CS346 - Advanced Databases Coursework Report

János Hajdu Ráfis	Jon Winfrey	Valentin Kodderitzsch
1935562	1907645	1931737

March 2023

1 Hive

Hive is a "distributed, [...] data warehouse system that enables analytics at a massive scale [...] using SQL" [5]. This allows for running SQL queries on large amounts of data, where queries are translated into Hadoop MapReduce functions.

The first implementation of the coursework queries was done in Hive, as group members were already comfortable with SQL. Implementing these first would also give a comparison point for verifying the correctness of MapReduce queries. The Hive implementation began with creating a schema for Hive to interpret the data files. The most interesting part of this was how quickly the tables were able to be created, showing that Hive is skipping many of the checks that a traditional relational database management system (RDBMS). This was also seen with how it didn't support some of the constraints that would usually be used in a SQL CREATE TABLE statement, such as primary keys and NOT NULL. In the context of hive and big data, this makes sense though, as it isn't importing or checking any new data, but simply saving a format for the data files when they are used in later map-reduce jobs.

The queries were then created, with the only major difficulty being in query 1B, where row numbers and partitions were used. This was done to allow for the least M-selling items for a store to be displayed properly.

The following section contains all four Hive SQL queries and their top 10 results as required by the coursework specification.

1.1 Q1A

Query 1A asks for the top $K = 10$ stores with the smallest revenue in ascending order for a given date range (inclusive). The SQL code is shown in figure 1 and the corresponding results are shown in figure 1a.

```
SELECT "ss_store_sk_" || ss_store_sk as ss_store_sk, sum(ss_net_paid) as revenue
FROM store_sales
WHERE ss_store_sk != "" AND ss_net_paid IS NOT NULL AND ss_sold_date_sk != ""
  AND ss_sold_date_sk BETWEEN [StartDate] AND [EndDate]
GROUP BY ss_store_sk
ORDER BY revenue ASC
LIMIT [K];
```

Figure 1: Hive: Query 1A

1.2 Q1B

Query 1B asks for the M least-selling items (by quantity) in ascending order from the N highest-selling stores in descending order for a given date range (inclusive). The SQL code is shown in figure 2 and the corresponding results are shown in figure 1b. Note that this query makes the assumption that items that have not been sold at all are discarded.

```
SELECT "ss_store_sk_" || t.store AS ss_store_sk,
    "ss_item_sk_" || t.ss_item_sk AS ss_item_sk,
    t.quantity_count AS total_quantity
FROM (  -- lowest selling items from the highest selling stores
        SELECT outer_ss.ss_store_sk AS store,
            outer_ss.ss_item_sk,
            outer_ss.quantity_count,
            ROW_NUMBER() OVER (
                PARTITION BY outer_ss.ss_store_sk
                ORDER BY outer_ss.quantity_count ASC
            ) AS RNUM
        FROM (  -- Least sold items
                SELECT lsi_ss.ss_store_sk,
                    lsi_ss.ss_item_sk,
                    SUM(lsi_ss.ss_quantity) as quantity_count
                FROM store_sales as lsi_ss
                WHERE lsi_ss.ss_store_sk != ""
                    AND lsi_ss.ss_sold_date_sk BETWEEN [startDate] AND [endDate]
                    AND lsi_ss.ss_quantity IS NOT NULL
                GROUP BY lsi_ss.ss_store_sk,
                    lsi_ss.ss_item_sk
            ) AS outer_ss
        WHERE outer_ss.ss_store_sk IN (
                -- Highest selling stores
                SELECT hss_ss.ss_store_sk
                FROM store_sales AS hss_ss
                WHERE hss_ss.ss_store_sk != ""
                    AND hss_ss.ss_sold_date_sk BETWEEN [startDate] AND [endDate]
                    AND hss_ss.ss_net_paid IS NOT NULL
                GROUP BY hss_ss.ss_store_sk
                ORDER BY SUM(hss_ss.ss_net_paid) DESC
                LIMIT [N] -- NOTE change limit N here
            )
    ) AS t
WHERE t.RNUM <= [M];
```

Figure 2: Hive: Query 1B

1.3 Q1C

Query 1C asks for the top $K = 10$ days within a given date range (inclusive) that have the highest net paid tax ("ss_net_paid_inc_tax"). Figure 3 shows the SQL code for query 1C and the corresponding results are shown in figure 1c.

```
SELECT "ss_sold_date_sk_" || ss_sold_date_sk, SUM(ss_net_paid_inc_tax) AS total
FROM store_sales
WHERE ss_sold_date_sk BETWEEN [startDate] AND [endDate]
GROUP BY ss_sold_date_sk
ORDER BY total DESC
LIMIT [K];
```

Figure 3: Hive: Query 1C

1.4 Q2

Query 2 asks for the floor space of the top $K = 10$ stores with the highest net pay. The SQL code for this query is shown in figure 4 and the results are shown in figure 1d.

2 Map Reduce

The coursework queries were also implemented using Map Reduce, a "programming paradigm that enables massive scalability across hundreds or thousands of servers in a Hadoop cluster" [6]. Using this framework,

```
SELECT S.s_store_sk AS store, SS.ss_net_sum AS net_sum, S.s_floor_space AS floor_space
FROM store S
INNER JOIN (
           -- Stores with the highest net pay
           SELECT ss_store_sk, SUM(ss_net_paid) AS ss_net_sum
           FROM store_sales
           WHERE ss_sold_date_sk BETWEEN [startDate] AND [endDate]
           GROUP BY ss_store_sk
           ) SS ON S.s_store_sk = SS.ss_store_sk
ORDER BY SS.ss_net_sum DESC, S.s_floor_space DESC
LIMIT [K];
```

Figure 4: Hive: Query 2

initial data files can be split up and worked on by a large number of *mappers*, which can map a function to each row. These mappers then send key-value pairs to reducers, which can perform operations for every value associated with each key received.

2.1 Issues in Floating-Point Arithmetic

Many of the queries required used the `ss_net_paid` field within the "store sales" table, which stores monetary amounts based on each sale. The initial implementation attempted for the 1A MapReduce query parsed these strings as single-precision floating point values. This appeared to work initially, but led to the aggregate values for `ss_net_paid` to be very close to being correct, but incorrect after around the 5th decimal place. This was attempted to be remedied by using double-precision floating point values, but the problem persisted with occasional input values being one pence off, and thereby incorrect aggregate values. To remedy this, the map-reduce tasks store all monetary values as 64-bit integers (longs) in terms of pence, their smallest increment. Parsing using doubles also was insufficient for correct results (due to the values occasionally being one pence off), so Java's BigDecimal [7] class was used for parsing and conversion to and from longs. This allows for accurate storage of decimal values throughout.

Apache MapReduce has the constraint that objects passed between mappers and reducers are 'writable' through inheritance. Classes are included in the library for basic types, but not for any class. With this approach for monetary values, LongWritable objects were passed between mappers and reducers. An alternative to this would have been to create writable wrappers for BigDecimal, but this was decided against as operations can be completed much faster for longs, and the values were guaranteed to be accurate without any issues with decimal parts. It was also assumed that the Hadoop implementation would be optimised for the basic types that were provided.

2.2 String Type For Fields

Some of the data fields, such as "ss_store_sk" and "ss_sold_date_sk" only ever stored values formatted as integers, but were used as Strings within the implementation. This was done for compatibility with the TPC-DS dataset format, which specified that these fields were identifiers, and as such should be able to hold "any key value generated for that column" [2, p. 17]. It is also notable that these fields didn't use the integer type defined by the TPC-DS documentation [2, p. 17], so it could be expected that over-values were possible. In our dataset the fields only contained integers, but due to this data documentation, values were interpreted as strings, with the only exception being for date ranges, where they had to be interpreted as integers.

2.3 Sorting

Since some of the queries required orderings based on the fields, sorting was required. In order to use Hadoop for these sections rather than external tools, additional MapReduce jobs were added. These contained a simple mapper which took in the data and output key-value pairs where the key was the item being sorted on, and all other fields were sent within the value. This allowed for the use of Hadoop sorting where the key-value pairs could be ordered between the map and reduce phases, and received in order.

2.4 Efficient Inter-Query Data Storage

All of the queries required multiple MapReduce jobs to complete properly in order to not use external tools for e.g. sorting. Unfortunately, the framework doesn't have a specific way to pass information from one reducer to

Store	Revenue
ss_store_sk_62	3182109769.33
ss_store_sk_109	3186596761.32
ss_store_sk_26	3189149422.15
ss_store_sk_56	3190660096.15
ss_store_sk_22	3191957487.69
ss_store_sk_13	3193239275.04
ss_store_sk_37	3194961275.76
ss_store_sk_55	3195322093.18
ss_store_sk_10	3195664092.24
ss_store_sk_79	3195685931.79

(a) Query 1A Results

Store	Item	Total Quantity
ss_store_sk_32	ss_item_sk_13788	74
ss_store_sk_32	ss_item_sk_23502	91
ss_store_sk_50	ss_item_sk_19278	86
ss_store_sk_50	ss_item_sk_50562	101
ss_store_sk_92	ss_item_sk_49206	95
ss_store_sk_92	ss_item_sk_6006	97

(b) Query 1B Results using $N = 3$, $M = 2$

Day	Total revenue
ss_sold_date_sk_2452277	272192985.36
ss_sold_date_sk_2451181	270862231.93
ss_sold_date_sk_2451546	269011207.69
ss_sold_date_sk_2451522	218299535.32
ss_sold_date_sk_2451159	217391100.15
ss_sold_date_sk_2451544	216896013.28
ss_sold_date_sk_2452245	216673725.55
ss_sold_date_sk_2451152	216297854.41
ss_sold_date_sk_2451172	215983973.13
ss_sold_date_sk_2451537	215618328.78

(c) Query 1C Results

Store	Revenue	Floor space
32	3226933911.31	6131757
92	3226481169.88	8573853
50	3223393872.18	7825489
8	3219565087.71	6995995
4	3219464747.31	9341467
40	3216651245.14	6465941
94	3216537792.35	9599785
58	3216346192.93	8393436
16	3214457245.02	5633347
49	3214108743.30	9206875

(d) Query 2 Results

Table 1: Results using $K = 10$, $startDate = 2450816$ and $endDate = 2452642$

the next mapper other than outputting the file to HDFS and loading it again into the next MapReduce job.

A naive approach for this intermediate storage would be to output all values as text in the file (as would normally be done for final outputs). This comes with a large downside for integer values though, as it means that in the next mapper, every value needs to be parsed from text back into a binary datatype that can properly represent an integer and be worked with. This conversion comes with a performance penalty, which could grow considerably in the context of MapReduce, where the amount of data being processed can be so large. Because of these issues, another file output type was used: SequenceFiles. These allow for binary output of data, meaning that data can then be loaded into mappers without any additional conversions. Another benefit is that they allow for keys and values to be specified and thereby a cleaner software architecture. Using this format means that the next mapper in the chain can receive information as keys and values. This is different from the text format where file lines are passed in as string values, with the key value being essentially unused.

2.5 Query 1A

Query 1A was implemented with two separate MapReduce jobs, and used the "store sales" data. Firstly, the aggregate values for `ss_net_paid` were calculated (with the mapper filtering out rows to wanted columns and valid rows within the date range). These were then sorted with the K highest being taken in the next MapReduce job. These had to be implemented as separate jobs, as the total revenue had to be found before the results were sorted by it.

2.6 Query 1B

Query 1B introduced significant challenges on the MapReduce side just like the HiveQL side. Unlike the other queries in SQL, it contained multiple sub-queries which meant that data from one set of jobs have to be used in another set of jobs that are not the input data.

The query is implemented with 4 MapReduce jobs. Two jobs for finding the N highest-selling stores, where the first job maps the `ss_store_sk` with the `ss_net_paid` and reduces by summing the net paid values, then another job is used to find sort and find the N highest. In this query, the sorting is implemented by a different

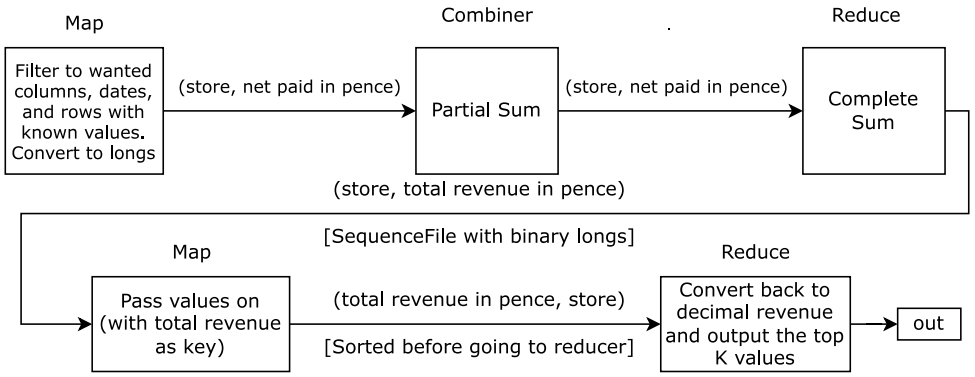


Figure 5: MapReduce architecture for Query 1A

method compared to the other queries. For finding the N highest-selling stores the whole data is not actually sorted in order to optimize performance. The implementation uses a `PriorityQueue` with a fixed size, in case the priority queue is not full the item is simply added to it. However, in case it is full it compares the item at the top of the queue with the current one and if it satisfies the sorting condition - in the case of highest selling stores it's higher than the lowest value in the queue - it removes the item and adds the new to the queue. This approach of sorting is efficient for smaller numbers but scales badly when values tend to the number of rows, however, it is assumed that due to the nature of the coursework, aggregating large amounts of data, limiting values are expected to be small.

After sorting, the output of this first section has to be accessible by the second pair of jobs. Unfortunately, this is not as simple as storing the result of the first section in global variables as the MapReduce jobs have no access to these. This issue was solved by reading the data from the output file of the first section in Hadoop and then storing the result in the configuration that the jobs have access to the values.

The second pair of jobs use "store sales" as well but it also filters by the output of the first (highest-selling stores) and then counts the quantities of the sales by store and item in the reducer. Once the quantities are summed up the data is passed to another job to sort and find the M least selling items in a similar manner described in the previous paragraph.

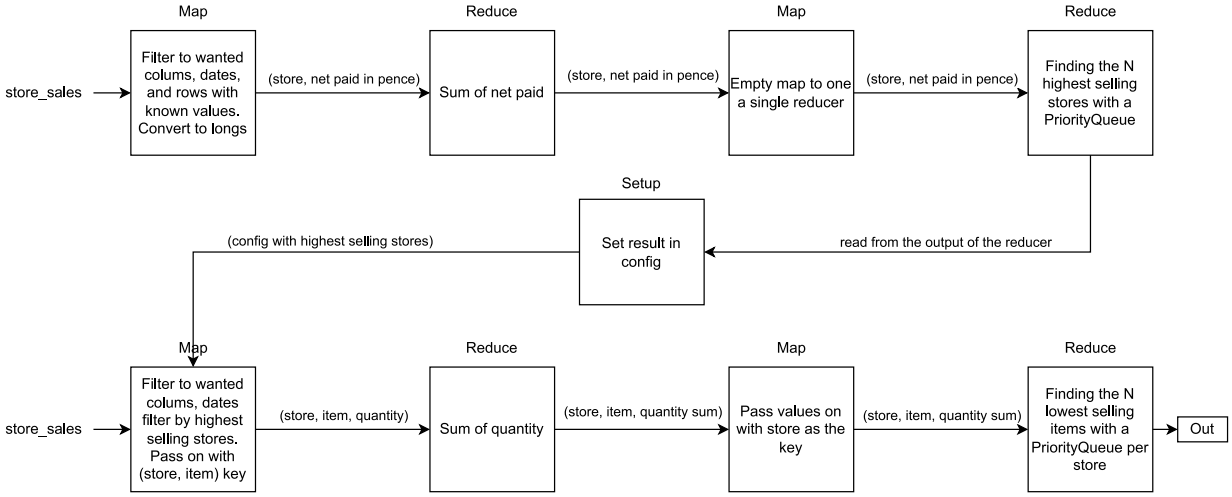


Figure 6: MapReduce architecture for Query 1B

2.7 Query 1C

Query 1C was implemented with two separate MapReduce jobs, using the "store_sales" data as input. The first mapper filtered to wanted columns and valid rows within the date range, and then sent values through a combiner (which created a partial reduction). In the reducer, aggregate values for the total net paid including tax were produced per date in the date range. After this, the results were sorted in another MapReduce job.

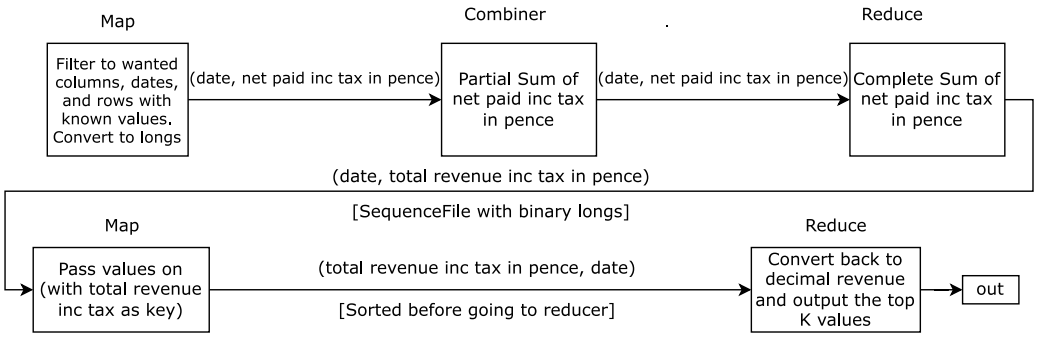


Figure 7: MapReduce architecture for Query 1C

2.8 Query 2

For both of the Join queries, it was important to reduce the amount of data before the join as much as possible, as joins are expensive operations. Because of this, the aggregate revenue for each store was calculated prior to join in both implementations. For the aggregations, combiners were used again to allow for a partial reduction to be done in the mappers to increase performance. For this query, the assumption was made that an inner join was wanted.

2.8.1 Reduce-Side Join

The Reduce side join was completed with aggregation on revenue, followed by a join on the store attribute, followed by sorting and ordering for output. Within the join MapReduce, there were two inputs: aggregated revenue, and the stores table (for floorspace). Because these inputs were structured differently, two separate mappers were used for these two inputs. As both of these mappers needed to go into the same reducer class, their outputs needed to be of the same type. The key (the store) was already of a matching `Text` type, but the value needed to be able to store revenues or floorspace, and needed an indication of which table the value came from. This was achieved with a new Writable class `TableLongWritable`, which contains an `IntWritable` to indicate which table was used, and a `LongWritable` for revenue in pence or floor space. Similarly, a new class was created to store the store and floor space together in the final sorting MapReduce. The overall architecture for this query can be seen in figure 8.

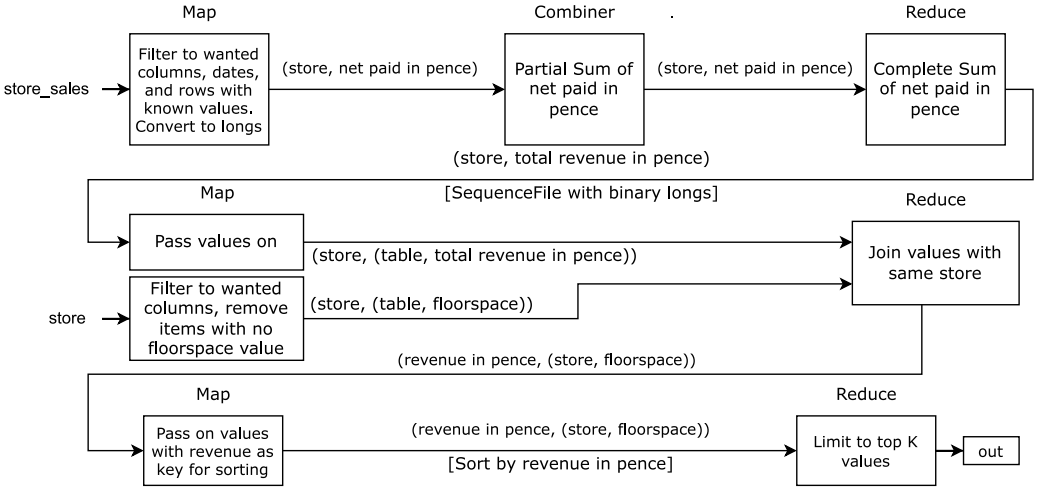


Figure 8: MapReduce architecture for Query 2 with a Reduce-Side Join

2.8.2 Memory Backed Join

The memory-backed join was completed in a slightly different way. Again, the aggregation was done first to reduce the size of the intermediate table used in the join. However, an additional step was used in order to try and increase performance - the number of values was limited to K *before* the join. This was especially difficult as the overall limit for the query is conceptually done after the ordering by overall revenue *and* floorspace, a value that is unknown before the join. Because of this, enough data was kept to be able to do the ordering on the floorspace later, by keeping all values that are associated with the top K *unique* revenues.

The pre-join limited (store, revenue) pairs were loaded into a hashmap after the file was cached in Hadoop for faster access by the mappers. This intermediate table was used over the store table, as it would have fewer values inside it - not all stores had sales in the "store_sales" table, so the aggregate revenues would have fewer items than the number of stores. This meant that the smaller table (aggregate revenues) should be the one to store in memory. After this data was loaded into memory, the mapper went through each line in the "stores" table interpreting the store and floorspace, and queried the HashMap for each store to get the revenue. These values were combined in a key-value pair containing a new `StoreFloorspaceWritable` object as the value, with the revenue being used as the key. Since the revenue was the key, these values could be sorted before going to the reducer. Once in the reducer, and conflicts with duplicate revenues were sorted on floorspace, and the values were limited again by K and output. The overall architecture for this query can be seen in figure 9.

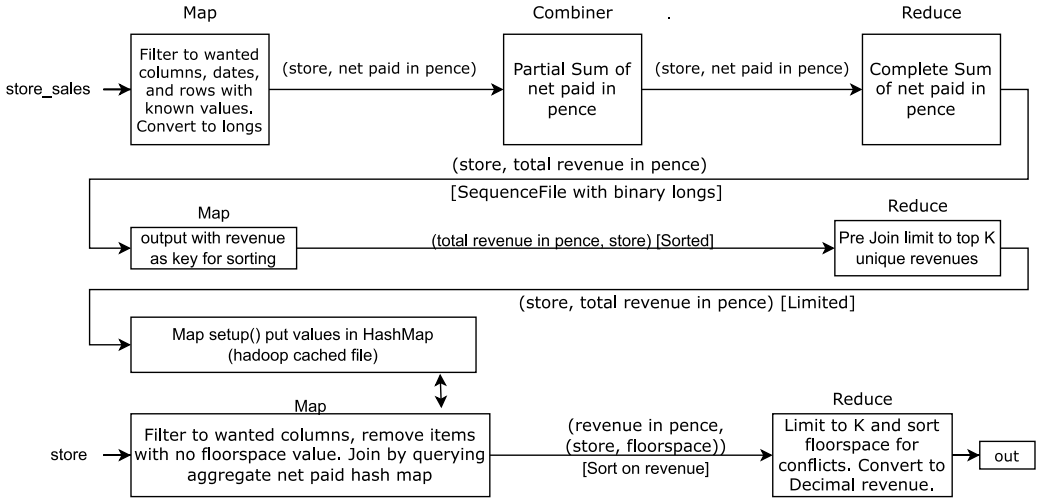


Figure 9: MapReduce architecture for Query 2 with a Memory-Backed Join

3 Exploratory Data Analysis

Explorative data analysis (EDA) as first described by Tukey in 1977 [9] is the process of reviewing the dataset at hand with an "open mind" [3] to gain a heuristic understanding of its main characteristics. For this coursework, we are given two datasets titled "store" and "store_sales". They are part of the TPC-DS[1] benchmark which is an OLAP[2] benchmark meant to model analytical queries run against a "big data" database.

To this end, the TPC-DS benchmark creates a schema that aims to model "the sales and sales returns process for an organization that employs three primary sales channels: stores, catalogs, and the Internet" (page 17) [2]. In total, the schema contains seven tables plus seventeen additional unnormalised tables. Columns follow a specific naming convention. Every column has a prefix which is the abbreviated table name. Some columns have a suffix denoting whether the attribute is a surrogate key (SK) or not. If a column has the SK suffix then it serves as a proxy identifier for that particular tuple.

As explained by Elmasri et al. [4] the query processing pipeline performs "SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes" (page 701). This means that the SELECT and PROJECT operations dictate the dimensionality of our resulting table, i.e. the number of rows and attributes returned for our query. The dimensionality of the query result then dictates the number of disk I/Os performed by the system which translates to a slower or faster running time of the query. Consequently, for our EDA we will only inspect the attributes used in the SELECT and PROJECT operations of our queries. Any other columns are deemed unimportant as they do not affect the runtime of our queries and will thus be ignored in the EDA.

3.1 Store

The store table has 29 attributes describing the store's location, name, number of employees, open hours and a variety of other closely related characteristics. A comprehensive list of its attributes can be found on page 24 of

[1] Transaction Processing Performance Council (TPC) - Decision Support (DS) [2]
[2] Online analytical processing

the TPC-DS documentation [2]. The "s_store_sk" attribute is the primary key (PK) of the table, with the only other attribute relevant to our queries being "s_floor_space". Overall, the store table has 112 records.

3.1.1 s_store_sk

The "s_store_sk" attribute is the PK of the "store" table and is a discrete, quantitative variable. In the schema it is stored as a string. As it is the PK all 112 "s_store_sk" values are unique and non-null. They range from 1 to 112 with a step size of 1.

This means that the imaginary company modeled by the TPC-DS benchmark operates 112 physical stores. Whether all stores are active remains to be seen through analysing the sales for these stores in the "store_sales" table. The "s_store_sk" attribute is one of the attributes that affects the number of records returned in the join of query 2.

3.1.2 s_floor_space

The "s_floor_space" contains 111 integers and 1 missing value, where 19 of the values are duplicates. This attribute is a discrete, quantitative variable stored as an integer in the schema. The square unit associated with the "s_floor_space" attribute is unknown. Statistics about the range (i.e. min and max), interquartile range (IQR) and mean of the attribute can be seen in table 2.

Table 2: Summary statistics for s_floor_space

Min	Max	IQR	Mean	Std.	CV
5,093,780	9,810,608	2,599,490	7,472,353.865	1,401,156.713	0.1875

Based on the standard deviation σ and the sample mean μ we can calculate the coefficient of variation (CV) which is defined as σ/μ. The CV value as explained in [1] is a useful statistic for non-negative values as it is independent of units and given as a ratio. As such, the CV value allows for direct comparison between different attributes even if they have vastly different scales and mean values as it is the case with the dataset at hand. A CV value of 1 or greater means high variablity and can be a starting point for further analysis.

Based on the CV value of 0.1875 we can conclude that the "s_floor_space" attribute has little dispersion around the mean. Additionally, based on the skewness value of -0.0722 we know that the distribution is approximately symmetric.

3.2 Store sales

The store sales table has 23 attributes describing when items were sold, at what quantity, for how much, and at which store. It also includes information about profits and taxes paid as well as some customer information. A comprehensive list of its attributes can be found on page 18 of the documentation [2]. The "ss_item_sk" attribute is one of two attributes determining the composite key (CK) of the store table. However, we will not look at the other attribute (ss_ticket_number) of the CK as it is not called in any of our queries. Another attribute to pay close attention to is "ss_sold_date_sk" as it is used in the SELECT operation of every single query. As such the "ss_sold_date_sk" will determine the number of records returned and hence the running times of the queries. The store sales table has 115,203,420 records in total.

3.2.1 ss_item_sk

The "ss_item_sk" attribute is part of the store sales table's CK. It is also used in a variety of PROJECT operations in our queries. As it is part of the CK it is non-nullable. Out of 115,203,420 records, only 52,000 are unique meaning that only 52,000 different items have been sold, and that there are many duplicates. The attribute is a discrete, quantitative variable and is stored as a string in the TPC-DS schema. Its summary statistics can be found in table 3. Additionally, this attribute is a foreign key (FK) to the items tables and links to its PK, i.e. the "i_item_sk" attribute.

Table 3: Summary statistics for ss_item_sk

Min	Max	IQR	Mean	Std.	CV
1	52,000	25,999.0	25,997.96	15,011.63	0.5774

3.2.2 ss_store_sk

The "ss_store_sk" attribute is also a FK of the "store sales" table which links to the PK of the store table which is "s_store_sk". It is also a discrete, quantitative variable stored as a string in the TPC-DS schema. However, as opposed to the "s_store_sk" attribute, the "ss_store_sk" attribute only has 58 unique values. About 4.5% of 115,203,420 records in the "ss_store_sk" column are also null.

This means that 58 out of 112 stores sold every item ever sold in our records. Additionally, in about 5.1 million cases we cannot identify in which store an item has been sold as the "ss_store_sk" column is null.

Table 4: Summary statistics for ss_store_sk

Min	Max	IQR	Mean	Std.	CV
1	112	57.0	56.33	32.89	0.5838

3.2.3 ss_sold_date_sk

The "ss_sold_date_sk" attribute is another FK of the store sales tables and links to the PK of the date table which is "d_date_sk". It is a discrete, quantitative variable that is stored as a string in the TPC-DS schema.

The TPC-DS documentation states that any dates must between January 1, 1900, and December 31, 2199. Additionally, all dates are stored in the Julian date-time format. This makes comparisons between dates easier, however, it impacts interpretability as they are simply incrementing integers. For example, January 1, 1900, will be stored as '2415021' in the TPC-DS tables.

The summary statistics of the "ss_sold_date_sk" attribute are shown in table 5. Once converted to the Gregorian date-time format and aggregated we know that the "ss_sold_date_sk" attribute holds 1,824 unique records from January 2, 1998, to January 2, 2003. This means that the store sales table represents 5 years' worth of sales data during which 115 million sales have been made.

Table 5: Summary statistics for ss_sold_date_sk

Min	Max	IQR	Mean	Std.	CV
2,450,816	2,452,642	893	2,451,779	526.87	0.0002

Upon closer inspection of the "ss_sold_date_sk" attribute a clear pattern in the distribution of the records can be observed as shown in figure 10. There seems to be a cyclical pattern in the sales dates which persists every year within the 5-year period. From the beginning of January until the beginning of August around 35,000 sales are made per day. As of 5 August around 80,000 sales are made per day. Then as of 5 November around 120,000 sales are made per day until the end of the year. As such, each year has 3 phases. The variance in the number of sales per day per phase is very low resulting in a near-uniform distribution of each phase. Notably, there are also 3 outliers in the dataset. On January 2nd of the years 1999, 2000, and 2002 over 150,000 sales are made in a single day.

The distribution of the "ss_sold_date_sk" attribute is extremely important to consider as it is used in the SELECT operation of every single query. As such the chosen date range in the SELECT operation will drastically change the number of records returned. For instance, if a date range is chosen between 5 February and 7 February (inclusively) then around $3 * 35,000 = 105,000$ will be returned. However, if another 3-day period date range is chosen in December then around $3 * 120,000 = 360,000$ records will be returned.

For the purpose of comparing the scalability of Hive against Hadoop is it paramount that any subset of the original dataset used has the same distribution in the "ss_sold_date_sk" column. More information can be found in the methodology section.

3.2.4 ss_quantity, ss_net_paid and ss_net_paid_inc_tax

The "ss_quantity", "ss_net_paid", and "ss_net_paid_inc_tax" attributes are all closely related. This is as expected as the more items have been sold, the higher the total revenue and thus the higher total tax paid. We expected a positive correlation between the attributes which has been confirmed as shown in figure 11.

The "ss_quantity" attribute records the number of times the same item has been sold during a single item type sale. It is a discrete, quantitative variable saved as an integer in the schema. There are only 101 unique values in the column. Based on the summary statistics shown in table 6 we know there are between 1 and

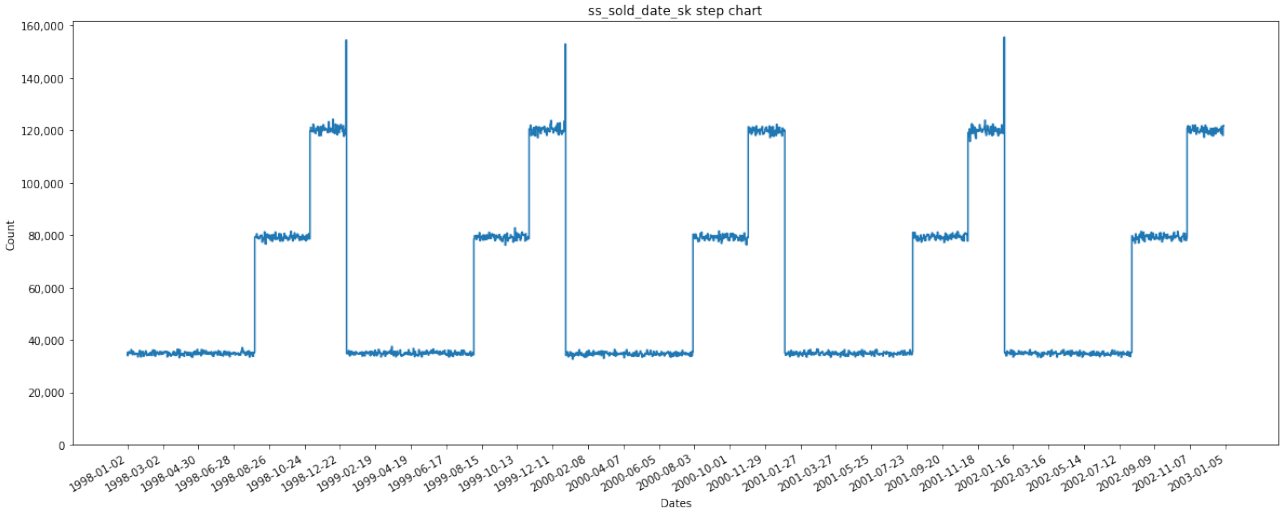


Figure 10: ss_sold_date_sk step chart

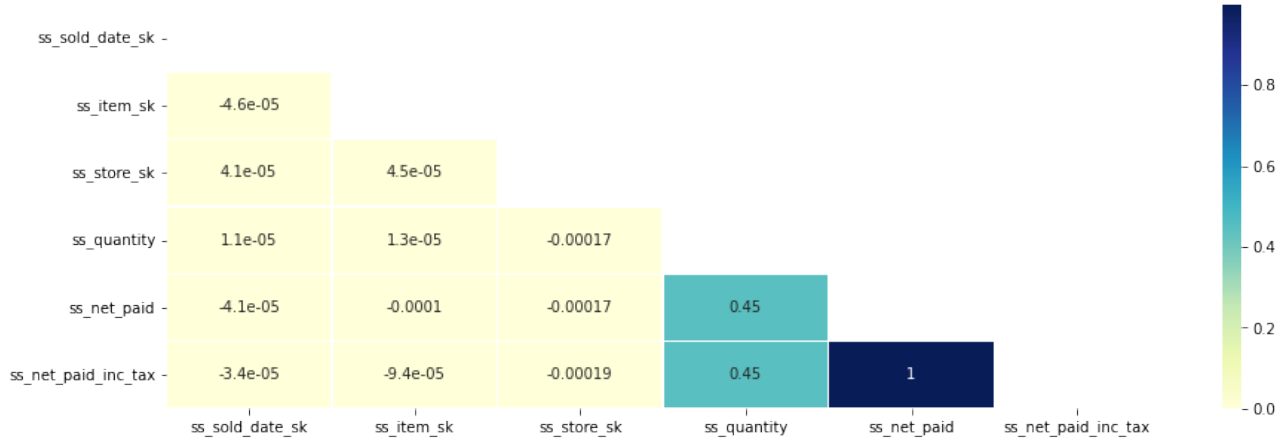


Figure 11: Correlation matrix of relevant store sales attributes

100 items sold at once. Upon further analysis we found that all "ss_quantity" records are nearly uniformly distributed which indicates that this attribute might have been synthetically populated.

Table 6: Summary statistics for ss_quantity

Min	Max	IQR	Mean	Std.	CV
1.0	100.0	49.0	50.5	28.86	0.5716

The "ss_net_paid" attribute records the total amount a customer has paid for one item type (which can include multiple of the same item). It is a discrete, quantitative variable and is stored as a decimal in the schema. Interestingly, the "ss_net_paid" attribute has a very high positive skew of 2.1422. This can be seen in the data where 7,662,532 positive outliers were detected with values greater than $Q3 + 1.5 * IQR$ (as defined by Tukey [8]). This means that the highest-selling items will be outliers.

The "ss_net_paid_inc_tax" attribute behaves almost identically to the "ss_net_paid" attribute. The difference is that the "ss_net_paid_inc_tax" attribute records the tax paid on the total revenue earned by selling one item type. Similarly to "ss_net_paid", the values in the "ss_net_paid_inc_tax" attribute are also positively skewed. The value of the "ss_net_paid_inc_tax" attribute skew is 2.1467 and we can detect 7,668,350 positive outliers. This again means that the items with the highest paid tax value will be outliers.

Table 7: Summary statistics for ss_net_paid

Min	Max	IQR	Mean	Std.	CV
0.00	19,744.00	2,128.08	1,721.1	2,182.28	1.268

Table 8: Summary statistics for ss_net_paid_inc_tax

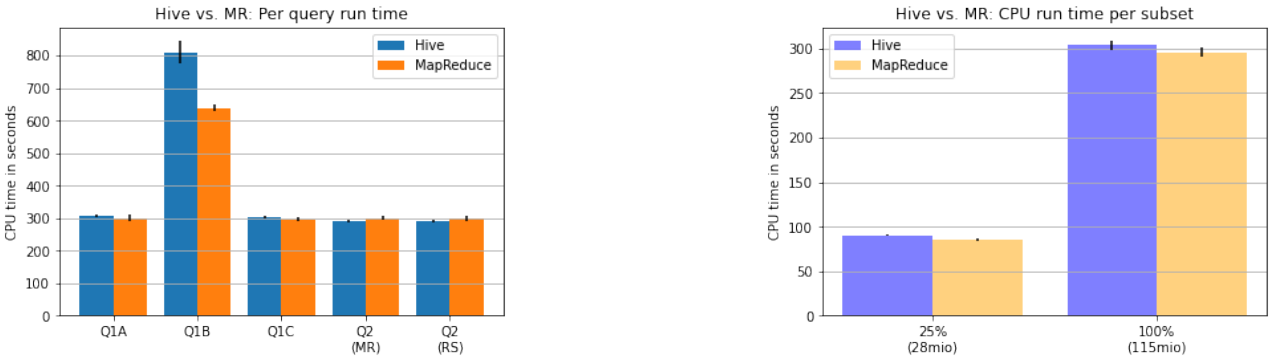
Min	Max	IQR	Mean	Std.	CV
0.0	21,344.38	2,223.21	1,798.54	2,281.9	1.2688

4 Comparison Methodology

In order to compare both approaches to big data queries, we profiled how long our Hive and our MapReduce implementations took to execute. This was done in the same way for the Hive and the MapReduce-based implementations, where the MapReduce sub-jobs were timed individually and these results were summed to work out the overall time. The wait times between jobs were ignored to avoid additional time being added due to hadoop jobs being queued and waiting to start. This was based on testing where we found that queueing multiple jobs at once simply led to a lot of jobs waiting while one executed, and we assumed that other groups using the hadoop cluster could also be adding wait times, so it was decided to ignore this external variable. On top of this, Hive also outputs a cumulative CPU time across nodes, but this was ignored, where we instead used the timestamps between the start and end of mapreduce jobs to avoid some of the time being counted multile times. In java, the `getStartTime` and `getFinishTime` job functions were used, where we also made sure that the time wasn't being combined from each node. These measures ensured that the times calculated from Hive and our MapReduce implementations were comparable.

When comparing the times between Hive and the MapReduce implementation, results were averaged over three runs and the sample variance was calculated. These tests also used sales from the entire date range. This was done to reduce the impact of MapReduce overheads on our results, and to be as representitive of the dataset as possible.

Queries were also done on specific date ranges to see how much the date range affected results. This was especially important with the distribution of how many items were sold for different date ranges, as the number of rows could change drastically based on this, as seen in figure 10. To this end we used query 1 C and testing for the same date range (56 days), but with 35, 80 and 120 thousands sales a day resulting in 1.9, 4,5 and 6.7 million records being aggregated. Strangely, changing this made little difference to the runtime possibly due to the initial overhead of map reduce. However, as shown in figure 12b the total number of records in the file does impact the overall running time.



(a) Query running times

(b) Dataset scaling (Q1C)

Figure 12: Hive versus MapReduce profiling

Additionally, we also compared the running times of all queries in Hive versus MapReduce as shown in figure 12a. As we can see query 1 B was faster using the MapReduce framework. This might have something to do with the fact that query 1 B is a "non-standard" SQL query due to the partitioning and number of subqueries. As such, subqueries may result in a potentially non-optimised Hive implementations of MapReduce.

The number of reducers used within our MapReduce-implemented queries led to some interesting results.

The default MapReduce implementation uses a single reducer for all jobs, but when this was changed to using more reducers the overall runtime either stayed the same or increased. This is assumed to be due to the dataset not being large enough for more reducers to be beneficial, where the majority of the time is being spent on other MapReduce overheads rather than issues in reducers being overwhelmed. This was attempted for aggregations and sorting, but neither showed any advantage to having various numbers of reducers, up to 50. Because the MapReduce implementation was using speculative execution to start reducers before all the mappers were finished, the load on the reducers may have been more spread out as well.

The limiting by K was also tested to see if it had an impact on performance in query C, but this showed no significant performance differences, likely as the same results were still being found, just not printed out at the end.

The effect of combiners was also tested, as queries 1A, 1C and 2 all aggregated revenues. This found that the combiners led to a 13% reduction in execution time, from 327 seconds to 285 seconds. This is as expected, as combiners will allow for results to be partially calculated within the mapper outputs, allowing for far fewer key-value pairs to be sent to the reducers, a likely bottleneck within the MapReduce framework.

The two joining methods also took a very similar amount of time, despite the memory-backed join being faster in theory. This was understandable for this dataset size, as the number of values being joined was quite small. Because of this, the runtime for the join was small with the majority of the time for the join queries being spent on the initial aggregation of "ss_net_paid" rather than the join itself.

5 Conclusion

Creating the queries with two different approaches highlighted the very large difference in implementation challenge for Hive and manually creating MapReduce jobs. Creation of jobs with Hive was very simple and easy due to it's declarative nature, and transferrable experience of SQL in the past. Conversely, the MapReduce implementation was challenging, especially being able to ensure correctness of results. In sections of the MapReduce implementation, we could be confident in the validity or results due to creating a Hive implementation first, and this made the issues with floating point arithmetic evident. For creating further and more complex queries, this would be vital for ensuring that a custom MapReduce was working as expected.

As seen in the results, the performance improvement from a manual MapReduce implementation was very minor, and actually marginally worse for the joins, where the aggregation sections took the majority of the time for most of the jobs. The only case where MapReduce might be appropriate is for nested or "non-standard" queries such as 1 B which uses more niche SQL features like partitions and is quite complex.

Considering that a sensible approach to any MapReduce implementation would involve creating a Hive query first to ensure correctness, our results show that the additonal work for manual MapReduce is unlikely to be worth the additional work for very minor performance improvements for most cases. The only exception could be "non-standard" queries meant to be run repeatedly as compiled queries.

6 Bibliography

References

[1] Hervé Abdi. "Coefficient of variation". In: *Encyclopedia of research design* 1.5 (2010).

[2] Transaction Processing Performance Council. *TPC BENCHMARK ™ DS Standard Specification Version 2.10.0*. Sept. 2018. URL: `https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.10.0.pdf` (visited on 03/10/2023).

[3] Yadolah Dodge. "Exploratory Data Analysis". In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 192–194. ISBN: 978-0-387-32833-1. DOI: `10.1007/978-0-387-32833-1_136`.

[4] R Elmasri et al. *Fundamentals of Database Systems*. 7th ed. Springer, 2015. ISBN: 978-0133970777.

[5] The Apache Software Foundation. *Apache Hive*. URL: `https://hive.apache.org/` (visited on 03/10/2023).

[6] IBM. *Apache MapReduce*. URL: `https://www.ibm.com/topics/mapreduce` (visited on 03/25/2023).

[7] Oracle. *BigDecimal (Java Platform SE 7)*. URL: `https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html` (visited on 03/25/2023).

[8] Georgy Shevlyakov et al. "Robust versions of the Tukey boxplot with their application to detection of outliers". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE. 2013, pp. 6506–6510.

[9] John W Tukey et al. *Exploratory data analysis*. Vol. 2. Reading, MA, 1977.