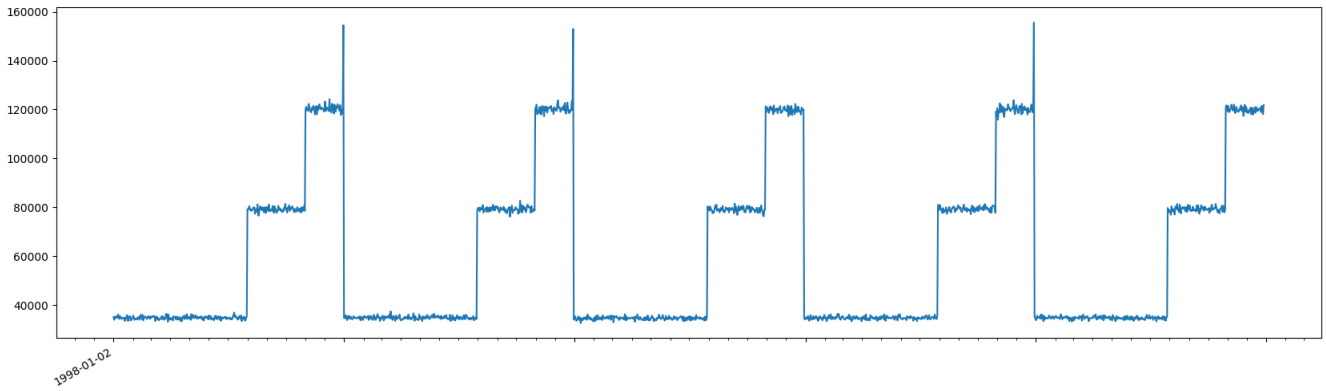

Figure 6: Line graph showing frequency of each individual date

By plotting a graph of the frequencies of each individual date, it can be shown that the number of sales every year follows a strict pattern:

- From 3rd January to 2nd August, sales per day are always between 32000 and 38000
- From 3rd August to 2nd November, sales per day are always between 76000 and 83000
- From 3rd November to 1st January, sales per day are always between 115000 and 124000

The most notable exception is on 2nd January, where in the years 1999, 2000 and 2002, sales peak to values between 152000 and 156000.

Interestingly, each of the three periods of low, medium or high sales have roughly the same amount of total sales during the period. This means each year's sales can be split into three partitions of similar sizes at the boundaries between the different periods. This fact could be useful for implementing partitioners for MapReduce jobs which use dates as the key, such as query 1.c.

2.2 Items

This section demonstrates the analysis of the item attribute within the dataset. First, a bar chart was used to check the distribution of items:

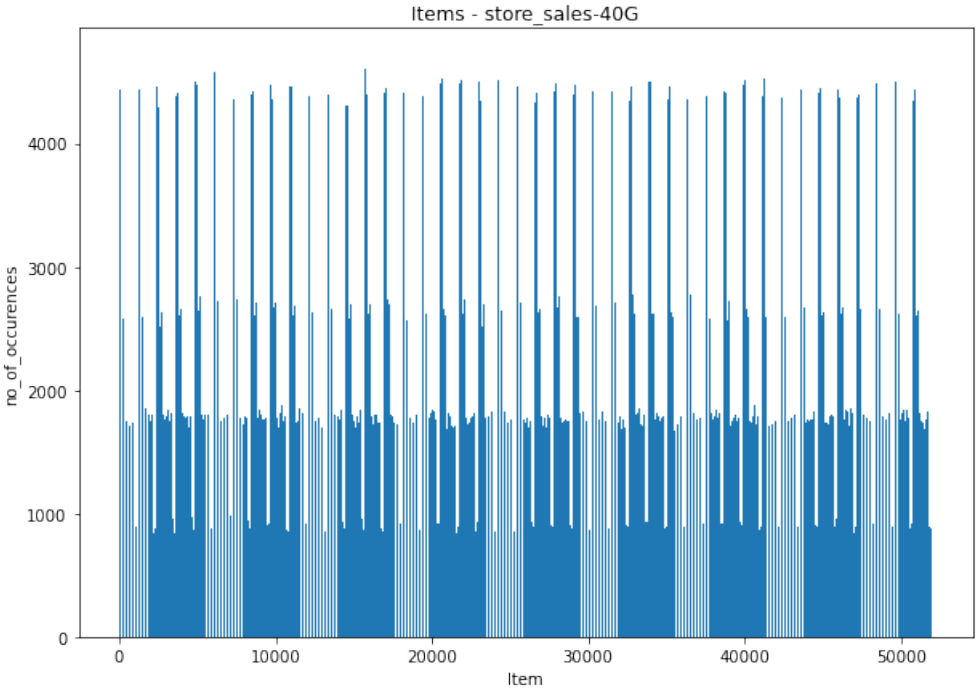

Figure 7: Bar chart obtained from using the item field of the dataset

Since the dataset contains a huge number of items, it was not easy to come to conclusions based on the chart. Therefore, different statistics were performed on the dataset, and the following results were produced:

```
mean of occurrence of each item:  2215.450384615385
variance of occurrence of each item:  1242093.311999871
standard deviation of occurrence of each item:  1114.4924010507523
```

Partitioners can be useful when the number of occurrences of different items vary. The standard deviation of this data-set suggests that the use of a partitioner may benefit query 1.b, which is the query that outputs the top highest selling items.

2.3 Store

Unlike the item field, store occurrences did not vary much.

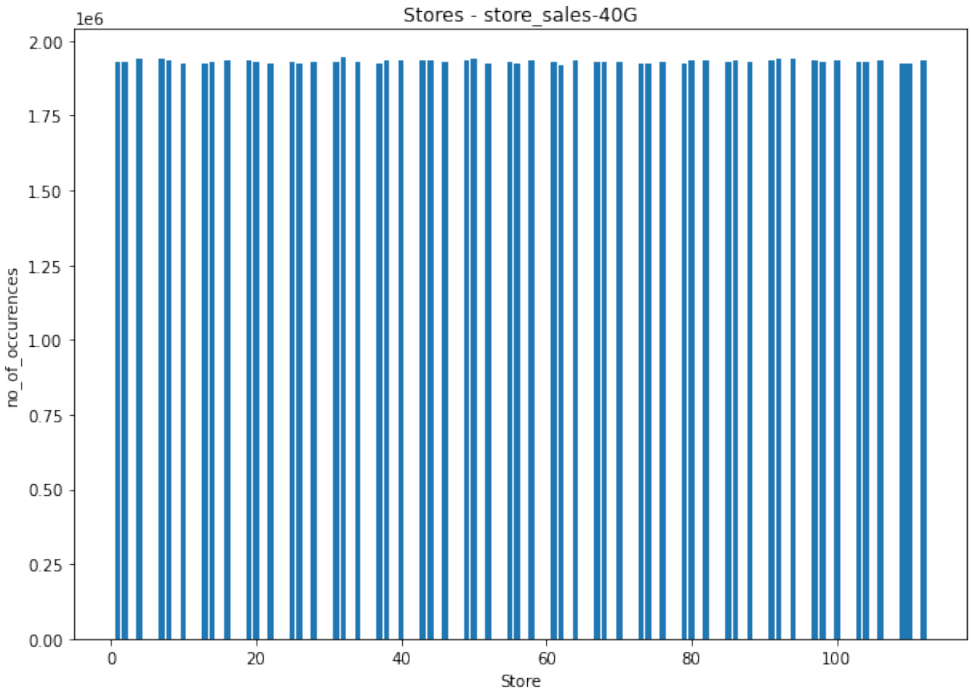

Figure 8: Bar chart obtained from using the store field of the dataset

```
mean of occurrence of each store:  1930221.9649122807
variance of occurrence of each store:  25037509.08648815
standard deviation of occurrence of each store:  5003.749502771711
```

In addition, the standard deviation of the data was small compared to the mean value, indicating that the spread of the data is small. As a result, it is less likely that performance, for any of the four queries, will benefit from the use of partitioners.

3 Query Implementation

This coursework required two different major queries. The first query required data from the *Store_Sales* table, while the second query was a Reduce Side Join query that required both the *Store_Sales* and the *Store* tables.

3.1 Query 1: Simple queries

After running some SQL commands on Hive for the first set of queries, we observed that it required two MapReduce jobs. This is because after manipulating the data, the second job was dedicated to sorting the data as required by the coursework specification. We will explain the basic implementation of queries **1.a**, **1.b** and **1.c**, and then mention any specific modifications and adjustments made in order for the queries to run properly.

For the first MapReduce job, we took the records in the table, which are processed in the form of key-value pairs, and used the Mapper function to take the required fields, which are also in the form of key-value pairs. After the mapper job finished, during the sort and shuffle phase, the values were aggregated according to their keys. The Reducer took the tuples of values, and for each unique key, iterated through the list of its values and added them together. This calculation produced a new value that was output by the Reducer.

The second job took the output of the Reducer of the first job and sorted the results using a TreeMap structure. This particular structure was chosen as the data it stores is already sorted. It may take longer to add

elements to such a structure in comparison to lists and hash maps (insertion requires $O(log(n))$), but we save time by not sorting the data after adding them to our data structure. In other words, if Merge Sort is used to sort a list, then all records will be added to the list in $O(n)$ time. Then, Merge Sort is done in $O(nlog(n))$ time. So we have a sorted list in $O(n + nlog(n))$ time. On the other hand, when TreeMaps are used, we have a sorted list in $O(nlog(n))$ time. We tested this theory with two different implementations of the first query. One implementation used TreeMaps, while the other used lists of a custom type *Record* (see Appendix). It should be mentioned that TreeMap elements are in the form of key-value pairs. For the queries of the first part of the exercise, the keys were the attribute that we wanted to be sorted, while the value was the attribute being aggregated.

3.1.1 Query 1.a

The first Mapper takes as input a record from the table and splits it into its individual attributes. Then, this record is checked to make sure it has at least 21 attributes that are not NULL (the index of the last attribute we want is 20 if we use a number system that starts from 0), and that the fields required do not contain *NULL* values. The fields required for this query are the *date, store number* and *revenue*. The **map** function ensures that the date is within the given range provided by the arguments of the program. Moreover, it produces a key-value pair of the form: *Store-Revenue*. The output of the Mapper is grouped by the store, and then the Reducer, for each store, iterates through its tuple of revenues, finds the sum of all the revenues within the tuple and stores the result. It then outputs its own key-value pairs in the form of *Store-Result*, where the result is calculated beforehand and is the sum of all the revenues within a tuple of values.

For sorting the data, the TreeMap and the number of records needed were initialised with the *setup* function. The TreeMap took *Revenue* as key and *Store* as value since we want our records to be sorted in descending order of revenue. If the length of the TreeMap exceeded parameter N, which defines the number of records we want to retrieve, then we removed the first key of the TreeMap, as this element had the smallest value of Revenue. Even though it is not wise to sort the data during the map stage, as it will be shuffled when aggregating the values, using TreeMap in the Mapper and the Reducer classes may be better since we will not need all the records for the Reduce stage but only the top N records, which are already found during the Map stage.

3.1.2 Query 1.b

Both query **1.a** and query **1.b** had very similar implementations. One notable difference, however, is that the key-value pairs output by the first Mapper are of the form: *Item - Quantity* and the Reducer of the first job sums up the quantities within a single tuple instead of revenue like in the first query. In addition, the key of the TreeMap this time is of integer type and is used to store the quantities sold.

3.1.3 Query 1.c

For the last part of the first query exercise, 1.c, the first mapper output pairs of the form: *Date-Revenue with tax* and used a predefined date range to find the revenue, including tax, obtained for a single day.

3.2 Query 2: Reduce Side Join

For the last query, an array list of type *Record* (see Appendix) was used instead of a TreeMap for sorting. This is because the last query required two attributes to be sorted instead of just one. The records had to be sorted in descending order of *Floor Space*, then descending order of *Revenue*. To sort based on two attributes, we overrode the **compare** function of the **Comparator** [3] class in java. In addition, we used two different Mapper classes, one for each table. The first Mapper class was for the *Store Sales* table, while the second Mapper class was for the *Store* table. The *Store Sales* Mapper output tuples of the form: *store_number, "sales" + net_paid*. It should be noted that the values output by both Mappers include the name of the table so that the **Reducer** knows which table the value comes from. The output of the second Mapper class is of the form: *store_number, "store" + floor_space*. What can be observed is that the key output of both Mapper classes is the same. This is due to the fact that the key output when two Mapper classes are used is the join condition. For the Reducer class, two different variables were required. One for the *floor_space* and another one for the *net_paid*. When iterating through the tuple of values output by the Mappers, the Reducer checks whether that value comes from the *Store Sales* table or the *Store* table. If the value is from *Store Sales*, it **adds** that value to the total revenue obtained by the store. On the other hand, if the value comes from the *Store* table, it **sets** that value to the floor space of the corresponding floor. The second job of this query consists of one Mapper class and one Reducer class like the previous queries. This second job is

responsible for sorting the output of the query in descending order of *floor_space*, then descending order of *net_paid*. The output of the first job was stored in a list of type *Record* and Merge Sort was used to sort the data. Therefore, as mentioned in Section 2.1, we obtained the desired output in $O(n + nlog(n))$ time.

4 Performance Analysis

To evaluate the performance of our Hadoop programs, we recorded the total *CPU time spent* for their MapReduce jobs to run with and without a combiner, (as well as a partitioner for query 1.c) and then compared it with the time it took to run the corresponding queries in **Hive**. Apart from the runtime of the programs, we took into consideration *MB-seconds* (the amount of memory allocated to run the MapReduce job multiplied by the number of seconds required to run the program) and *vcore-seconds* (the number of vcores allocated by the program multiplied by the time, in seconds, it took for the program to finish execution) [7]. In addition, after analysing the Date field of the dataset, we observed that there were certain months where income was less than other months. As a result, we split months into three categories; low, mid and high. Low months were months with revenue around 35000 per day, mid months were months with revenue around 80000 per day, and high months were months with revenue around 120000 per day.

Tables 1 - 3 illustrate the various results obtained by both Hive and Hadoop after running the queries.

Query	Start Date	End Date	Time (ms)	MB-seconds	vcore-seconds	Notes
1.a	2451146	2452268	595,510	5,587,403	4,650	Given Dates
1.a	2450848	2450876	582,260	5,470,327	4,560	Low month
1.a	2451060	2451090	564,620	3,916,744	3,216	Mid month
1.a	2451121	2451151	552,440	3,720,511	3,068	High month
1.b	2451146	2452268	762,050	5,933,321	4,969	Given Dates
1.b	2450848	2450876	623,350	3,888,719	3,226	Low month
1.c	2451392	2451894	597,670	4,074,812	3,389	Restricted Dates
2	2451146	2452268	622,160	3,283,013	2,712	Given Dates
2	2450848	2450876	563,220	2,872,179	2,364	Low Month
2	2451060	2451090	580,760	3,291,727	2,704	Mid Month
2	2451121	2451151	603,190	4,339,089	3,606	High Month

Table 1: Hive Results

Query	Start Date	End Date	Time (ms)	MB-seconds	vcore-seconds	Notes
1.a	2451146	2452268	544,020	4,226,793	3,510	Given Dates
1.a	2450848	2450876	443,220	2,963,673	2,424	Low month
1.a	2451060	2451090	470,580	3,418,978	2,827	Mid month
1.a	2451121	2451151	469,550	3,397,776	2,807	High month
1.b	2451146	2452268	616,730	3,790,527	3,141	Given Dates
1.b	2450848	2450876	459,240	3,326,929	2,743	Low month
1.c	2451392	2451894	524,420	2,797,730	2,296	Required Dates
1.c	2451392	2451894	529,870	3,395,537	2,795	2 partitions, default partitioner
1.c	2451392	2451894	556,690	4,126,707	3,426	2 partitions, custom partitioner
1.c	2451392	2451894	552,660	4,942,579	4,127	5 partitions, default partitioner
1.c	2451392	2451894	585,890	5,852,466	4,895	5 partitions, custom partitioner
2	2451146	2452268	540,870	3,224,058	2,662	Given Dates
2	2450848	2450876	415,840	3,459,873	2,855	Low month
2	2451060	2451090	435,190	2,159,618	1,756	Mid Month
2	2451121	2451151	444,430	3,144,857	2,595	High Month

Table 2: Hadoop Results (with Combiner)

Query	Start Date	End Date	Time (ms)	MB-seconds	vcore-seconds	Notes
1.a	2451146	2452268	586,780	5,111,381	4,242	Given Dates
1.a	2450848	2450876	436,360	3,682,895	2,044	Low month
1.a	2451060	2451090	454,790	4,357,664	3,630	Mid month
1.a	2451121	2451151	443,230	2,782,623	2,289	High month
1.b	2451146	2452268	620,660	3,731,712	3,075	Given Dates
1.b	2450848	2450876	453,410	3,596,804	2,976	Low month
1.c	2451392	2451894	521,700	2,981,700	2,444	Required Dates
2	2451146	2452268	573,260	4,125,298	3,401	Given Dates
2	2450848	2450876	408,300	2,677,255	2,173	Low Month
2	2451060	2451090	419,420	2,319,647	1,890	Mid Month
2	2451121	2451151	423,060	2,503,170	2,044	High Month

Table 3: Hadoop Results (without Combiner)

Overall, our query implementations seem to outperform Hive in terms of runtime.

The data indicates that the use of a combiner can be beneficial depending on the query and the range of dates used. For instance, when running **query 1.a** with dates 2451146 and 2452268 (1123 days) or 2450816 and 2452642 (1826 days), we get better results by using a combiner rather than not using a combiner at all. However, this is not the case when for the same query, we use the dates 2450848 and 2450876 (29 days) or 245160 and 245190 (31 days). This suggests that combiners are beneficial when there is a large amount of data to process, but otherwise are detrimental. This knowledge could have possibly been used to enhance our implementations by deciding at runtime whether or not to use a combiner depending on the amount of data.

Moreover, the data shows that the use of a partitioner is detrimental in query 1C. Increasing the number of partitions results in slower performance, which could be due to the fact that the cluster that executes the query consists of only one core, so has limited parallelism capabilities. This slowness is worsened by using a custom partitioner which separates the dates into the different time periods over Hadoop's default partitioner, which uses a simple hash function, showing that our custom partitioner was either too inefficient or too inaccurate.

Furthermore, the data suggests that the size of the range of dates affects performance whilst the number of sales during those dates does not. Queries executed with Hadoop show significantly better performance when reducing the range of dates used, compared to Hive where the difference is less noticeable.

5 Comparison of Frameworks

To compare Hive and Hadoop, we should consider various things, such as the resources (CPU, RAM) available, time constraints for development, how important performance is, and how big the dataset we want to query is.

Familiarity with how MapReduce works is essential in order to use Hadoop [4]. However, Hadoop is more flexible as you can customise your own partitioner. Although Hive provides the user with the option to specify which fields to partition [5], it is not as detailed as a partitioner class in Java. It is worth mentioning that EDA can be beneficial before deciding to use a partitioner, as you can find patterns in specific fields, and from these patterns, you can determine whether a query can benefit from a partitioner or not. Furthermore, with Hadoop, you have the option to choose what structures to use for sorting or storing data, whereas with Hive, you just use an SQL-Like language to query one or more datasets.

In the real world, when considering which of the two frameworks to use, it is essential to consider the knowledge of the workforce. For instance, many people are familiar with SQL but do not know how MapReduce works, as it is not as popular. Suppose a programmer is not confident in using MapReduce in Java. In that case, development may stall, especially when hundreds of lines of Java code may be required to produce a query that can be written in 8-10 lines of Hive Query Language (HQL). This can be an issue, especially if a large number of different queries have to be conducted on a single dataset. It is nearly impossible for someone to write a large number of programs just to query data. Finally, a major difference between the two is that Hadoop can be used without Hive, whereas it is difficult to use Hive without Hadoop [6]. On the other hand, using both technologies together can make processing big data very easy.

6 Conclusion

To summarise, we successfully implemented the queries required for this coursework in Java, using Hadoop. Exploratory Data Analysis proved very helpful for deciding what date ranges to use. By comparing our programs with Hive, we can safely say that our queries offer better performance.

References

[1] TPC: *TPC-DS Version 2 and Version 3*
https://www.tpc.org/tpcds/
Accessed: 23/03/2022

[2] APACHE: *APACHE HIVE*
https://hive.apache.org/
Accessed: 23/03/2022

[3] Oracle: *Interface Comparator*
https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html
Accessed: 23/03/2022

[4] Geeks for Geeks: *Difference Between Hadoop and Hive*
https://www.geeksforgeeks.org/difference-between-hadoop-and-hive/
Accessed: 23/03/2022

[5] Data Flair: *Hive Partitions, Types of Hive Partitioning with Examples*
https://data-flair.training/blogs/apache-hive-partitions/#:~:text=Apache%20Hive%20makes%20this%20job,of%20partition%20column(s).
Accessed: 23/03/2022

[6] Priya Pedamkar: *Hadoop vs Hive*
https://www.educba.com/hadoop-vs-hive/
Accessed: 23/03/2022

[7] Manjunath Ballur: *Aggregate Resource Allocation for a job in YARN*
https://stackoverflow.com/questions/33866888/aggregate-resource-allocation-for-a-job-in-yarn/33868015#33868015
Accessed: 23/03/2022

7 Appendix

The class below can be modified to use different numbers of fields, as well as different fields.

```
  /**
 * The record class
 */
public static class Record {
  private String storeNumber;
  private Long floor;
  private Double netPaid;

  // class constructor
  public Record(String storeNumber, Double netPaid, Long floor) {
    this.storeNumber = storeNumber;
    this.netPaid = netPaid;
    this.floor = floor;
  }
  /**
   *
   * @return store number
   */
  public String getStoreNumber() {
    return storeNumber;
  }
  /**
   *
   * @return floor space
   */
  public Long getFloor() {
    return floor;
  }
  /**
   *
   * @return net paid
   */
  public Double getNetPaid() {
    return netPaid;
  }
}
```

Figure 9: The *Record* class