Project in Data Intensive Systems (DAT500)

Aleksander Vedvik
University of Stavanger, Norway
al.vedvik@stud.uis.no

Vidar André Bø
University of Stavanger, Norway
va.bo@stud.uis.no

ABSTRACT

A pipeline designed to analyze the correlation between the stock market and news is used to evaluate the performance of several Spark optimizations. The results show that reducing the usage of UDFs and increasing the amount of memory available to the Spark executors is a way to increase the performance of this workload.

KEYWORDS

Stocks, News, Spark, Hadoop, Performance optimization

1 INTRODUCTION

The use-case for the application is to see if there is a correlation between stock prices and how the various companies and sectors are discussed in the news. This will be achieved by running a sentiment analysis on the news, and then comparing how the related stocks are performing in that time period. The Spark section of the project will be optimized using the following techniques: Increasing executor memory, removal of UDFs, dataframe caching, and changing join strategy. These will be tested to find the most effective ones for the given workload. The results section of this report will mainly feature the results of the optimization.

2 METHOD

Apache Hadoop and Spark were used to design a pipeline to determine if news articles affect stock prices.

2.1 Design

2.1.1 Hadoop. The stock and news datasets are converted to csv using Python MRJobs. For the stock dataset, each JSON file represents a single stock and its entire price history. Because of this, the MRJobs are constructed using `mapper_raw` instead of the standard `mapper`. This makes sure that each mapper gets served one entire file as it would be impossible to process the stock across multiple mappers. The result is a csv file where each row contains the price of a single stock for a given day. The news dataset is converted from JSON to a csv where each row is a single news article. This MRJob uses a normal `mapper` since there are multiple news in a single file. The MRJobs will output the result to HDFS.

2.1.2 Spark. The results from Hadoop were further processed in Spark. The pipeline in Spark was divided into two scripts: sentiment analysis and main. The sentiment analysis was time-consuming, and dividing the code made it simpler to optimize each part of the pipeline separately.

First, sentiment analysis was run on the content of each news article as shown in Figure 1. Afterward, the month in which the article was published was added to the dataset in the format "YYYY-MM". The news dataset was saved as a temporary parquet file for further processing in the main script.

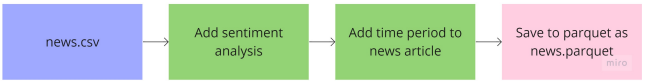

Figure 1: Overview of sentiment analysis

The next step in the pipeline was to process the stocks dataset as shown in Figure 2. Only the symbol name of the stocks was stored in the results from Hadoop. Hence, a separate dataset was joined to the stocks dataset to obtain the company name of all the stocks. The change in stock price per month was also calculated for each stock before the result was stored in a delta table. New stocks were added and existing stocks were updated, such that all records in the delta table were unique. All stocks from the delta table were then fetched and stored in a dataframe.

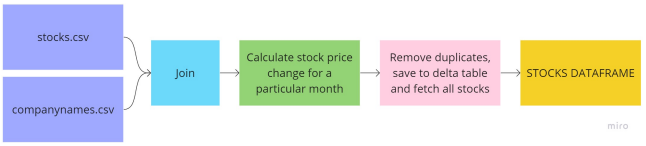

Figure 2: Overview of how the stocks dataframe is transformed

As shown in Figure 3, the news dataset was read from the parquet file created from the sentiment analysis stage and compared to all news articles stored in the delta table. Only new news articles were further processed.

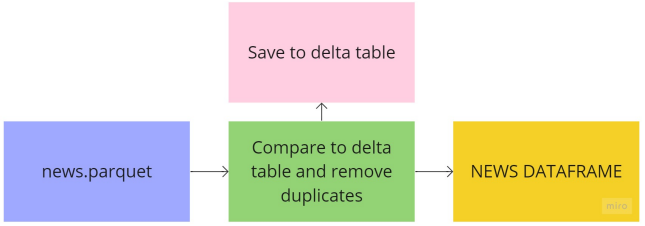

Figure 3: Overview of how the news dataframe is transformed

Furthermore, the news dataframe from Figure 3 was joined with a dataset containing the most common words from each financial sector as shown in Figure 4. Each word in the sectors dataset was joined with each row in the news article dataframe. This made it possible to search for occurrences or mentions in the content of each news article. Afterward, the dataframe was filtered such

Project in Data Intensive Systems (DAT500), IDE, UiS
2023.

that only news articles containing words from the sectors dataset were kept. This transformation added a column named sector to the news dataset. Subsequently, the dataframe was grouped by the sector name and the time period of each news article. Hence, the dataframe consisted of the number of articles for each sector per month. Thereafter, the dataframe was joined with the stocks dataframe from figure 2, and the stock price change was added for each sector. The last transformation was adding the correlation between the stock price change for each sector and the sentiment of the news articles for this sector. Ultimately, the dataframe was grouped only on the sector such that the average correlation was shown. The results from the last two transformations were stored in a delta table, where new entries were added or existing entries were updated.

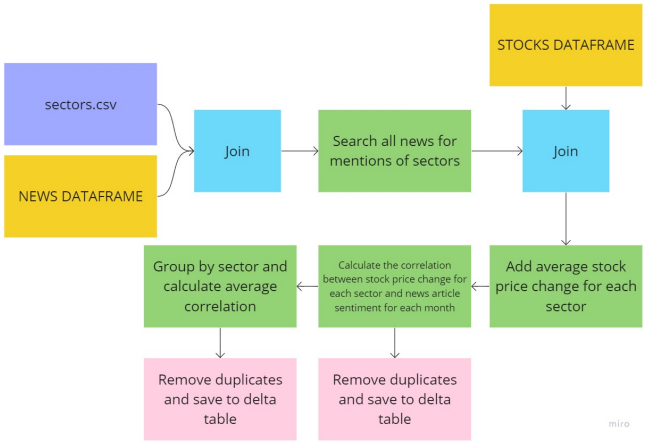

Figure 4: Overview of how the stocks and news dataframes are joined to find sectors

The last step of the pipeline is shown in Figure 5. This is very similar to the process in Figure 4, but each news article was searched for occurrences or mentions of stocks instead of sectors. The proceeding transformations are more or less identical to the transformations done to the sector dataset, but now concerning stocks. The results were saved to delta tables.

Figure 5: Overview of how the stocks and news dataframes are joined to find stocks

2.2 Optimization

2.2.1 Memory. The default amount of memory available for a Spark executor is 1 GiB [2]. Increasing this has the potential of increasing performance [3], and since each VM has 8 GiB RAM, the available memory will be set to 6 GiB. This is done using the `spark.executor.memory` setting in the configuration.

2.2.2 Cache. Because of the increase in the executor memory, there should be more room for caching dataframes in memory. For some workloads, this can lead to a quite drastic performance increase [3]. This optimization was invoked by calling `.cache()` on frequently used dataframes. Spark supports fractional caching of dataframes, but there is a performance penalty if the entire dataframe does not fit in cache [1]. This might be a concern for the largest of the dataframes, such as the one containing the stocks. 4 dataframes were cached, those being the frames containing news, sectors, company names, and stocks.

2.2.3 User Defined Functions. Spark allows the user to define custom functions, called user defined functions (UDF), which can be used in Spark SQL. One of the issues with PySpark UDFs is that they have slower performance than Scala UDF because the PySpark UDFs require data movement between the JVM and Python, which is quite expensive [1, p. 114-115].

One optimization that drastically reduced the overall execution time of the pipeline was to get rid of UDFs. First, a UDF was created to search all news articles for mentions of either stock names or words describing a sector. A simplified version of the function is shown in listing 1. A similar UDF was defined to search the news articles for sectors, as shown in listing 2. The UDFs worked by going through every single stock/sector provided to the UDF and comparing it to every word in the text of the news article. The list of stocks provided to the find stocks UDF comprised all unique entries of stocks in the stocks dataframe, which exceeded more than 5000. The list of sectors was slightly different. The sectors dataset was created such that the 11 stock market sectors [4] are placed in the first row. Afterward, each subsequent row consists of one word characterizing the sector, such as the word "Bank" which is most likely related to the sector "Financials". In total, there were about 10-20 words describing each sector in the sectors dataset. Each UDF returned the stock symbol or sector name as a list, respectively.

```
def find_stocks_udf(news_article_text, list_of_stocks):
    stocks = set()
    for stock in list_of_stocks:
        for word in news_article_text:
            if stock == word:
                stocks.add(stock)
                # Article mentions the stock, hence
                # no need to go through the rest of
                # the article:
                break
    return list(stocks)
```

Listing 1: Simplified version of the UDF for finding mentions of stocks in the text of a news article

```
def find_sectors_udf(news_article_text, list_of_sectors):
    sectors = set()
    for sector_row in list_of_sectors:
        for i, key_word in enumerate(sector_row):
            for word in news_article_text:
                if key_word == word:
                    # The names of the sectors are
                    # stored in the first row:
                    sectors.add(list_of_sectors[0][i])
                    # Article mentions the sector, hence
```

```
                    # no need to go through the rest of
                    # the article:
                    break
    return list(sectors)
```

Listing 2: Simplified version of the UDF for finding mentions of sectors in the text of a news article

Each UDF was replaced by transformations on the dataframes instead. Listing 3 shows a simplified version of the method that replaced the find stocks UDF. This was done by doing a cross-join of the news dataframe and the stocks dataframe. The stocks dataframe was first modified to only contain unique entries and only the necessary columns. Then the joined dataframe was filtered by the condition that the name of the stock must be mentioned in the content of the news article. Finally, the dataframe was grouped by the symbol of the stock and the time period (month). Three additional columns were added using the aggregate function, with the first counting the number of rows, which represented the number of articles a particular stock was mentioned in. The second column was simply the name of the stock, and the third was the average of the sentiment of all news articles for the given time period. Listing 4 shows a similar transformation which replaced the find sectors UDF. However, it is slightly different due to how the sectors dataframe was organized. Instead of joining all sectors with all news articles, only one sector was joined at a time. Then each news article was filtered on the condition of containing one of the words characterizing the sector. Subsequently, duplicates were dropped to prevent counting the same article multiple times when grouping and a column containing only the name of the sector was added to each row in the dataframe. Ultimately, the dataframe was grouped by sector and time period, similar to the stocks. This transformation was done for all sectors, and thus a union was performed on the result for each sector to coalesce all rows into one dataframe.

```
def search_for_stocks_and_aggregate(df_news, df_stocks):
    return df_news.join(df_stocks).filter(
        col("Title").contains(col("name"))
        | col("Text").contains(col("name"))
        | col("ShortDescription").contains(col("name")))
        .groupBy("symbol", "time_period")
        .agg(count("*").alias("number_of_articles"),
            first("name"),
            avg("sentiment"))
        .orderBy(col("symbol"), col("time_period").desc()
    )
```

Listing 3: Simplified version of the transformation that replaced the UDF for finding mentions of stocks in the text of a news article

```
def search_for_sector(df_news, df_sectors, sector: str):
    return df_news.join(df_sectors.select(sector))
        .filter(
        col("Title").contains(col(sector))
        | col("Text").contains(col(sector))
        | col("ShortDescription").contains(col(sector)))
        .dropDuplicates(["Title"])
        .withColumn("sector", lit(sector))
        .groupBy("sector", "time_period")
        .agg(count("*").alias("number_of_articles"),
            avg("sentiment"))
```

Listing 4: Simplified version of the transformation that replaced the UDF for finding mentions of sectors in the text of a news article

2.2.4 Join Strategy. Join operations are a common type of transformation where two datasets are combined. Spark offers a series of join transformations, such as inner join, cross join, etc. Common for all joins is that they trigger a large amount of data movement, often referred to as shuffle. The two most common join strategies are broadcast hash join and sort merge join [1, p. 187]. Broadcast hash joins are suited for joins where one of the datasets is much smaller than the other, and sort merge joins are efficient when two large datasets are sortable over a common unique key [1, p. 187][5, p. 3].

Different join strategies were tested to see if they impacted the performance of the pipeline. The dataframe containing the stocks was transformed to only contain unique entries based on the symbol of the stock before it was joined with the news articles. Hence, the stocks dataframe was significantly smaller than the news dataframe, meaning a broadcast hash join would be well suited. Also, each executor was allocated enough memory to fit the whole stocks dataframe. Similarly, the transformation that joined sectors and news articles was also well suited to be a broadcast hash join. Broadcast hash join is the default behavior for dataframes up to 10 MiB, but could also be invoked by calling `broadcast()` on the dataframes in the join operation.

Sort merge join is another strategy that could improve performance when two large datasets are merged. The transformations `groupby()`, `distinct()`, and `orderby()` are used multiple times throughout the pipeline, and thus a sort merge join could potentially improve the execution time on some of the transformations. One transformation joined all news articles for a particular period with all the stocks for the same period. This transformation is the second join in Figure 4. These are two large datasets that could potentially benefit from a sort merge join instead of a broadcast hash join. Sort merge join was tested by setting spark.sql.autoBroadcastJoinThreshold to -1, which forces Spark not to use broadcast hash joins.

2.3 Implementation

The project repository can be found here: https://github.com/vidarandrebo/dat500-project.

The Spark code was separated into several files to keep it organized and easier to read. All transformations were put in a file transformations.py which was used by both the sentiment analysis script and the main script. Similarly, all user defined functions were added to a separate file. Also, to separate concerns, all storage-related queries in Spark were moved to the file storage.py. Lastly, a show script was created to print all the results stored in the delta tables.

Data schemas were enforced on all datasets read from either csv or parquet files, meaning fields such as prices or timestamps were interpreted as correct types.

The pipeline was made such that the results were stored in delta tables, but only new entries were added to ensure that there were no duplicates. Otherwise, the results were used to update the entries

in the delta tables. Hence, new data could be processed and added to the existing results.

3 EXPERIMENTAL EVALUATION

Different optimizations were tested as described in the previous section. All optimizations used the configuration with 6 GiB of memory instead of the baseline with 1 GiB of memory. This was because some of the optimizations, such as caching, had a higher consumption of memory. The longest-running stage is referred to the transformations related to searching the news dataset for stocks or sectors and aggregating the results.

3.1 Experimental setup

The cluster used for testing is configured using OpenStack. There are in total 4 VMs, 1 called "namenode", which is used as the driver node, and 3 called "datanodes", which are used as executors. Each VM is configured with 4 virtual CPU cores and 8 GiB of ram. The VMs are running Ubuntu 20.04 and kernel 5.4.0. All tests are performed using Spark 3.3.1 and Hadoop 3.3.5. The datasets used consist of 7.4 GiB of stock pricing history and 556 MiB of news, both in JSON format. In addition, there are some small files containing a conversion table from stock symbol to company name and a list of sectors.

3.2 Results

All tables show the running time, garbage collection (GC), etc. for each scenario. All optimizations were run with 6 GiB memory except for baseline 1 GiB.

Table 1: 1 GiB vs 6 GiB of executor memory

	Task time	GC Time	Peak JVM Memory On-Heap / OffHeap
Executor 0, 1 GiB	1.7 h	4.0 min	923.8 MiB / 179.8 MiB
Executor 1, 1 GiB	1.7 h	4.1 min	986.5 MiB / 182.4 MiB
Executor 2, 1 GiB	1.7 h	3.9 min	824.3 MiB / 179.4 MiB
Driver 1 GiB	37 min		
Executor 0, 6 GiB	1.6 h	1.3 min	2.5 GiB / 182.9 MiB
Executor 1, 6 GiB	1.7 h	1.4 min	3.1 GiB / 185.1 MiB
Executor 2, 6 GiB	1.6 h	1.3 min	3.3 GiB / 181.7 MiB
Driver 6 GiB	35 min		

Table 2: Spill at stage 4 of main with 1 and 6 GiB of executor memory

	Spill Memory	Spill Disk
Executor 0, 1 GiB	0 B	0 B
Executor 1, 1 GiB	97.3 MiB	14.3 MiB
Executor 2, 1 GiB	97 MiB	11.9 MiB
Executor 0, 6 GiB	0 B	0 B
Executor 1, 6 GiB	0 B	0 B
Executor 2, 6 GiB	0 B	0 B

Table 1 shows that there is a reduction in time used for garbage collection as well as a higher maximum memory utilization by the JVM when increasing the memory available to the executors.

Table 3: Cache vs baseline with 6 GiB of executor memory

	Task time	GC Time	Input
Executor 0, 6 GiB, baseline	1.6 h	1.3 min	1.2 GiB
Executor 1, 6 GiB. baseline	1.7 h	1.4 min	2 GiB
Executor 2, 6 GiB, baseline	1.6 h	1.3 min	2.2 GiB
Driver 6 GiB, baseline	35 min		
Executor 0, 6 GiB, cache	1.6 h	56 s	2.4 GiB
Executor 1, 6 GiB, cache	1.7 h	59 s	3 GiB
Executor 2, 6 GiB, cache	1.6 h	59 s	3.5 GiB
Driver 6 GiB, cache	35 min		

Table 4 shows the metrics for the longest running stage for the baseline configuration with 6 GiB memory. The stage is divided into 13 tasks with all but one partition being around 12 MiB. The last partition was about 6 MiB. The median running time of each task was 10 min.

Table 4: Baseline 6 GiB - Longest running stage

Metric	Min	Median	Max
Duration	5.0 min	10 min	10 min
GC	2 s	5 s	5 s
Input size / Records	6.1 MiB / 57996	11.9 MiB / 116423	12.1 MiB / 117001
Shuffle Write Size / Records	267.9 KiB / 5474	423.3 KiB / 9036	427.6 KiB / 9110
Task Deserialization Time	6.0 ms	49.0 ms	67.0 ms

Table 5 shows the metrics for the longest running stage when UDFs were used as described in 2.2.3. Each executor was configured with 6 GiB memory. Compared to the non-UDF version, which is shown in table 4, it is possible to see that the median execution time for each task is significantly longer. Similarly, the shuffle write size is also larger. Removing UDFs yielded a decrease of about 76 % in execution time and a 73 % decrease in shuffle write size for the longest-running stage.

Table 5: UDF 6 GiB - Longest running stage

Metric	Min	Median	Max
Duration	11 min	43 min	48 min
GC	9 s	4.9 min	4.9 min
Input Size / Records	6 MiB / 57996	11.8 MiB / 116423	12 MiB / 117001
Shuffle Write Size / Records	1.3 MiB / 72916	1.6 MiB / 89549	1.6 MiB / 91034
Task Deserialization Time	19.0 ms	0.1 s	0.1 s

Table 6 shows the aggregated metrics for the longest-running stage when using UDFs, sort merge joins and the baseline with 6 GiB memory. The task time for the baseline and the sort merge join is quite similar, but the shuffle write size is about 234 % higher when using sort merge joins. The task time and shuffle write size is even higher when UDFs were used. Also, there was an increase in peak JVM memory when using UDFs.

Table 6: Aggregated metrics by executor for longest running stage when using UDFs, Sort Merge Join and baseline

Metric	Executor 0	Executor 1	Executor 2
Task Time	41 min	45 min	40 min
Task Time / GC Time (UDF)	3.1 h	3.1 h	2.9 h
Task Time / GC Time (SortMergeJoin)	43 min	43 min	43 min
Shuffle Write Size / Records	1.6 MiB / 36015	1.9 MiB / 41588	1.7 MiB / 36225
Shuffle Write Size / Records (UDF)	6.4 MiB / 358675	7.7 MiB / 430315	6.4 MiB / 360088
Shuffle Write Size / Records (SortMergeJoin)	3.9 MiB / 36026	4.2 MiB / 39272	4.1 MiB / 38530
Peak JVM Memory OnHeap / OffHeap	3.5 GiB / 171.4 MiB	3.4 GiB / 173.1 MiB	2.6 GiB / 168.6 MiB
Peak JVM Memory OnHeap / OffHeap (UDF)	4.8 GiB / 162.6 MiB	4.5 GiB / 164.3 MiB	4.5 GiB / 168.6 MiB
Peak JVM Memory OnHeap / OffHeap (SortMergeJoin)	3.3 GiB / 185.6 MiB	2.9 GiB / 185.9 MiB	2.9 GiB / 182.2 MiB

Table 7 shows the running times for the main part of the pipeline for each optimization. By increasing the memory of each executor the running time was reduced by 5.4 %. Removing the most time-consuming UDFs the duration was reduced by 69 %. Furthermore, cache did not seem to improve the running time, but forcing Spark to use sort merge join reduced the running time by 17 %.

Table 7: Average running times for each optimization of the main script

Optimization	Duration
Baseline - 1 GiB	37 min
Baseline - 6 GiB	35 min
Cache - 6 GiB	35 min
Sort Merge Join - 6 GiB	29 min
UDF - 6 GiB	1.9 h

Figure 6 shows the timeline of the longest-running stage with the baseline configuration. The first 12 tasks are evenly distributed, but one task is running after all the others causing almost all executors to sit idle for about 5 min. Figure 7 shows the timeline for the longest-running stage when forcing Spark to use sort merge joins. Here, the tasks are evenly distributed across all executors. Most of the time is spent on execution, but it is also evident that there is more shuffling involved compared to Figure 6. Since broadcast hash join is only supported for equi-joins, Spark decided to use broadcast nested loop join because no join conditions were specified for the joins in this stage. Hence a cartesian product join was used instead to join the datasets.

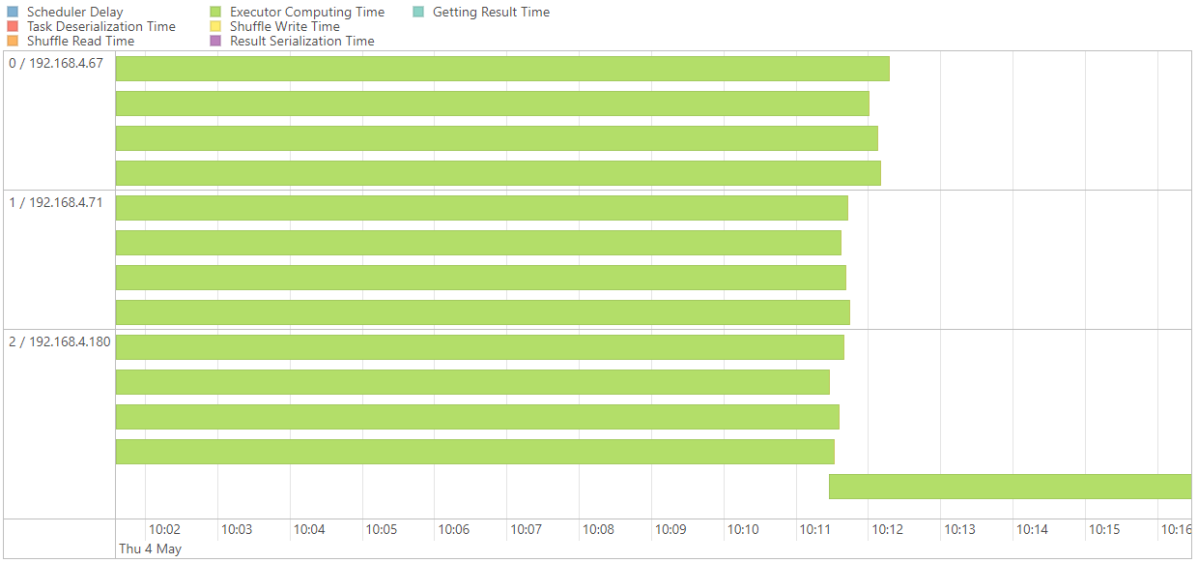

Figure 6: Timeline of the longest running tasks with baseline configuration and 6 GiB memory for each executor

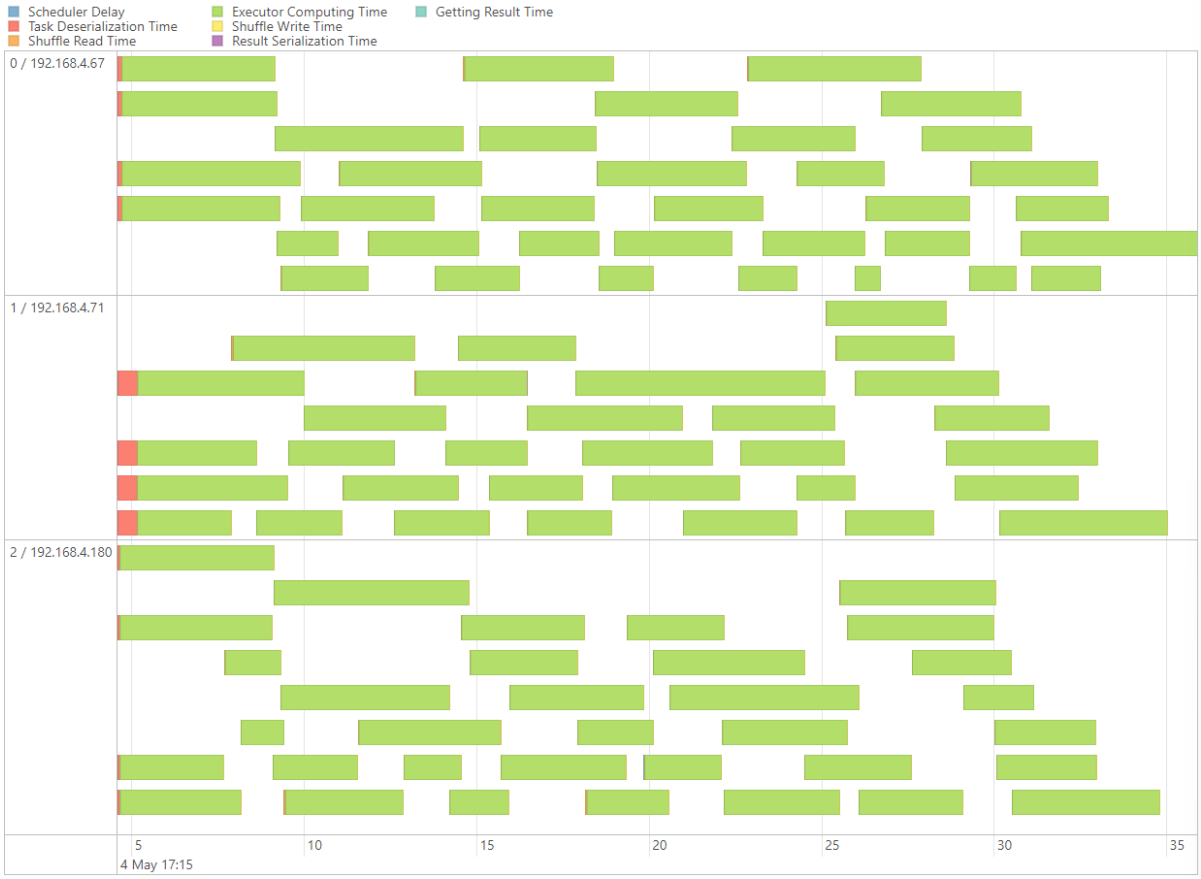

Figure 7: Timeline of the longest running tasks with sort merge join as default join and 6 GiB memory for each executor

3.3 Discussion

The higher JVM memory usage and lower GC time shown in Table 1 indicates that Spark can process garbage less frequently when more memory is available, thus increasing performance. This combined with the reduction in spill shown in Table 2 are the primary benefits that were observed from increasing the executor memory.

Caching of dataframes seems to have little effect on performance for this workload. As can be seen in Table 3, the benefit of reduced GC time is diminished by other tasks, such as the increased amount of data read from storage.

It is evident from Table 4, 7, 5 and 6 that the biggest improvement in execution time was by removing UDFs and thus reducing serialization.

Spark created 13 tasks for the longest-running stage, which is related to the transformations used for searching the news articles for stocks and sectors, as mentioned in section 3. This can be seen in Figure 6. All executors sit idle while the last task is executed, indicating that the partitioning of the dataset is not optimal. Figure 7 shows that a larger number of smaller partitions are more evenly distributed across all executors for the whole running time of the stage, meaning that the cores on the executors sit idle for a shorter time resulting in better overall performance.

The sort merge join should perform worse due to a larger amount of shuffling required, but the distribution of tasks is more evenly split between the executors as can be seen in Figure 7. Table 6 also shows that there is a significantly larger amount of shuffle write compared to when broadcast joins were used. Due to how the stage is partitioned in the sort merge join, the overall time is reduced, which can be seen in Table 7. However, forcing Spark not to use broadcast joins is not an optimal solution because several joins could benefit more from a broadcast hash join, such as the joins between the smaller stocks dataset and the larger news dataset.

4 CONCLUSION

Optimizing the execution times of the pipeline created in Apache Spark proved to not be an easy task. Many optimizations are done by default by Spark, but 4 optimizations were chosen to further improve the performance of the pipeline. Increasing the memory for each executor resulted in less GC and mitigated spill. It also made it possible to cache dataframes and let Spark do more memory-intensive improvements. However, caching frequently used dataframes did not yield a decrease in execution times due to the increased amount of data read from storage. The biggest reduction in execution time was due to the removal of UDFs and replacing them with transformations leveraging the Spark API. Lastly, join strategies can impact performance by a significant amount.

Future work could establish if using different join strategies would improve the pipeline even further. Bucketing could be tested in conjunction with sort merge joins. Also, each transformation could be looked at carefully and improved even further by trying to reduce the number of wide transformations.

REFERENCES

[1] Jules Damji, Brooke Wenig, Tathagata Das, and Denny Lee. 2020. *Learning Spark* (2 ed.). O'Reilly Media, Sebastopol, CA.

[2] Apache Software Foundation. 2023. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html

[3] Preeti Gupta, Arun Sharma, and Rajni Jindal. 2018. An Approach for Optimizing the Performance for Apache Spark Applications. In *2018 4th International Conference on Computing Communication and Automation (ICCCA)*. IEEE, Greater Noida, India, 1–4. https://doi.org/10.1109/CCAA.2018.8777541

[4] Rebecca Lake. 2020. A Guide to the 11 Market Sectors. https://finance.yahoo.com/news/guide-11-market-sectors-142851510.html

[5] Randall T. Whitman, Michael B. Park, Bryan G. Marsh, and Erik G. Hoel. [n. d.]. Spatio-Temporal Join on Apache Spark. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2017-11-07) *(SIGSPATIAL '17)*. Association for Computing Machinery, 1–10. https://doi.org/10.1145/3139958.3139963