

Hadoop vs Hive

Neema Raiyat

September 24, 2023

Contents

1	Exploratory Data Analysis (EDA)	1
1.1	store_sales table	1
1.2	store table	3
2	Explanation of Solutions	3
2.1	Hadoop	3
2.1.1	Use of EDA	3
2.1.2	Query 1A	4
2.1.3	Query 1B	4
2.1.4	Query 1C	5
2.1.5	Query 2: Reduce-Side Join	6
2.1.6	Query 2: Memory-Backed Join	6
2.2	Hive	7
2.2.1	Query 1A	7
2.2.2	Query 1B	7
2.2.3	Query 1C	8
2.2.4	Query 2: Joins	8
3	Performance Analysis: Hadoop vs Hive	8
3.1	Results	9
3.2	Speed	10
4	Conclusion	11
	References	11

1 Exploratory Data Analysis (EDA)

Python was used for our exploratory data analysis (EDA) as it comes with many good data visualization libraries that could help find patterns in the dataset. Testing began on the 1G dataset, and once all data analysis was done, the same code was ran on the 40G dataset.

1.1 store_sales table

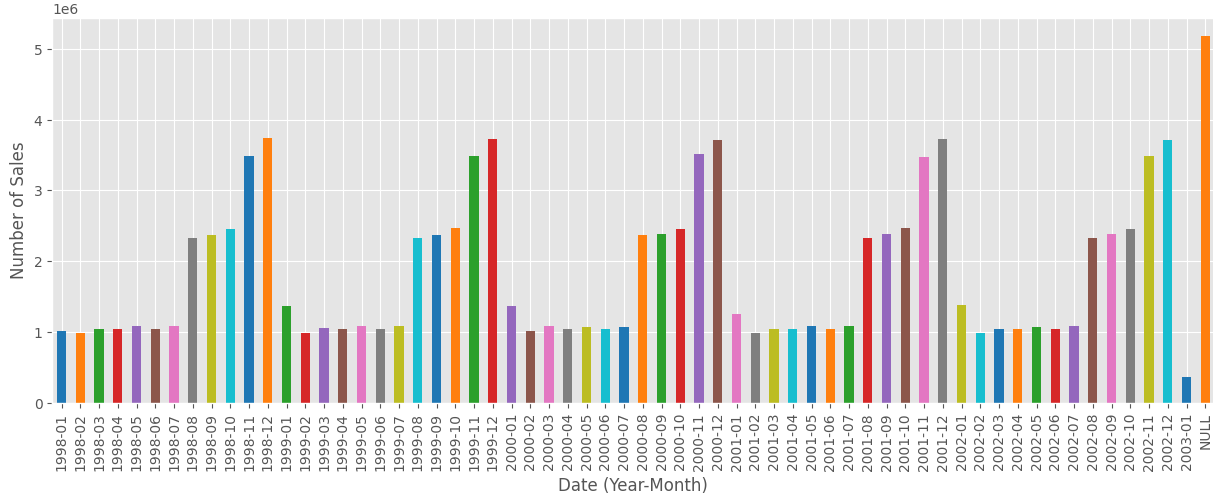


Figure 1: Number of sales for each month

Figure 1 shows how the number of sales varies in the 1998-2003 time period. Each year you can see that the later months have the most sales. Specifically, August to December have far more sales than the first 6 months of the year (January to June). Most of the sales in the second half of the year occur in November and December, with the most number of sales being the December of each year, most likely due to it being the Christmas holiday period. After the year has ended, the January of the following year has slightly more sales than the rest of the first of that year (February to June). Furthermore, Figure 1 shows that there are more than 5 million sales with the date set to null (see rightmost bar).

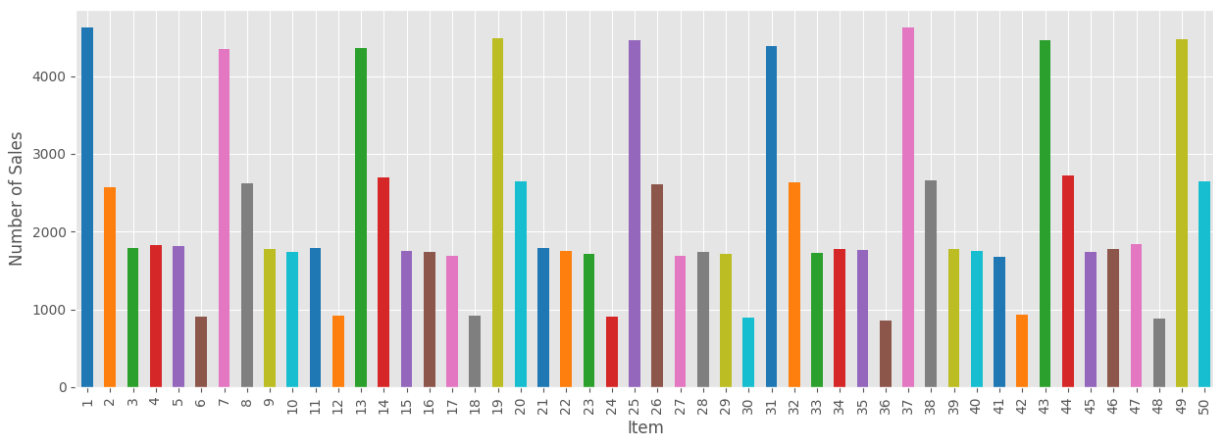


Figure 2: Number of sales for the first 50 items

Figure 2 shows the number of sales of the first 50 items (`ss_item_sk`). There is clear pattern that extends to all 18,000 items in the dataset (but is not shown due to the graph being very cluttered since there would be 18,000 labels on the x-axis). The pattern repeats every 6 items, the first item sells the most, the second item sells the second most (approximately half the number of sales as the first), then

the next 3 items sell the third most, each selling approximately the same amount. Finally the sixth item sells the least and the pattern then repeats.

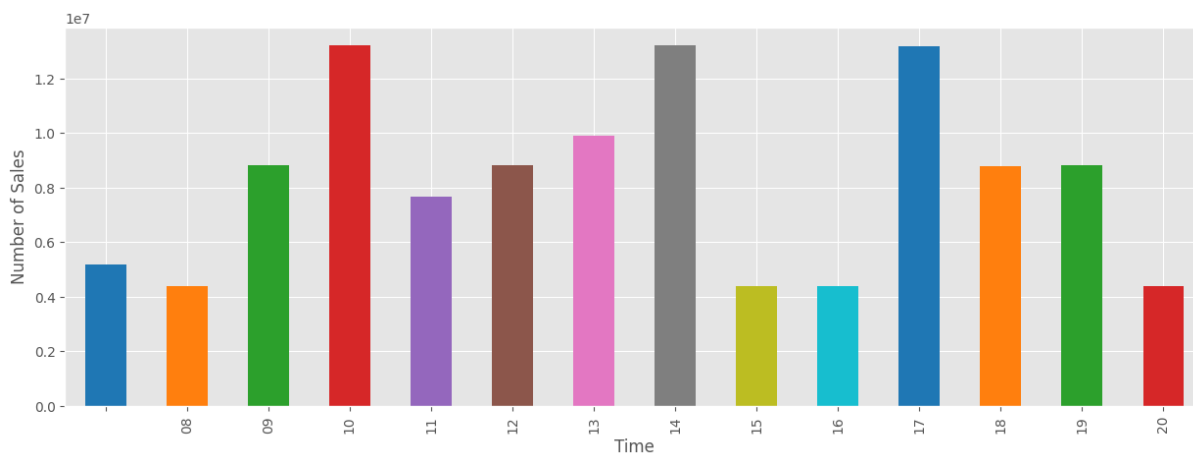


Figure 3: Number of sales in each hour of the day

Figure 3 shows what the most popular times of day are, most popular meaning the most sales. You can see that (24-hour clock) the time ranges 10:00-11:00, 14:00-15:00 and 17:00-18:00 are the most popular times, each range having approximately the same number of sales. The least popular sales are in the time ranges 08:00-09:00, 15:00-17:00 and 20:00-21:00.

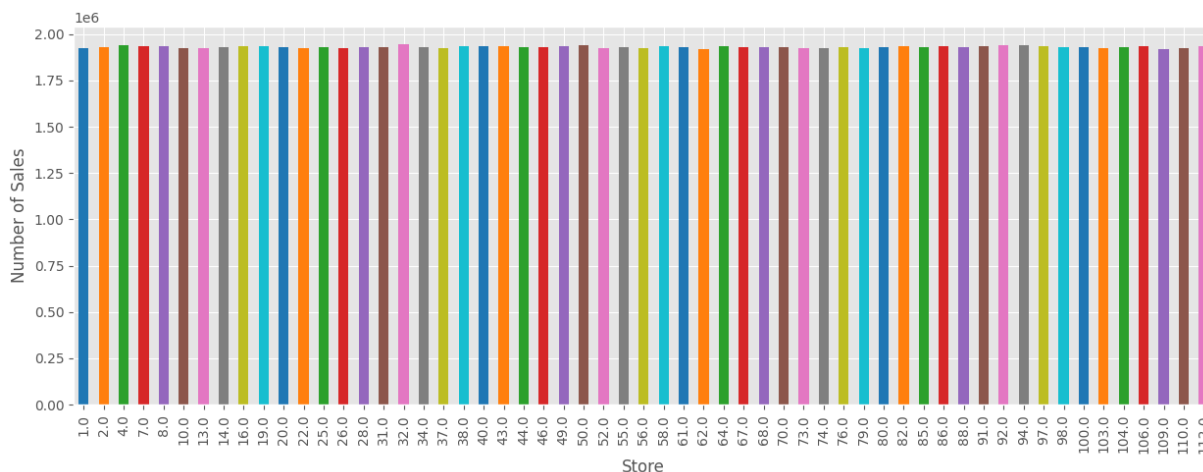


Figure 4: Number of sales for each store

Figure 4 shows that each store had approximately the same number of sales.

Attribute	Type / Format	Minimum Value	Maximum Value
ss_sold_date_sk	integer (julian calendar)	1998-01-02	2003-01-02
ss_item_sk	integer (primary key)	1	18,000
ss_store_sk	integer (f.k. to s_store_sk)	1	112
ss_quantity	integer	1	100
ss_net_paid	decimal (7 digits, 2 decimal places)	0.00	19744.00
ss_net_paid_inc_tax	decimal (7 digits, 2 decimal places)	0.00	21344.38

Table 1: Types, maximum and minimum values of important attributes in store_sales table

1.2 store table

Attribute	Type / Format	Minimum Value	Maximum Value
s_store_sk	integer	1	112
s_floor_space	integer	5093780.0	9810608.0

Table 2: Types, maximum and minimum values of important attributes in store table

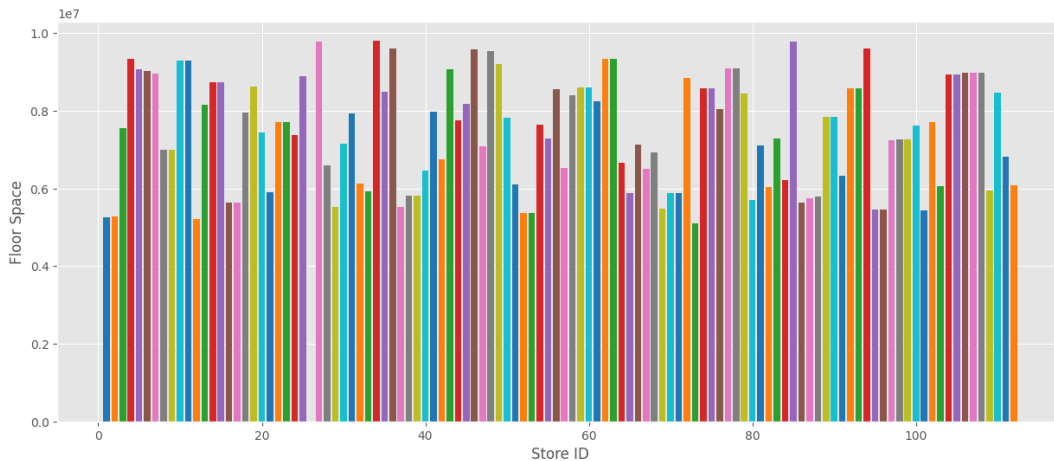


Figure 5: Floorspace for each store

Figure 5 shows a random distribution and hence no discernable relationship between the store and its floorspace.

2 Explanation of Solutions

2.1 Hadoop

2.1.1 Use of EDA

The exploratory data analysis informed various design decisions. For example, the EDA revealed the maximum and minimum values of each attribute in both the `store_sales` and `store` table which proved useful when deciding what data types to assign in Hadoop (Java). Please note that below, we refer to sending multiple types of data in ‘tuples’. In the actual code, this was done using string concatenation,

however it is semantically easier to think of them as tuples and hence the solutions will be explained as if tuples were used.

The EDA also showed that each store had approximately the same number of total sales. This information was useful as it indicated that there would be no bottlenecks if the store is used as the key when passing tuples from the mapper to the reducer, since each reducer would receive the approximately the same number of data points. However, the EDA revealed that there was a pattern with the number of total sales for each item. Although this information was not used for our queries, it could have perhaps been used for Query 1B. For example items with a lower amount of sales could be passed to the same reducer and items with a larger amount of sales are passed to unique reducers. This would ensure that the computation is more evenly spread and reduce bottle-necking, thus improving efficiency. Additionally the EDA showed that there were only 112 stores in the `store` table, which was useful when implementing the memory-backed join as it showed that `store` is a small file that can be linearly iterated over very quickly.

2.1.2 Query 1A

Query 1A is obtained by using two cascading MapReduce jobs, i.e. the output of the first MapReduce job is the input of the second. The first MapReduce job uses the mapper `NetSumMapper` and the reducer `NetSumReducer`. `NetSumMapper` partitions the data according to store ID (`ss_store_sk`) so that each reducer is responsible for a particular store, of which there are 112, and hence 112 reducers. The key emitted by the mapper is the store ID (`ss_store_sk`) and the value emitted is the net paid (`ss_net_paid`). Note, this is only emitted if the sold date of the item (`ss_sold_date_sk`) falls within the specified time period. This check is done in the mapper as opposed to the reducer to prevent the creation of excess reducers that are processing data we don't need and hence reduce efficiency.

Each reducer will receive the list of net paid values for the store its responsible for and then sum them in order to find the total revenue in the specified time period generated by that store.

The second MapReduce job, defined by `LimitMapper` and `LimitReducer`, is concerned with sorting and limiting the output of the previous job. The mapper `LimitMapper` receives as input, a tuple where the key is a store ID (`ss_store_sk`), and the value is the revenue. Since we are looking to find the stores with the least revenue, they must all be sorted in the same place so sorting cannot be parallelized. Therefore, the mappers simply emit all the data they receive to a single reducer. This is done by emitting all the data with the same key, in this case, set to '1'. The value is set to the data received (store ID, revenue).

The stand-alone reducer receives all the stores with their corresponding revenue and then sorts them according to revenue in ascending order, and outputs the top k stores along with their revenue.

2.1.3 Query 1B

Query 1B is a more involved query, which has three cascading MapReduce jobs. The purpose of the first MapReduce job is to find the total number of sales for a particular item in a particular store in the given time period. The second MapReduce uses the output from the first to calculate the total number of sales for each store. This is found by summing the total number of sales of each particular item in that store. The M lowest selling items of each store are found by sorting the list of items. The third MapReduce job sorts the output of the previous by total store sales in descending order. This finds the top N highest selling stores. For each one of these N stores, and each M sorted items in them (sorted by number of sales in descending order), output the store ID, the item ID and the number of sales of that item. More detail below.

The first MapReduce job uses the mapper `QuantitySumMapper` and reducer `QuantitySumReducer`. `QuantitySumMapper` partitions the data according to a combination of the item ID (`ss_item_sk`) and store ID (`ss_store_sk`). The key emitted is therefore a tuple: containing: item ID, store ID. The

corresponding value is a tuple containing the store ID and the quantity of that item sold. Below is a representation of what the `QuantitySumMapper` emits:

Key: (item ID, store ID)	Value: (store ID, quantity)
--------------------------	-----------------------------

A reducer is therefore defined for each item in each store. Each reducer will receive a list of quantities for a particular item in a particular store the reducer is responsible for. These quantities are summed to find the total number of sales for that item (`quantity sum`). The reducer then outputs this sum:

Key: (item ID, store ID)	Value: (store ID, quantity sum)
--------------------------	---------------------------------

The second MapReduce job uses the mapper `StoreQuantityMapper` and reducer `StoreQuantityReducer`. `StoreQuantityMapper` partitions the output of the previous MapReduce job by store ID. The value is a tuple containing the number of sales for an item as well as that item's ID. The following is emitted:

Key: (store ID)	Value: (quantity sum, item ID)
-----------------	--------------------------------

A reducer `StoreQuantityReducer` is defined for each store ID. It receives a list of tuples which contain the total number of items sold for a particular item, i.e. (`quantity sum`, item ID). This list of tuples is sorted by `quantity sum`. In order to find the total number of sales for that store (so that we can order by highest selling stores), we sum each `quantity sum` as this is essentially the sum of all the sales of each item in that store, which is equal to the total number of sales (`total store sales`). Now that we have a sorted list of the total number of items sold for a particular item, we can find the M least selling items. The reducer then outputs the total number of sales of the store its responsible for, as well as the sorted list (of length M) of tuples containing the total number of sales of each item (Key and Value have been abbreviated to K and V respectively):

K: (store ID)	V: (total store sales, [(quantity sum, item ID), ...])
---------------	--

The third MapReduce job uses the mapper `OrderLimitMapper` and the reducer `OrderLimitReducer`. `OrderLimitMapper` simply sends all the outputs from the previous MapReduce run to the same reducer by setting the key for all emitted values to '1'. Mapper emits:

K: 1	V: (store ID, total store sales, [(quantity sum, item ID), ...])
------	--

The stand-alone reducer `OrderLimitReducer` receives all the data emitted by all the mappers. This is sorted by `total store sales` in descending order. This allows us to get the data of the top N highest selling stores. For each store in this top N, and for each of the M items in this store (which is stored in the list of sorted tuples), the store ID, the item ID and the total number of sales for that item is outputted by the reducer.

2.1.4 Query 1C

Query 1C adopts a very similar solution to query 1A; it uses two cascading MapReduce jobs, one responsible for filtering sales within a specified time period and summing them according to their date of sale, the other job responsible for sorting and limiting.

The first MapReduce job uses the mapper `NetPaidTaxMapper` and the reducer `NetPaidTaxReducer`. `NetPaidTaxMapper` partitions the data according to the date (`ss_sold_date_sk`) of sale so that each reducer is responsible for a particular day in the specified time period. The number of days in the time period specified will equal the number of reducers used. The key is the date (`ss_sold_date_sk`), and the value is the net paid including tax (`ss_net_paid_inc_tax`). This is only emitted if `ss_sold_date_sk` falls within the specified time period.

Each reducer will receive the list of net paid including tax values for the date its responsible for and then sum them. The output of the reducer is the date set as the key, and the sum as the value.

The second MapReduce job, defined by `LimitMapper` and `LimitReducer`, is concerned with sorting and limiting the output of the previous job. The mapper `LimitMapper` receives as input, a tuple where the key is a date (`ss_sold_date_sk`), and the value is the sum of the `ss_net_paid_inc_tax` on that date. Each mapper emits these tuples to the same reducer, by setting the key to '1', and the value being the tuple (date, sum of net paid including tax).

The stand-alone reducer receives all the dates with their corresponding total net paid including tax and then sorts them according to this total in descending order, and outputs the top k days along with their total `ss_net_paid_inc_tax`.

2.1.5 Query 2: Reduce-Side Join

For the reduce-side join to find the floor spaces of the K highest net paid stores, two mappers are used in the first MapReduce job. The first mapper is assigned to the `store_sales` table and goes through each record to find the net pay and store IDs of each sale. After filtering out unneeded records, (i.e. records with null values or that fall outside the given date range) the mapper uses the store ID as the key for the reducers, and the value as a tuple containing the string "ss" and the net pay. The string "ss" is used so that the single reducer can identify that the data has been gathered from the `store_sales` table. Similarly, the second mapper sets the key as the store ID and the value to a tuple containing the string "s" and the floor space.

There are now separate reducers for each store ID which take in data from the two mappers. By iterating through the data given to each reducer, the net pay is summed if it has the "ss" identifier or the floor space is set if it has the "s" identifier. Note the floor space will only be set once as the each record of the store table has unique store IDs. This data is then written to an intermediate file so it can be used by another MapReduce job.

Similar to solutions before, the second MapReduce job uses the intermediate file as the input and the mappers pass the required data to a single reducer by using the same key of "1" for each line of data. The reducer then sorts the results and limits it using the given value of K.

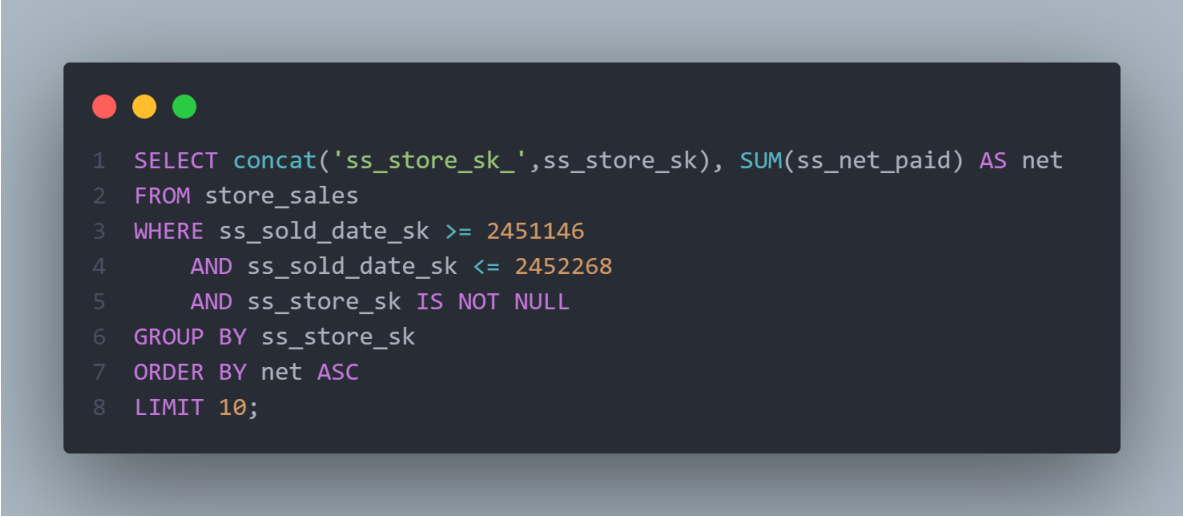
2.1.6 Query 2: Memory-Backed Join

For the memory-backed join, two cascading MapReduce jobs are used. The mapper of the first job extracts the store ID and net paid values from the `store_sales` table and partitions the data according to store ID. This is then passed to a reducer which runs linearly through the cached `store` table to extract the corresponding floor space of the current store. This step is performed in the reducer as opposed to the mapper, to prevent it being done for every line in the `store_sales` file, which would have been inefficient. The reducer then sums the net paid values and passes the store ID, floor space and the sum of net paid values of the current store to the next job.

The second MapReduce job deals with the sorting and limiting of the data extracted in the first job. The mapper emits the data with a key of '1', so that all the data is passed to the same reducer. The reducer then sorts the data according to net paid and floor space and outputs the top k values.

2.2 Hive

2.2.1 Query 1A



```

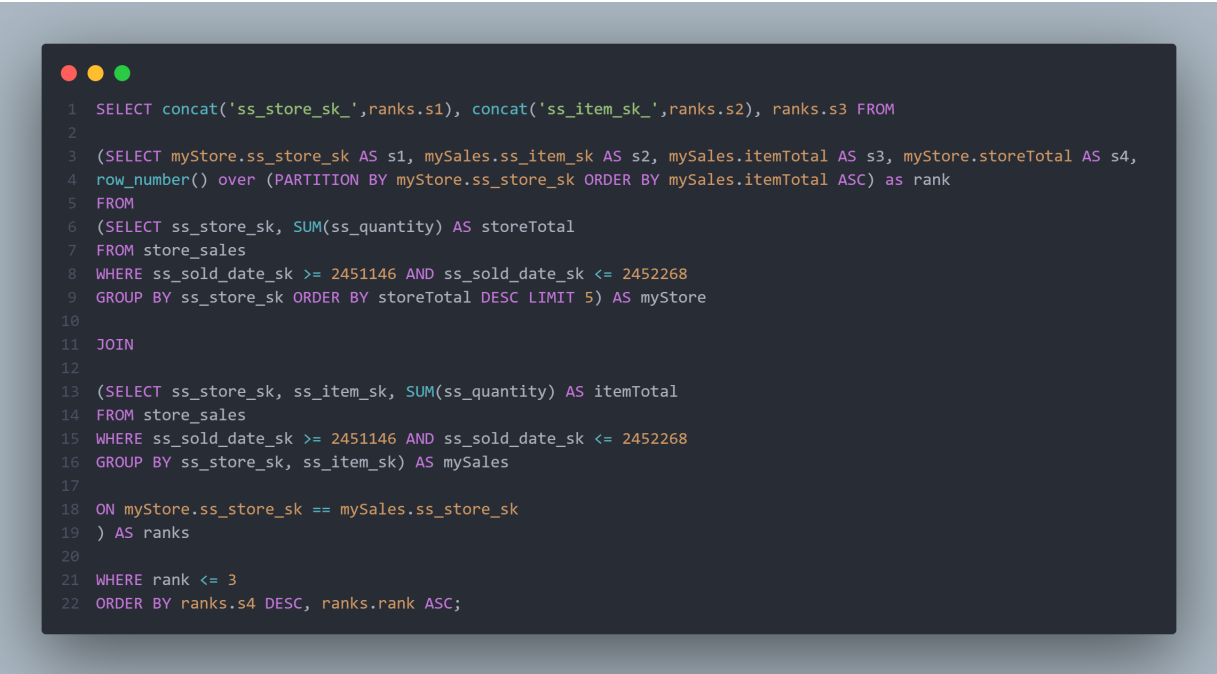
1 SELECT concat('ss_store_sk_',ss_store_sk), SUM(ss_net_paid) AS net
2 FROM store_sales
3 WHERE ss_sold_date_sk >= 2451146
4       AND ss_sold_date_sk <= 2452268
5       AND ss_store_sk IS NOT NULL
6 GROUP BY ss_store_sk
7 ORDER BY net ASC
8 LIMIT 10;

```

Figure 6: HiveQL code snippet for query 1A

The query groups each unique store by **ss_store_sk** and sums all of the revenue for each item sold by that store in the given time period. This is then ordered in ascending order and limited by k.

2.2.2 Query 1B



```

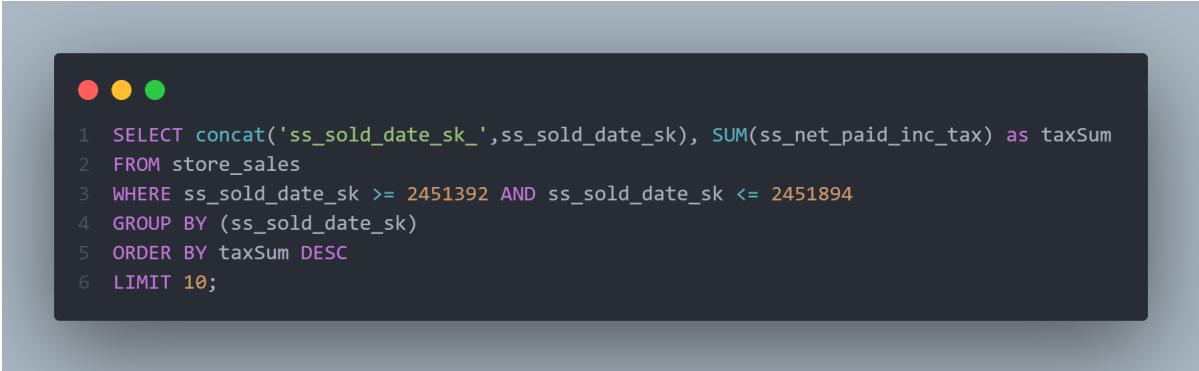
1 SELECT concat('ss_store_sk_',ranks.s1), concat('ss_item_sk_',ranks.s2), ranks.s3 FROM
2
3 (SELECT myStore.ss_store_sk AS s1, mySales.ss_item_sk AS s2, mySales.itemTotal AS s3, myStore.storeTotal AS s4,
4 row_number() over (PARTITION BY myStore.ss_store_sk ORDER BY mySales.itemTotal ASC) as rank
5 FROM
6 (SELECT ss_store_sk, SUM(ss_quantity) AS storeTotal
7 FROM store_sales
8 WHERE ss_sold_date_sk >= 2451146 AND ss_sold_date_sk <= 2452268
9 GROUP BY ss_store_sk ORDER BY storeTotal DESC LIMIT 5) AS myStore
10
11 JOIN
12
13 (SELECT ss_store_sk, ss_item_sk, SUM(ss_quantity) AS itemTotal
14 FROM store_sales
15 WHERE ss_sold_date_sk >= 2451146 AND ss_sold_date_sk <= 2452268
16 GROUP BY ss_store_sk, ss_item_sk) AS mySales
17
18 ON myStore.ss_store_sk == mySales.ss_store_sk
19 ) AS ranks
20
21 WHERE rank <= 3
22 ORDER BY ranks.s4 DESC, ranks.rank ASC;

```

Figure 7: HiveQL code snippet for query 1B

The top N stores are found using the sum of the sale quantities for each item of each store. This is joined to another query to find all the quantities of items sold by these top N stores. Using the **row_number()** function the records are ranked so that the items with the least number of items sold have the smallest rank for each store. The rank is then limited by M so only the M fewest items sold are selected.

2.2.3 Query 1C

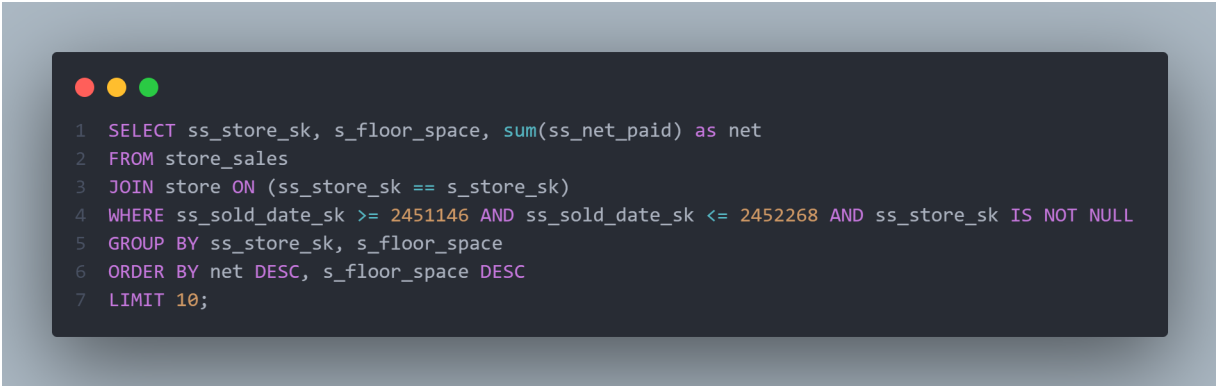
A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays a HiveQL query with line numbers 1 through 6. The query selects the concatenation of 'ss_sold_date_sk_' and 'ss_sold_date_sk' along with the sum of 'ss_net_paid_inc_tax' as 'taxSum' from the 'store_sales' table. It filters for dates between 2451392 and 2451894, groups by 'ss_sold_date_sk', orders by 'taxSum' in descending order, and limits the results to 10 rows.

```
1 SELECT concat('ss_sold_date_sk_',ss_sold_date_sk), SUM(ss_net_paid_inc_tax) as taxSum
2 FROM store_sales
3 WHERE ss_sold_date_sk >= 2451392 AND ss_sold_date_sk <= 2451894
4 GROUP BY (ss_sold_date_sk)
5 ORDER BY taxSum DESC
6 LIMIT 10;
```

Figure 8: HiveQL code snippet for query 1C

The query groups each date by `ss_sold_date_sk` and sums all of the net paid including tax (`ss_net_paid_inc_tax`) for each day in the given time period. This is then ordered in descending order and limited by `k`.

2.2.4 Query 2: Joins

A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays a HiveQL query with line numbers 1 through 7. The query selects 'ss_store_sk', 's_floor_space', and the sum of 'ss_net_paid' as 'net' from 'store_sales'. It joins with the 'store' table on 'ss_store_sk == s_store_sk'. It filters for dates between 2451146 and 2452268 and ensures 'ss_store_sk' is not null. It groups by 'ss_store_sk' and 's_floor_space', orders by 'net' in descending order and 's_floor_space' in descending order, and limits the results to 10 rows.

```
1 SELECT ss_store_sk, s_floor_space, sum(ss_net_paid) as net
2 FROM store_sales
3 JOIN store ON (ss_store_sk == s_store_sk)
4 WHERE ss_sold_date_sk >= 2451146 AND ss_sold_date_sk <= 2452268 AND ss_store_sk IS NOT NULL
5 GROUP BY ss_store_sk, s_floor_space
6 ORDER BY net DESC, s_floor_space DESC
7 LIMIT 10;
```

Figure 9: HiveQL code snippet for query 2

The query groups each unique store by the `ss_store_sk` and sums all of the revenue for each item sold by that store in the given time period. This is then ordered in ascending order and limited by `k`. The query sums the net paid of items for each store within the date range from `store_sales`. This is then joined with `store` to get the floor space associated with each store. We then limit the results by `k`.

3 Performance Analysis: Hadoop vs Hive

This section analyses the performance for each query within Hive and Hadoop. It focuses on comparing results as well as computing speeds.

3.1 Results

Store	Hive	Hadoop
ss_store_sk_109	2000891451.65	2000891451.65
ss_store_sk_62	2001356972.77	2001356972.77
ss_store_sk_73	2002496753.42	2002496753.42
ss_store_sk_13	2005044284.81	2005044284.81
ss_store_sk_26	2006646910.95	2006646910.95
ss_store_sk_37	2007411333.56	2007411333.56
ss_store_sk_56	2007617878.05	2007617878.05
ss_store_sk_110	2007840718.64	2007840718.64
ss_store_sk_76	2009872832.22	2009872832.22
ss_store_sk_79	2010489119.70	2010489119.70

Table 3: Hive and Hadoop output for query 1A with input - start: 2451146, end: 2452268, K: 10

Store	Item	Hive	Hadoop
ss_store_sk_92	ss_item_sk_3309	46	46
ss_store_sk_92	ss_item_sk_10233	80	80
ss_store_sk_92	ss_item_sk_7587	87	87
ss_store_sk_50	ss_item_sk_48207	62	62
ss_store_sk_50	ss_item_sk_18105	133	133
ss_store_sk_50	ss_item_sk_21729	139	139
ss_store_sk_4	ss_item_sk_20811	73	73
ss_store_sk_4	ss_item_sk_39837	97	97
ss_store_sk_4	ss_item_sk_12825	103	103
ss_store_sk_32	ss_item_sk_33081	68	68
ss_store_sk_32	ss_item_sk_963	99	99
ss_store_sk_32	ss_item_sk_30112	114	114
ss_store_sk_8	ss_item_sk_5841	41	41
ss_store_sk_8	ss_item_sk_14067	83	83
ss_store_sk_8	ss_item_sk_15393	83	83

Table 4: Hive and Hadoop output for query 1B with input - start: 2451146, end: 2452268, M: 3, N: 5

Store	Hive	Hadoop
ss_sold_date_sk_2451546	269011207.69	269011207.69
ss_sold_date_sk_2451522	218299535.32	218299535.32
ss_sold_date_sk_2451544	216896013.28	216896013.27
ss_sold_date_sk_2451537	215618328.78	215618328.77
ss_sold_date_sk_2451521	215392369.46	215392369.45
ss_sold_date_sk_2451533	214772795.73	214772795.73
ss_sold_date_sk_2451532	214618693.96	214618693.95
ss_sold_date_sk_2451851	213953384.07	213953384.07
ss_sold_date_sk_2451891	213918317.29	213918317.29
ss_sold_date_sk_2451880	213849908.61	213849908.61

Table 5: Hive and Hadoop output for query 1C with input - start: 2451392, end: 2451894, K: 10

Store	Net Paid	Floorspace
92	8573853	2039983202.07
50	7825489	2036624698.89
8	6995995	2032733764.65
32	6131757	2028710419.82
4	9341467	2028145948.34
94	9599785	2026137085.11
38	5813235	2024183575.31
82	6035829	2023916557.05
97	7232715	2023659816.02
112	6081074	2023012491.83

Table 6: Hive output for query 2 with input - start: 2451146, end: 2452268, K: 10

Memory-backed Join			Reduce-side Join		
Store	Net Paid	Floorspace	Store	Net Paid	Floorspace
92	8573853	2039983202.07	92	8573853	2039983202.07
50	7825489	2036624698.89	50	7825489	2036624698.89
8	6995995	2032733764.65	8	6995995	2032733764.65
32	6131757	2028710419.82	32	6131757	2028710419.82
4	9341467	2028145948.34	4	9341467	2028145948.34
94	9599785	2026137085.11	94	9599785	2026137085.11
38	5813235	2024183575.31	38	5813235	2024183575.31
82	6035829	2023916557.05	82	6035829	2023916557.05
97	7232715	2023659816.02	97	7232715	2023659816.02
112	6081074	2023012491.83	112	6081074	2023012491.83

Table 7: Memory-backed and Reduce-side join output for query 2 with input - start: 2451146, end: 2452268, K: 10

3.2 Speed

The speed of the approaches for each query was investigated by varying the number of days within the date range of the queries. The effect of changing the value of k was also investigated, but it was found that increasing k had little effect on the speed of the queries. This makes sense as using k to limit the number of outputs is computationally easy and is the last step of all the queries. Therefore these tests are not shown here..

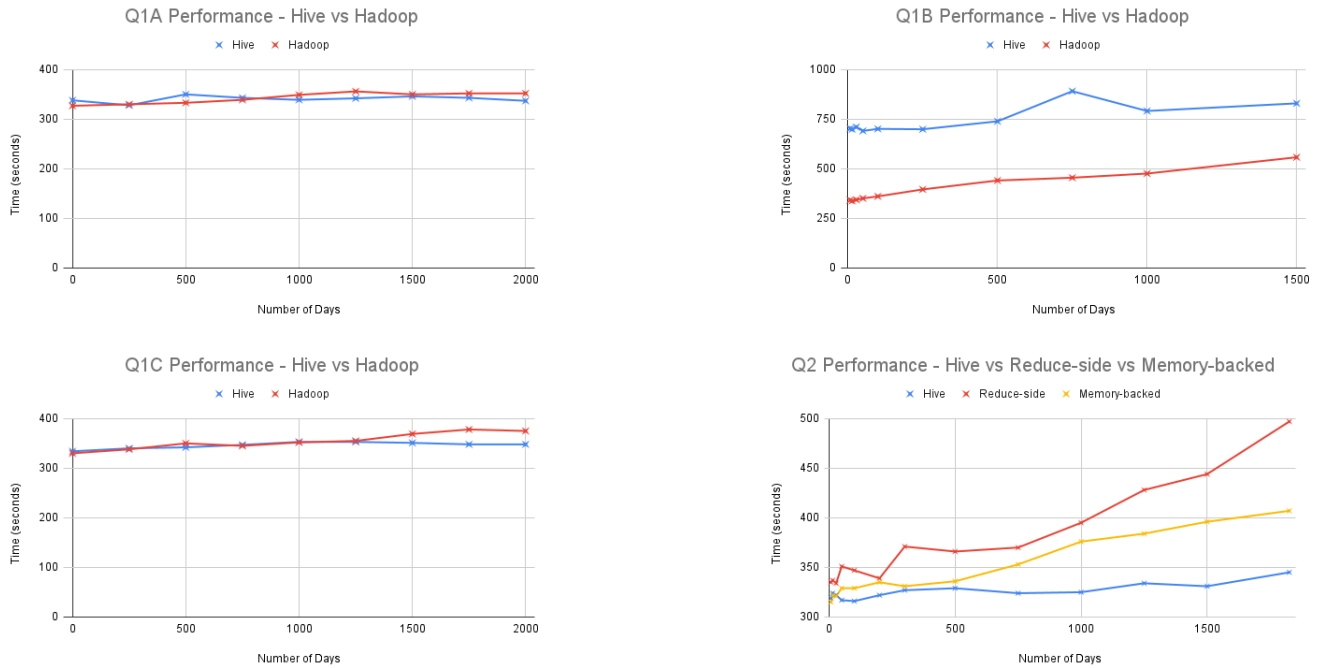


Figure 10: Performance Analysis Results

4 Conclusion

From the performance analysis above, it is common that Hive outperforms Hadoop for both short and long date ranges, such as in queries 1C and 2. However, this may not reflect that Hive is always a better technology to use for querying big data. For example, in query 1B, Hadoop provides consistently quicker results than the Hive query, and this could be a result of the Hive query being complex. Even though Hive tries to optimise the query so that it can run efficiently, it still depends on how the query was written in the first place. Therefore, for complex tasks, Hive may not be the best technology to use as coders are more likely to create solutions which require more stages of processing and perform less efficiently than a solution using Hadoop may.

As well as this, Hadoop also gives the coder more flexibility when writing the solutions. It can be helpful for the user to decide for themselves how many MapReduce jobs they require for a task and other preferences, such as how to handle records which have empty values. Furthermore, knowledge of the data itself can be used, such as data gathered from the EDA, so that the jobs can be optimised more than what Hive may be capable of. For example, knowledge of when sales are more likely to occur each year could be used to optimise query 1A (by filtering out unlikely date ranges) if the accuracy of the results is not as important as the performance.