CS346 Advanced Databases Coursework Report

Oliver Taylor **Group 29** Thomas Ludlow
1810882 1814232

This report documents the solutions and findings of the authors in the CS346: Advanced Databases coursework on Big Data Analytics. This includes the implementation details of a specified set of queries on a sample dataset, using both the Hadoop and Hive MapReduce frameworks. Furthermore, a series of optimisations for the Hadoop queries are documented, and the impact on performance of the queries is tested and evaluated. Ultimately, the effectiveness of the Hadoop and Hive frameworks are compared with respect to both performance and developer experience.

1 HADOOP implementation

1.1 Query 1: “Top-K” pattern

The general structure of these three queries is very similar, with each comprising two successive MapReduce jobs.The general structure of these jobs is based on the WordCount example program recommended in the CS346 module content. [6] In job 1 of each query (Fig. 1), the mapper performs projection and sum-aggregation on the input records to obtain the key-value pairs relevant to the query. The second job follows the “top-K” design pattern, based on the approach described by Miner & Shook. [1] This allows the K pairs with the greatest summed value to be found.

The only difference between each of the three queries are the specific fields each query selects, aggregates and orders by. For instance, query 1a selects *ss_store_sk* as the key, and aggregates and orders by the *ss_net_profit*. Conversely, query 1b selects *ss_item_sk* as the key and aggregates and orders by the *ss_quantity*. Thus, once the initial design pattern had been established, it could be reused for each of the three queries with only minor changes to the data types where appropriate.

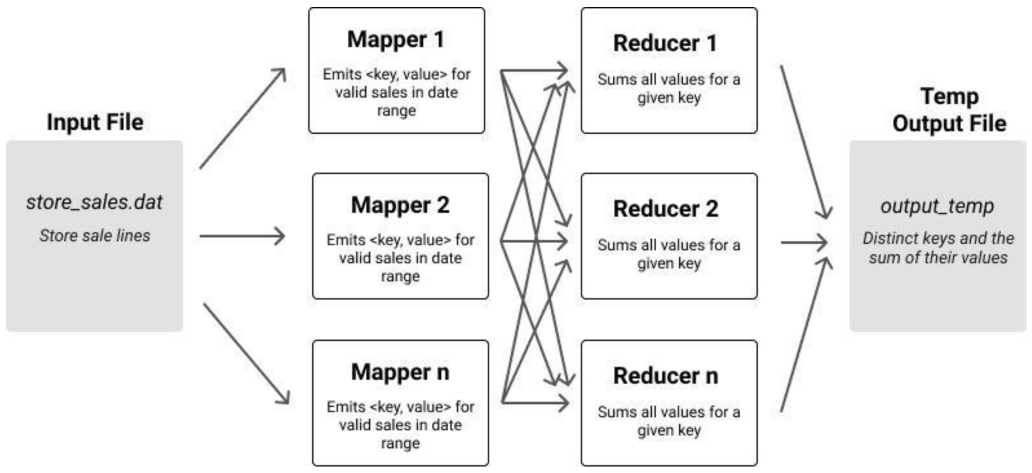

Input File
store_sales.dat
Store sale lines
Mapper 1
Emits <key, value> for valid sales in date range
Mapper 2
Emits <key, value> for valid sales in date range
Mapper n
Emits <key, value> for valid sales in date range
Reducer 1
Sums all values for a given key
Reducer 2
Sums all values for a given key
Reducer n
Sums all values for a given key
Temp Output File
output_temp
Distinct keys and the sum of their values

Figure 1 - *projection and date range map-reduce job*

In order to apply a projection to the *store_sales* records, each sale record attribute was individually parsed into an array, from which the relevant key-value pairs could be extracted. For query 1a, these were *(ss_store_sk, ss_net_profit)* pairs. The queries were also constrained by a prescribed date range; this was achieved by comparing the date attribute of each record with the date bounds supplied as command-line arguments and only emitting the pairs which fell within the range.

Job 1 mapper - *data validation*

When debugging the mapper during development, it was evident that many of the input records contained incomplete or invalid data. As such, it was necessary to check that none of the values relevant to each query were invalid. Generally, there were 3 distinct ways in which data could be invalid; the validation methods applied in each case are detailed in the Fig.2.

Type of invalid data	**Validation method**
Empty attributes (e.g. attribute is parsed as empty string)	Simple *if*-check; only emit the record if the relevant fields are not equal to the empty string.
Invalid numeric data (e.g. profit, quantity, date)	Parsing the data into a numeric type (e.g. Double.parseDouble) throws NumberFormatException. Wrap the method in a *try-catch* and discard the record if an exception is thrown.
Missing attributes (some records had missing "|" delimiters)	Wrap record indexing operations in a *try-catch* and discard the record if an ArrayIndexOutOfBounds exception is thrown.

Figure 2 - *Java stream code for sorting a map by descending value*

Job 1 reducer - *data aggregation*

During the shuffle and sort phase, the pairs outputted by the mapper are grouped by their key and assigned to a reducer; this process is handled by the underlying Hadoop framework. In the reduce phase, the values associated with each key are aggregated by summing their values. This applies to queries 1.a, 1.b and 1.c. Once aggregation is complete, the resulting key value pairs are written to a temporary output file so that they may be accessed by the subsequent job. (Fig.1)

Job 2 mapper - *local top-K*

The input of Job 2 is read from the temporary output file of Job 1. (Fig.3) These input records are divided into equally sized splits, each of which is processed by a single map task. Each map task maintains a value-sorted Map structure which may contain at most K key-value pairs. In turn, each pair in the local split is added to the structure, hence when this process is complete, the Map contains only the top K pairs from the input split, sorted by value. Subsequently, the top-K pairs from each mapper are emitted to the reduce phase. [1]

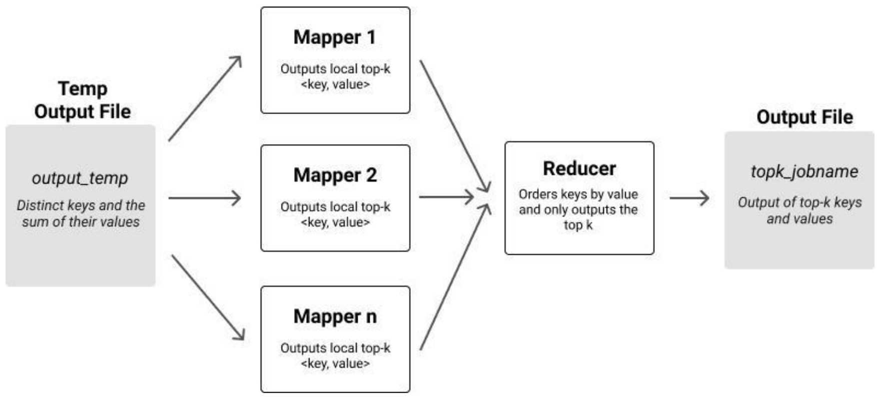

Temp Output File
output_temp
Distinct keys and the sum of their values
Mapper 1
Outputs local top-k <key, value>
Mapper 2
Outputs local top-k <key, value>
Mapper n
Outputs local top-k <key, value>
Reducer
Orders keys by value and only outputs the top k
Output File
topk_jobname
Output of top-k keys and values

Figure 3 - *Top-K design pattern job*

Value-ordered map

The map structure utilised within the Top-K queries was a LinkedHashMap; this subclass of java.util.Map maintains the insertion order of its key-value pairs. [7] In order to guarantee a value-based ordering it was necessary to sort the map upon insertion of each key-value pair. Furthermore, any pairs which were not in the top K were discarded, as per the top-K design pattern. [1] An alternative approach would be to add all pairs to the map, then subsequently perform a single sort operation. Whilst this would require less sort operations, it would be much more memory intensive. For large inputs, the size of the map could exceed the memory of the mapper, resulting in costly IO operations.

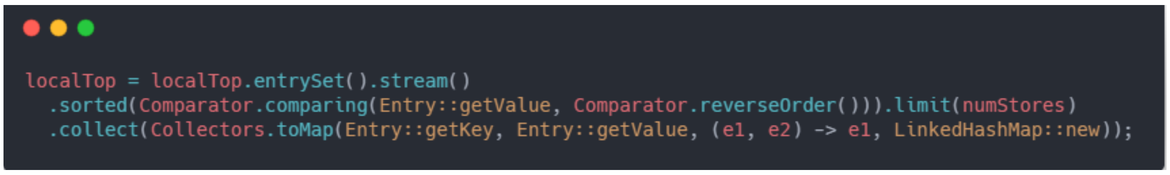

```
localTop = localTop.entrySet().stream()
  .sorted(Comparator.comparing(Entry::getValue, Comparator.reverseOrder())).limit(numStores)
  .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
```

Figure 4 - *Java stream code for sorting a map by descending value*

The aforementioned sorting process was implemented using the *sorted* method from the Java Stream API. (Fig.4) By passing a custom Comparator to the *sorted* method, [8] it was possible to sort the map by value in descending order. The *limit* method was also applied to the output of the *sorted* method in order to truncate the result to the top K key value pairs. Importantly, the Java Stream API allows the limit function to be applied before the stream is *collect*-ed. This prevents any superfluous data from being added to the resulting data structure, hence no subsequent truncation operation is required.

Job 2 reducer - *global top-K*

The final step in the structure of query 1 was to obtain the global top K key-value pairs. This was achieved by setting the number of reduce tasks to 1, thereby directing the outputs of all mappers to a single reducer to be added to a single map structure. (Fig.3) Finally, within the *cleanup* method of the reducer, the map contents would be sorted using the aforementioned stream-sort pattern (Fig.4), producing the *global* top K key-value pairs; these pairs form the query output.

An important consideration is the performance bottleneck imposed by using only a single reducer. When subjected to large inputs, the reducer's memory capacity may be exceeded, in which case sorting would require many costly disk I/O operations. Such large inputs may arise due to two factors: firstly, increasing the number of input records may require the number of input splits *n* to increase, thereby increasing the total number of input records to the reducer, given by *O(K*n)*. Secondly, by increasing K, the number of records emitted from each mapper increases. For sufficiently large *K*, no truncation would occur in the mapper and hence the reducer could be forced to sort *all* of the input records of job 2. Therefore, it is clear that the size of *K* should be restricted when using this pattern. [1]

1.2 Query 2: Reduce-side join

The second query comprised three separate MapReduce jobs in total. The first job was identical to job 1 from query 1a; this job selects all *(ss_store_sk, ss_net_profit)* key-value pairs from store_sales.dat, groups them by key and aggregates the values by computing their sum. The second job involved a new design pattern, called the reduce-side join. This pattern joins each output record of query 1 with the corresponding *s_num_employees* attribute (from stores.dat) for the given *ss_store_sk*. This design pattern was adapted from the examples given in the CS346 module seminars, and will be explained in detail within this section. The third and final job utilised the aforementioned top K design pattern to order the records by the ss_store_sk attribute **in ascending order** (as per the CS346 FAQ), limited to the top K records.

Job 2 mappers - *value tagging*

The second job, which performs the reduce-side join, required two input file paths; one for each table in the join. These were specified using the *MultipleInputs.addInputPath* driver command. Furthermore, a distinct mapper was created for each input file; these were named TagStoreEmployeeMapper and TagStoreProfitMapper.

The purpose of the TagStoreEmployeeMapper was to perform a projection on the input data from stores.dat and concatenate (or "tag") the value string of each key-value pair with the string "emp", as described in Fig.5. Similarly, the TagStoreProfitMapper tagged each value from the job 1 output with the string "prof", however no other projection was required. These pairs would then be emitted from the map phase.

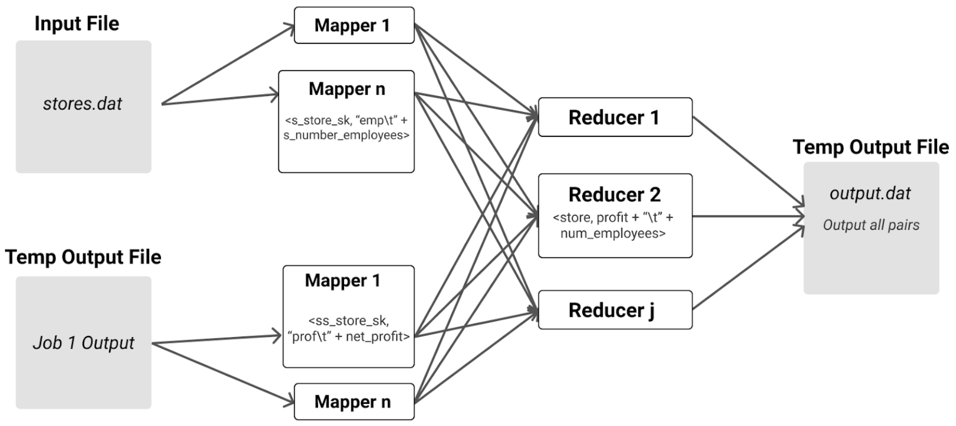


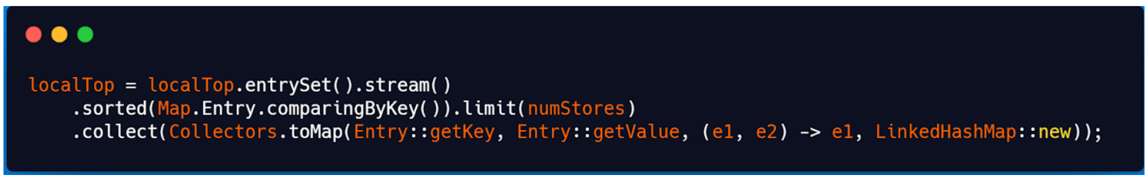
Figure 5 - Query 2, job 2: *reduce-side join*

Job 2 reducers - *joining*

Subsequently, in the shuffle and sort phase, Hadoop groups the outputs from the instances of both mappers, thus forming the inputs to the reduce phase. Using String operations, these tags are removed and the accompanying values are combined into a single tab-separated string, (Fig.5) forming the value emitted alongside the given store key. For each key, if the *net_profit* attribute is empty then a placeholder value of 0 is emitted. This covers the edge case where a store has no sales records. At this stage, if the num_employees attribute is empty, the entire joined record is omitted, since the data is not entirely valid.

Job 3 - *modifications to the top K pattern*

The third and final job requires the output of job 2 (the reduce-side join) to be sorted by *store_sk* (the key). Consequently, the previously-used top K pattern is applied, however the comparator of its stream-sort must be modified to enable sorting on the key, rather than the value. (Fig.6) It should also be noted that the comparator did require a reverse-order clause, since the *store_sk* attributes need to be emitted in **ascending order** (as per the CS346 FAQ).



```java
localTop = localTop.entrySet().stream()
    .sorted(Map.Entry.comparingByKey()).limit(numStores)
    .collect(Collectors.toMap(Entry::getKey, Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
```

Figure 6 - *Stream sort implementing sorting ascending by key*

2 Hive implementation

Having created the Hadoop MapReduce queries for all the specified problems the group focused its efforts on creating equivalent Hive solutions. This approach required the group to use declarative HiveQL queries as opposed to imperative Java code which utilises the Hadoop framework. Hive converts query inputs (written in HiveQL) into Hadoop map-reduce jobs (like those presented in section 1) and runs them in the background, therefore abstracting the complexities of writing MapReduce code away from the developer.

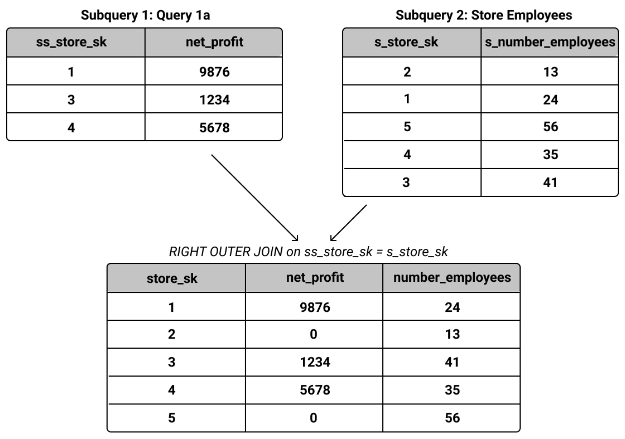
To access the data in Hive the group first created external tables for each datafile provided for the coursework (store_sales.dat and stores.dat). These tables are external to Hive, meaning the data stored isn't directly managed by Hive, yet still available for use. The data types for the columns are provided within the TPC-DS documentation, [2] with the only difference in our implementation being *identifier* type columns being replaced with *bigint* to match the types available within Hive (see [Hive External Table Creation Commands] in the documentation of the software submission). It was also necessary to specify the field delimiters, since the provided .dat files used "|" characters as opposed to "," which is the default for Hive.

The HiveQL queries to match the hadoop code for problems 1a, 1b and 1c all follow a similar structure defined as follows:

1. *SELECT* - The columns from the table to be selected.
2. *FROM* - The external table(s) where the data is stored.
3. *WHERE* - Specific conditions to be met for the row to be considered. For the case of these queries the data is checked to see if it is within the date range provided, and that the data is not null.
4. *GROUP BY* - The value to group the data on, usually the key from the hadoop <key, value> pairs.
5. *ORDER BY* - Which selected value should the returned data be ordered by (usually descending because the top-k pattern is followed).
6. *LIMIT* - Restrict the number of top rows outputted to match the "top-k" pattern.

This structure follows a basic SQL query structure for aggregating data, as taught in CS258: Database Systems.

Similarly, the second query follows this structure but requires a join to connect the two tables. The join utilises two subqueries, one which matches query 1a and the other to get the employee count for each store in the stores table. The rows are then right outer joined on the store identifier attributes in the respective tables to produce an overall output which matches the problem specification -- where the employee count for each store is outputted alongside the net_profit. As the query is using a right outer join there is the possibility that some data from the left subquery will be NULL, which makes the output not match the requirements specified. To fix this, the SQL *COALESCE* operator is used when selecting data, this operator is provided two arguments: *net_profit* and *0* with the net_profit figure being outputted if it's not a null value and 0 otherwise. [4] Figure 7 shows this process, with example results of the individual subqueries and how the data is outputted after being joined.


Subquery 1: Query 1a

ss_store_sk	net_profit
1	9876
3	1234
4	5678

Subquery 2: Store Employees

s_store_sk	s_number_employees
2	13
1	24
5	56
4	35
3	41

RIGHT OUTER JOIN on ss_store_sk = s_store_sk

store_sk	net_profit	number_employees
1	9876	24
2	0	13
3	1234	41
4	5678	35
5	0	56

Figure 7 - *An example of the join query, sorting the output by net_profit descending*

3 Exploratory data analysis

3.1 Date format

In order to observe the structure of dates within the TPC-DS dataset, exploratory data analysis (EDA) was carried out on the date_dim.dat file. According to the TPC-DS documentation [2] this table contained a single row for each calendar day. Additionally, upon observing the contents of the table, it was clear that the order of the surrogate keys was consistent with the chronological order of the dates contained within the table. This implied that these keys could be used as integer comparators for dates they represented.

3.2 Date range in store_sales.dat

Since each of the specified queries is parameterised by a date range, it was necessary to understand the range of dates which spans the dataset in order to control how much of the data is included in the query. Using basic HiveQL queries, the MIN and MAX values of ss_sold_date_sk in store_sales.dat were found. These were 2450816 and 2452642 respectively. Upon testing the queries on the maximal date range it was found that their running times were not extensive, thus this date range was used for the majority of the other experiments and tests within this project.

3.3 Number of records in store_sales.dat

Through a simple HiveQL query (using the count(*) operator) it was possible to count the number of records in the store_sales.dat table: this totalled 115203420. Using a WHERE clause to restrict the date, the number of records in the date range specified for query 1.c was found to be 11668924. Thus it can be concluded that records within this date range account for approximately 10% of the data within store_sales.dat. Given that the sales represented in store_sales.dat occur over a period of 1824 days, and approximately 10% of the sales occur over a 251 day period (the date range specified in query 1.c), it is apparent that the records in store_sales.dat are not uniformly distributed over the date range.

3.4 Number of aggregated keys in each query

When analysing the performance of each query, it is important to understand the approximate amount of data that each MapReduce job processes. For all queries in this project, the number of records in the input of job 1 depends entirely on the date range specified, since this range will correspond to a number of records in store_sales.dat. However, it is important to note that the number of records in the input of job 2 (for queries 1.a, 1.b and 1.c) also depends on the number of distinct values in the group-by column specified in query 1. (Fig.8) For instance, query 1.a groups by ss_store_sk of which there are 58 unique values in store_sales.dat. Hence, job 1 will output at most 58 records. Conversely, there are 1824 distinct dates in store_sales.dat, therefore job 1 of query 1.c will output at most 1,824 records; each denoting the total net profit on a specific day.

Attribute	Distinct values
ss_store_sk	58
ss_item_sk	52000*
ss_sold_date_sk	1824

* a restricted date range is used when this attribute is queried.

Figure 8 - *Data analysis results*

These results certainly have performance implications with respect to job 2 (finding the top K) since they effectively dictate the number of key value pairs it receives as input. However, it is also important to consider the effect of this data on job 1. For instance, fewer distinct group-by attributes will likely lead to a greater number of aggregation operations overall, hence putting more pressure on the first MapReduce job. These considerations are explored further on in section 5.4 of this report.

4 Optimisations

4.1 Mapper input split size & slowstart

The first and most significant optimisation presented is to adjust the minimum input split size of the *job 1* mapper of each query. [10] By observing the Hadoop configuration files, it was evident that the minimum input split size was set to 0 by default. Consequently, Hadoop created 120 splits from the input data, each of which was assigned to a single map task. Due to the small size of the data splits, each map task was initially observed to execute in approximately 6 seconds (subject to server load). By increasing the minimum split size, more data could be allocated to each task, in turn reducing the processing overheads incurred by launching a large number of map tasks. However this also increased the execution time and memory consumption of each map task.

Initially, it was deduced that the number of map tasks should ideally be a multiple of 6, since the cluster was able to handle a total of 6 map or reduce jobs concurrently due the allocated virtual core limitations. This was implemented using the FileInputFormat.setMinSplitSize method. [9] Through experimentation, it was deduced that the smallest number of splits (without exceeding the memory limits of the mapper) was 12. This was later reduced to 11 to allow the subsequent *reduce* task to begin in parallel with the second round of map tasks; this modification was perceived to provide a small decrease in run time.

On occasion, it was observed that a map task may fail and rejoin the task queue, causing the reduce task to begin prematurely. As such, the failed map task would often be delayed until the completion of all other map tasks. To avoid this, the *slowstart.completedmaps* parameter was set to 0.75. This prevents any reduce tasks from being launched until at least 75% of map tasks are completed, thus reducing the probability that a failed map will be blocked by a waiting reduce task. Furthermore, by delaying the reducer task start, the job would have computation to be performed, rather than sitting dormant and taking a CPU core whilst it waited for the mappers to complete.

4.2 Combiner

The simplest optimisation applied was the introduction of a combiner. This adds an extra step to each map task which performs intermediate aggregation on the local mapper outputs. Hence, if a mapper has multiple records with the same key, they are combined and hence less records are emitted to the reduce phase. This optimisation was simple, because each combiner could simply utilise the existing reducer classes. Therefore, this only required the reducer class object to be passed to the *Job.setCombinerClass* method.

4.3 Uber mode

The final optimisation which was ultimately included in the final iteration of the Hadoop code was the activation of Uber mode. By observing the run times of the MapReduce jobs, it was clear that the Top-K jobs were very lightweight since their run time was very minimal, and they often only utilised a single mapper and a single reducer. Uber mode is designed to run all job tasks within a single JVM, and hence doesn't require the ResourceManager to schedule each constituent job. [5] This removes the overhead of resource allocation, which can dramatically reduce run time for small jobs. This was enabled for job 2 of queries 1.a, 1.b and 1.c, as well as jobs 2 and 3 of query 2. It should

4.4 Compression

In addition, two compression optimisations were tested. These were mapper output compression and job output compression respectively. These optimisations performed poorly when tested, thus were not ultimately added to the implementation. In practice, these forms of compression would likely be more beneficial in distributed systems which span large physical distances, incurring heavy data transfer overheads. By compressing the outputs, the data transfer overheads can be greatly reduced at the cost of running a compression algorithm and a decompression algorithm at the source and destination respectively. For smaller systems, such as in this project, these compression overheads greatly exceed the performance gained from decreasing data transfer costs and are hence not feasible.

5 Experiments

To ensure that the optimisations implemented we're improving job performance, a large number of experiments were performed with varying implementations. All of these optimisations were applied to the Hadoop code, as the Hadoop framework provides fine-grain control over many aspects of the job when compared to the lack of freedom Hive queries provide.

5.1 Performance metrics

As both Hive and Hadoop use map-reduce jobs to perform the underlying data analysis, developers are able to use metrics provided within the Hadoop cluster management interface to compare performance between queries. Many metrics are displayed, but the most comparable ones are memory megabyte seconds and CPU virtual core seconds. These specifically provide the developer with a measure of memory and CPU consumption over time. Ideally, these values should be reduced as much as possible through optimisations to reduce the resources consumed by each job. The length of the job (in seconds) can also be used as a metric to judge the overall job performance. It should be noted that all of these metrics may be affected by variable load on the server, since multiple users could be running map-reduce jobs at any given time.

Despite CPU vcore-seconds and memory megabyte-seconds being good overall performance indicators, some optimisations specifically aim to reduce other job metrics. A good example of this is compression optimisations which aim to reduce the overall amount of data being transferred during a job. In these cases, other job-specific metrics such as HDFS bytes written and map output bytes can be recorded in order to appropriately measure the improvement in performance.

5.2 Experimental method

To enable the group to judge performance improvements across different code optimisations, a baseline reading was taken for all queries, without applying any optimisations. Following this, optimisations were tested cumulatively, with performance metrics being recorded for each iteration of the optimised code. These benchmarks were then compared against the readings from the previous optimisation iteration to judge whether the changes were beneficial to job performance. Each tested query was run three times, and an average of each metric was taken. This was done to reduce the effect of server load variability on the overall results. In circumstances where load varied drastically between runs, the fastest benchmark was recorded as the final result. In cases where one clear outlier was present, the result was disregarded.

5.3 Results

Figure 9 details the results of running each iteration of optimisations and recording the performance metrics as previously described. The *Addition* column denotes the optimisation added in each iteration of the code. The table omits the results of queries 1b and 1c due to their inherently similar structure when compared to query 1a. Nevertheless, specific statistics comparing their performance to query 1a will be discussed in the evaluation below.

Query 1a					*Query 2*			
Addition	**MB-s**	**CPU-s**	**secs**		**Addition**	**MB-s**	**CPU-s**	**secs**
Hadoop					**Hadoop**			
None	3156318	2581	432		*None*	2479371	1977	365
Split size	1742499	1398	286		*Split size*	1319477	1012	258
Combiner	1334573	1102	183		*Combiner*	1289382	1000	227
Uber mode	1312170	1089	176		*Uber mode*	984001	786	155
Hive					**Hive**			
N/A	3035797	2491	389		*N/A*	2786667	2256	387

Figure 9 - *optimisation experiment results for query 1a and query 2 respectively*

5.4 Evaluation of results

According to the data obtained across the four distinct queries, the runtimes of the Hive implementations are between 6% slower and 16% faster than the equivalent un-optimised Hadoop queries. Due to the variable load on the server, it is difficult to discern whether these differences are significant, however they do show that the run times of the default approaches are not drastically different in any case.

When the split size optimisation is applied, the greatest improvement is observed in query 1c which gains a 2.2 times speedup over the un-optimised code. The least improved query is number 2, the reduce-side join, which benefits from a 1.4 times speedup. This decreased effectiveness is likely because query 2 contains an additional job (compared to the others) which is not suited to this optimisation. Overall, this optimisation appears to provide the most significant performance increase in each query, with the exception of query 2.

The next optimisation added a combiner to job 1 of each query. The performance benefits observed here varied significantly by query. Query 1a had a speedup of almost 1.6 times, whereas query 1c had a speedup of only 1.025 times, which can certainly be considered negligible. During testing, it was observed that the combiner of query 1a reduces the size of the data by a factor of 33,800 times, whereas query 1c only sees a decrease by a factor of 4,100. This is supported by the results of the EDA which showed that query 1a aggregates on the least numerous attribute (at only 58 distinct values), which would imply that the data of query 1a can be combined into the fewest number of records, hence justifying the observed effectiveness.

The final optimisation applied Uber mode to the smaller jobs within each query. To effectively evaluate its performance it is necessary to consider the performance of these smaller jobs in isolation. However, due to the issue with variable server load, the duration of these jobs were observed to vary quite wildly between runs. The most representative example of the effect of uber mode was job 2 of query 1a which ran in 19 seconds without uber mode, and 11.5 seconds with uber mode applied, giving a job speedup of 1.65 times. To better demonstrate this speedup, further tests would be required at times when the server is under less load.

In summary, the optimised Hadoop queries provided a speedup of 2.46 times to query 1a; 2.21 times to query 1b, 2.48 times to query 1c; and 2.35 times to query 2, relative to the unoptimised Hadoop queries. Therefore, it can be deduced that the run time of all queries was significantly improved by these optimisations. Additionally, when compared to the Hive queries, the optimised Hadoop queries were 2.21 times faster for query 1a; 1.86 times faster for query 1b; 2.51 faster for query 1c and 2.50 times faster for query 2. Thus, it can be concluded that the optimised Hadoop queries are significantly faster than the Hive queries.

Furthermore, none of the optimisations appeared to consistently and significantly increase either CPU-seconds or megabyte-seconds across all queries, hence these optimisations do not appear to impose a trade-off between run time and compute resources.

6 Comparisons between Hive and Hadoop

6.1 Implementation complexity

When comparing the speed and ease of development between both platforms, it was clear that creating Hive queries was both easier and quicker than the equivalent Hadoop implementations. This can be largely attributed to the declarative nature of Hive queries in comparison to the imperative Hadoop framework code. For queries made in Hadoop, much more consideration must be given to the creation of the complex implementation logic, thus increasing the time taken to create the query and also the potential for coding errors. Most importantly, the Hive query language is likely familiar to most developers due to its similarities to SQL. According to a 2020 survey by StackOverflow 54.7% of developers use SQL [3] therefore most developers have existing knowledge which can be applied to HiveQL. Overall, this makes HiveQL very accessible to the majority of developers.

6.2 Performance

Overall, Hadoop was found to be more performant than Hive across the board (according to section 5.4). This is to be expected as Hadoop provides a greater number of configuration options for the developer, allowing for fine-grained control and optimisations to suit the specific data and task at hand. Conversely due to the greater level of abstraction provided by Hive, the solutions it produces are more generalised and are therefore performance is less tuned to the particulars of the query.

6.3 Real world use cases

In practicality, the use of these frameworks would depend upon the context in which the data analysis is performed. With Hive queries being observed, on average, to be less performant than optimised Hadoop queries, it is expected that developers would favour Hive for data analysis in a context where query performance is unimportant and ease of implementation is prioritised. A good example of this is the exploratory data analysis undertaken for this project, in which each query was only run once and did not need to be highly optimised.

On the other hand, Hadoop is more favourable in applications where the job is likely to run many times. This is likely to be seen commercially in situations where data can be used to enhance sales for a business. A good example of this is likely to be a periodically-running marketing email, where the data used to target individual users is found through analytics running on a big data platform such as Hadoop. Having said this, a developer wouldn't want to use either of these frameworks to power real-time blocking requests as even the minimum time of any query in this report is 148 seconds, far too slow for any blocking query. A better approach would use a cache, making a trade-off between a faster response time and having potentially stale data to power the user requests. The developer can then create a CRON job which runs asynchronously and refreshes the cache periodically to minimise the downsides of this approach.

References

[1] Miner, D. and Shook, A., 2012. *MapReduce design patterns: building effective algorithms and analytics for Hadoop and other systems*. " O'Reilly Media, Inc." pp.58-65.

[2] Transaction Processing Performance Council, 2015. TPC BENCHMARK ™ DS. Available online: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.1.0.pdf, Last accessed: 24/03/2021 pp.18-25.

[3] Stack overflow, 2020. Stack overflow developer survey 2020. Available online: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages, Last accessed: 24/03/2021.

[4] Apache software foundation, 2019. Language manual UDF. Available online: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-ConditionalFunctions, Last accessed: 24/03/2021.

[5] Anshudeep, 2018. Uber mode in Hadoop. Available online: https://www.netjstech.com/2018/04/uber-mode-in-hadoop.html, Last accessed: 24/03/2021.

[6] Apache software foundation, Apache Hadoop 3.3.0 - MapReduce Tutorial: Example: Word count v1.0.
Available online:
https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0, Last accessed: 24/03/2021.

[7] Oracle, LinkedHashMap (Java Platform SE 8).
Available online: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html, Last accessed: 24/03/2021.

[8] Oracle, Stream (Java Platform SE 8): Comparator,
Available online:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#sorted-java.util.Comparator-, Last accessed: 24/03/2021.

[9] Apache software foundation, FileInputFormat (Apache Hadoop Main 3.3.0 API): setMinSplitSize, Available online:
http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/FileInputFormat.html#setMinSplitSize-long-, Last accessed: 24/03/2021.

[10] UshaK, 2018. How to improve map-reduce performance.
Available online:
https://knpcode.com/hadoop/mapreduce/how-to-improve-map-reduce-performance/, Last accessed: 24/03/2021.

Individual Appendix - Thomas

Contributions

- Setup the system and development environment
- Created the first 3 Hadoop map-reduce queries
- Created hive external tables
- Created the first 3 hive queries
- Helped Ollie research optimisations despite me not implementing them myself
- Ran benchmarks on optimisations long into the night when the server load was minimal
- Testing of the outputs (to see if they match and are generally correct following our EDA)
- Debugged the queries

What I learnt

Nothing within this coursework I had done before and therefore everything was a learning experience. In this coursework I learnt about Hadoop (Mappers, Reducers, Combiners, Job management, Optimisations, Partioners). I also learnt about join techniques in Hadoop despite us not doing them as we undertook some research to learn more about them. I also learnt about Hive, which is my preferred way of writing queries. By creating the tables for Hive I learnt about the differences between tables and a bit about how hive organises data. I also learnt about the similarities and differences between SQL and HiveQL. We originally intended on implementing a few hive optimisations so I learnt about hive indexing, however this isn't included in our final project or report. I also learnt the importance of metrics, as me and Ollie used the web interface for Hadoop jobs to understand a lot about our jobs so we could optimise them more heavily.