Cookie Data Analysis and Optimization in PySpark

Github:https://github.com/olufisayo-akindele/Spark-optimization

Marius Skårdal and Olufisayo Togun
University of Stavanger, Norway
oa.togun@stud.uis.no,m.skardal@stud.uis.no

ABSTRACT

As data continue to be generated at an exponential rate, the importance of analyzing big data has become important for businesses and entrepreneurs. With the volume of data being processed big data analysis can be time consuming making it essential to optimize the performance of data processing. Optimization techniques can improve the processing time significantly and improve the overall performance, enabling businesses to analyze large datasets effectively and efficiently.

This project explores optimization techniques for improving the performance of big data processing using Apache Spark. Specifically, we investigate the impact of caching, partitioning, and compression techniques on the efficiency of Spark jobs. We also examine the role of syntax optimization and vectorization in improving the processing speed of large datasets. Our experiments show that caching is an effective optimization technique in Spark, particularly when working with large datasets. We also find that compression techniques can significantly improve the performance of Spark jobs by reducing network transfer time.

Additionally, syntax optimization and vectorization can further enhance the processing speed of Spark jobs by minimizing redundant computations and leveraging specialized data structures. [1] Our findings demonstrate that applying a combination of these optimization techniques can lead to significant improvements in the performance and efficiency of Spark jobs. These results can help inform best practices for optimizing big data processing using Apache Spark.

1 INTRODUCTION

In this project we will be focusing on implementation and optimization when considering big data analysis, our goal is to run a large data set efficiently through optimization algorithms such as caching, partitioning, shuffle compression, syntax optimization, and vectorization. As we dwelled through, we were testing out various optimization techniques to improve performance on our set of data which explains the number of algorithms used or attempted. Our use case consisted of extracting information from cookie data to make a model that can recommend products for certain age groups.

Caching was chosen to optimize computational processes and is a technique used for storing the results of the computation in memory, which can help boost performance as Spark computes iteratively. Caching helps improve performance in two ways. First, it helps minimize network traffic which can indirectly cut down on shuffle run time. This means that subsequent computations can be performed on the cached RDD without having to load from disk, and data can be shared across the nodes in the cluster as seen in Figure 1.

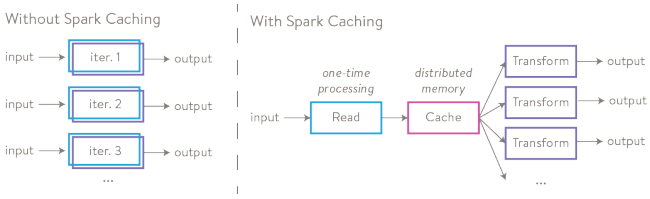

Figure 1: Caching in PySpark

Secondly, caching can improve performance by reducing computation as data is stored in memory, it can be accessed for further computations. By caching our aggregated results, we can avoid doing redundant computations and reuse these results which we expect to save time on. [2]

In a Pyspark cluster, the data processing tasks are distributed among the nodes which divides the input data into partitions and assigns each partition to a worker node for processing. The problem is that some worker nodes may finish their tasks faster than others which leads to some nodes being idle. [3] What this means is that before these nodes can perform subsequent tasks they need to wait for the other node to finish, this is the key aspect of why we decided to test out partitioning.

Partitioning helps to avoid the problem described, by ensuring that each node has roughly the same amount of data to process by dividing the input data into a smaller number of partitions and then assigning them to the nodes which helps to balance the workload across the nodes. Which we expect to result in faster processing times.

We decided to try out shuffle compression, which is a configuration parameter that controls whether Spark should compress the shuffling over a network. When shuffle compress is enabled, Spark compresses the data that needs to be transmitted over the network during shuffling, which helps alleviate network congestion. The drawback is that the method may increase processing overhead on the nodes since the data needs to be compressed and decompressed.

Another optimization method we tried was to optimize our syntax, this is to try and optimize our performance even more but is more of a time-consuming method if we are going to improve

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

every function we are using. We decided to test out for a few functions where we used a basic lambda function instead of a sum and approx_count _distinct instead of CountDistinct. [4]

Lambda functions are typically more efficient, flexible, and easier to understand than doing a full-on aggregation, the reason for this is since data is distributed across multiple nodes in the cluster lambda functions are typically smaller in size making them easier to serialize. Another benefit of using a lambda function is the lazy evaluation in PySpark, meaning that data is not processed until an action is triggered, Lambda functions are processed in the same way, but they are only executed when needed.

approx_count _distinct is a more efficient function than CountDistinct in Spark because it approximates the number of distinct values with a high degree of accuracy by only processing a small sample of the data, making it more effective on a large dataset. approx_count _distinct also uses less memory as a result in comparison with CountDistinct which has to process the entire dataset to count the number of unique values.

The last optimization technique we implemented was vectorization, which in Spark involves processing data by operating on entire arrays or vectors at once. We expect this to improve performance with the machine learning algorithms that we have, as we do not have to use element by element in the array. By converting our data frame into a single array we can take advantage of vectorized operations to process the data faster, many of Spark's MLlib functions also supports vectorization making it so we expect it to enhance the performance of our models. [5] The following figure displays how vectorization reads a column of strings.

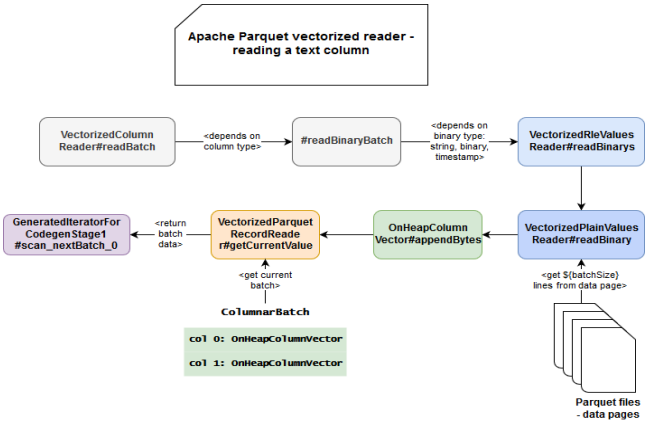

Figure 2: Vectorization in PySpark (source: waitingforcode)

In the figure we see that vectorization accumulates multiple inputs into a single batch and processes them all at once. [6]

2 METHODOLOGY

2.1 MapReduce in Hadoop

Hadoop MapReduce is a distributed data processing framework used for big data analysis and is designed to handle large volumes across a network of computers. [7] The framework consists of two primary steps, in which we input our data which is typically unstructured, and use a mapper that applies a transformation of inputs to input pairs, producing a set of key-value pairs as output. These key-value pairs are then partitioned so that the key values will know which reducer they will go to. In our specific case, we only used a mapper as we had no interest in aggregating the data. After the partitioning step, the key values are written to HDFS to be stored, as the dataset contained indices our code made sure to move them below the header, so our dataset would be compatible with a CSV format. Below is a figure that displays the pipeline our dataset went through.

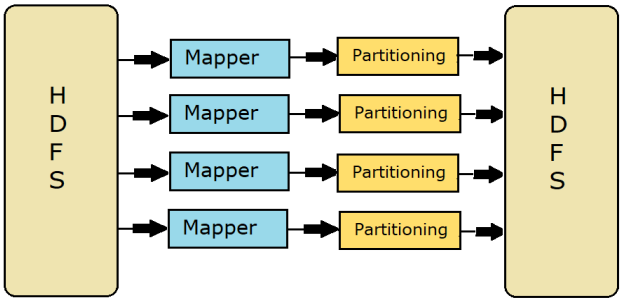

Figure 3: Hadoop pipeline for our data

2.2 Design

Two datasets which were the output of our MapReduce were used, 1 large dataset of about 8.5 GB, 13 columns, and 58 million rows and a small dataset with 5.7mb, 3 columns, and 270001 rows.

The spark session was defined, 6 executors were used, and delta table extensions were added to the configuration. Two sets of CSV files were loaded into two data frames, schemas, and headers were defined for the data frames being loaded. The two data frames were combined (joined_df) using a join function and stored on HDFS using the delta table format.

All rows with more than 40% null values were dropped and two columns "request _cnt" and "is_male" had the portion of the string" t" removed and converted to integers. The pre-processed ("filtered _df") data frame was saved to HDFS using the delta format.

The column" distinct_models _count" was dropped from the combined data frame and a new column was added named "age _bin" which was used to group age range into bins(0-17,18-29,30-49 and 50+). A function(filter_by _optimized) was developed to extract the top 3 phone models and model manufacturer for each age bin.

A machine learning model was developed to predict the age of all distinct user _ids. A linear regression model was used and 75% of the distinct user _id data frame (ml _df) was used as training data while the remaining 25% was used as test data.

Part 1:

Step 1: The columns in the predicted age table were all renamed with the string " _r" added to their previous names. The recommended product table(overall_market _need) was merged with the predicted age table(predicted _age).Note: at this point of the code the tables grow three times larger because we have three products recommended for each user.

Step 2: Two new columns were created called "combined _col1" and "combined _col2" which were a concatenation of "user_id", "cpe _manufacturer _name", "cpe _model _name" and "user _id _r", "cpe _manufacturer _name _r", "cpe _model _name _r" for "combined _col1" and "combined _col2" respectively.

All rows that had similar machine values for any of the columns combined _col1 and combined _col2 were dropped. note: This step was done in order to avoid recommending the same product that the user uses to login to the user.

Step 3: All duplicate rows were with the dropped with regards to the column "user _id".

Part 2:

The above steps in part 1 were repeated for the initial preprocessed data frame but in place using "user _id" for steps 2 and 3 in part 1 index was used. This was done for 2 reasons to recommend products for each login and to test out the algorithm in part 1 on actual big data.

The resulting table from part 1 and part 2 were saved to HDFS using the delta table format. Key Note: It is worth mentioning that in our code design, we did not implement the delta table merge but basically if we were to do that we would have reuploaded our saved that in the earlier steps and worked with that and do a delta merge if the final tables and not been created and if they have not been created a new table would be made. The delta table has transactions logs stored along with the data which can help prevent defining your data and repeating completed steps.

2.3 Optimization & implementation

Optimizing is a key part of working with big data and improving the performance of our code, when dealing with big data a variety of performance issues can appear, such as slow processing times, increased memory usage and bottlenecks in data transfer. . In our case, we noticed that two specific issues were slowing down our code such as shuffling and task deserialization as seen in Figure 4 and 5.

Shuffle runtime refers to transferring data between nodes in a distributed computing environment, in our case we were transferring volumes of data through the network by aggregation processes which significantly slows down our code. Task deserialization refers to converting data from a serialized format into a format that is understood by Spark, Since our data was serialized by using a CSV format it caused a bottleneck in our code's performance.

Figure 4: Unoptimized DAG

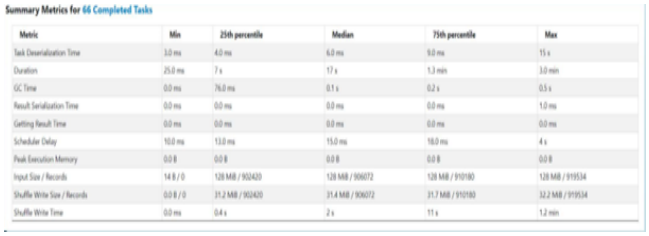

Figure 5: Unoptimized Summary Metrics

2.4 Caching

When we implemented caching into our code, it is simple to implement as you can add .cache() to any data frame or array that you wish to store. The drawback of caching is that it is easy to fill the memory up causing an OOM or "Out of Memory Error", meaning that we have to clear the cache as we run our code. [8] The way we did this was after every significant transformation of data, such as an aggregation we used the .unpersist() function to clear the cache for that specific data frame. The challenge linked with caching is that it is often difficult to know exactly how to optimize the method itself, like where caching is needed and not needed. Below you can see a figure of how we cached our data. Caching was implemented on all data frames in our code with the exception of the last data frame which was significantly large to cause an out-of-memory error as the size of the data frame will more than double during the transformations occurring in the final step. [8]

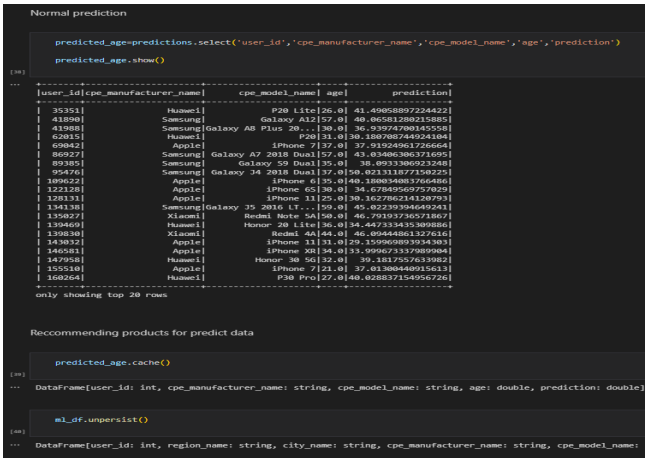

Figure 6: Implementation of caching

As showcased in the figure above, after each transformation we cache the new data frame and unpersist the previous which helps for later computation.

2.5 Partitioning

Implementing partitioning in PySpark is simple as we used the .repartition() command, the command rearranges the number of partitions used. The drawback of using repartitioning is that it is difficult to determine the quantity of partitions to use, as we need to test out and figure what works best. The number of partitions used affects the parallelism of the operations meaning processing the data across the different nodes in the cluster. This can impact the performance of computing, where too few partitions could result in inefficiency and skewness, and too many partitions can increase the overhead leading to longer processing times. The partition was implemented on the largest data frame initial load from HDFS at the start of the spark code. [9]

Figure 7: Implementation of partitioning

Figure 8: Amount of partitions

As seen in the figure above, we tested out for 100 partitions and other sizes as well, to see how it would affect the overall performance of the PySpark job.

2.6 Shuffle Compress

Shuffle compress is an optimization technique we implemented in the Spark configuration, the technique is designed to reduce the size of the data transferred between nodes and this technique is supposedly turned off by default, we checked the spark configuration by printing out whether it was turned on or off, but it turned out to be off. As a result, we decided to include the shuffle compression as an optimization technique and the implementation is shown below.

Figure 9: Shuffle Compress Configuration

Figure 10: Shuffle Compress turned off

After enabling shuffle compress in the config, we got the following print statement:

Figure 11: Shuffle Compress turned on

2.7 Syntax Optimization

For the syntax optimization we chose to implement a few functions, such as using lambda functions instead of a traditional sum and approx_count _distinct instead of CountDistinct to see how they would affect performance. The implementation of the code is shown below.

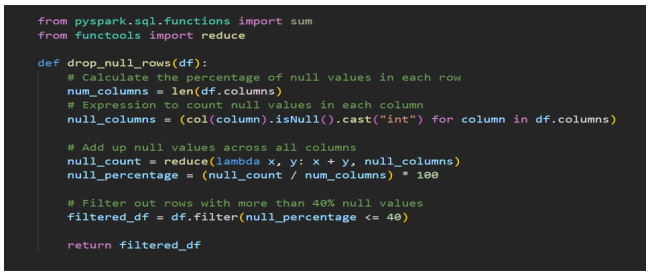

Figure 12: Implementation of lambda function

```
filtered_df.groupBy('user_id').agg(approx_count_distinct('cpe_model_name').alias('distinct_models_count')).orderBy('distinct_models_count', ascending=False).show()
```

Figure 13: Implementation of approx _count _distinct

The drawback of syntax optimization is the time required to optimize the functions that we are using, as it can take a while to optimize every function used in the document, which is why we only tried out a few.

2.8 Vectorization

Applying an operation to the entire set of values all at once is possible with vectorization. By gathering numerous inputs into a single batch and processing them all at once, performance can be improved. Vectorization in Spark is associated with column-oriented data sources like Apache Parquet, Apache ORC, or Apache Arrow. In this case study vectorization was implemented user defined function. The challenge with using vectorization is that it does not allow Spark to optimize its operation when the vectorized function is being executed. [10]

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

@udf(returnType=StringType())
def remove_tab(s):
    return s.replace("\t", "")

filtered_df = filtered_df.withColumn("request_cnt", remove_tab(col("request_cnt")))
filtered_df = filtered_df.withColumn("is_male", remove_tab(col("is_male")))
filtered_df = filtered_df.withColumn("request_cnt", col("request_cnt").cast("integer"))
filtered_df = filtered_df.withColumn("is_male", col("is_male").cast("integer"))


from pyspark.sql.functions import regexp_replace, col

# remove unwanted \t in the DataFrame
filtered_df = filtered_df.withColumn("request_cnt", regexp_replace(col("request_cnt"), "\t", ""))
filtered_df = filtered_df.withColumn("is_male", regexp_replace(col("is_male"), "\t", ""))
filtered_df = filtered_df.withColumn("request_cnt", col("request_cnt").cast("integer"))
filtered_df = filtered_df.withColumn("is_male", col("is_male").cast("integer"))
```

Figure 14: Implementation of Vectorization

3 RESULTS

Here are the following results for each of the optimization methods used, in the first figure we can see the optimized DAG for caching.

Figure 15: Optimized DAG for Caching

Figure 16: Optimized DAG for Partitioning

Test	Execution time
1	4281.82
2	4260.49
3	4278.24

Table 1: Execution times for Partitioning

Test	Execution time
1	4382.92
2	4737.16
3	4588.45

Table 2: Execution times for Shuffle-compress

Test	approx _count _distinct Execution time	CountDistinct Execution time
1	111.38	160.21
2	115.26	154.38
3	109.79	156.71

Table 3: Execution times for approx _count _distinct vs CountDistinct

Test	Execution time (Vectorization)	Execution time (Spark built-in)
1	144	145
2	157	141
3	255	64

Table 4: Execution times for Vectorization vs Spark

4 DISCUSSION

Upon running the code without any optimisation technique implemented it is observed that the code crashes due to excessive shuffling and memory spill, it was also observed that there were irregular partitions at the data nodes when certain commands were executed and our input basically was a large data frame of about 8 GB. For these reasons, we decided to implement 4 optimization techniques which include: shuffle-compress, caching, partition and

Test	Execution time
1	1070.15
2	1075.49
3	1078.24

Table 5: Execution times for combined best optimiztion (caching & shuffle-compress)

vectorization. After implementing caching into our Spark code, we observed a significant improvement in run time with the whole code being executed on an average of 1220 seconds. It was observed that most portions of the code that had bottlenecks with shuffle read and write quantities were reduced. The machine learning portion of the code ran about 4 times faster with a heavy reduction in shuffle activity. There was also no spill observed when caching was implemented.

The default partition size for Spark is 200, but When spark partitions our data frame after it has been loaded we noticed that Spark was running on 66 partitions despite 200 being the default value, we believe Spark had optimized repartitioning. [5] So when we were trying out different partitions there was not much help in changing the values, as the runtime decreased. By looking at the figure below you can see how the code ran for 100 partitions, we also tried experimenting with 132 and 200 partitions but we only had longer executions time. The reason why partition was a bad optimization in our case was that it increased the amount of shuffling that occurred between the data nodes, this also in turn significantly increased the serialization and deserialization time when different jobs were executed. Partition was not significantly useful for us at it increased our run time. The code used to print out the number of partitions used is included here.

As seen in the results, the repartitioning did not enhance performance, but instead slowed down the code, using 100 partitions took us the most time with 4281.82 seconds.

Due to the fact that we had a large data frame, we decided to implement vectorisation, overall it was noticed that vectorisation did not significantly improve the run time, it performed at par with the non-vectorised portion of the code, the operation for which vectorisation was used was not complex and if we had more complex operations on which we could use vectorization we might observe significant improve meant in the execution time of the code. It was observed that one of the execution times of the vectorization optimization was significantly high than the others, this is probably due to the instability of the cluster.

Shuffle-compress is a configuration setting that was used as an optimization technique, the shuffle-compress basically compresses files while they are shuffled using a default codec snappy, the code ran after this optimization was implemented without crashing but the average execution time for the optimization was about 4500 seconds as seen in Table 2. Shuffle compress did not stop excessive shuffling from occurring but it reduced the amount of memory spill that occurred as a result of the excessive shuffling. It is worth mentioning that shuffle compress enabled our code to run faster but did not perform as optimally as caching.

we got the best results when we combined the caching and shuffle compress, as seen in Table 5 and the reason for this is that caching saves us time, prevents excessive shuffling and minimizes memory access while shuffle compress, compresses data being sent across nodes. The average time for the combination of these optimizations was noticed to be about 1070 seconds as seen in Table 6.

Some extract optimization techniques to mention are syntax optimizations. Since approx _count _distinct approximates based on a sample of the data and does not compute for the entire data frame this method was expected to be faster than using CountDistinct. we replaced the sum function with the lambda addition function and this also significantly improved our code run time, we believe the reason why lambda worked faster was because the operation we used for a small complex operation, in this case using the regular sum was less efficient than the lambda function. When running the code we saved about 40 seconds each on both optimizations this is not much but for a larger set of data, it will improve the performance. Table 3 shows how the run time of a portion of the code improved when approx _count _distinct was used in place of countDistinct.

5 CONCLUSION

In this project, we aimed to optimize the performance and computation of cookie data analysis using Spark, Our primary objective when discovering the challenges in our DAG was to reduce the shuffle write time and task deserialization, which were identified as the performance bottlenecks in our pipeline.

We started optimizing by exploring caching, which stores RDD's in memory, reducing the number of times it needs to be computed which reduces the overall processing time. We found that caching was effective in reducing shuffle write time because it reduces the amount of data that needs to be shuffled across the network.

We also tried partitioning, which involves dividing data into smaller chunks, and by partitioning RDD's, the intention was to reduce the amount of shuffling across the network and improve the efficiency of parallelism. But as discussed previously, Spark was already using 66 partitions and it was working better than the other tests we ran, so we decided to test out shuffle compression, which involves compressing the data shuffled across the network to reduce the amount of network traffic and improve performance.

To further optimize our code, we looked into syntax optimization and vectorization. Syntax optimization involves writing more efficient code by avoiding unnecessary operations and using efficient syntax, while vectorization involves converting data into numerical representations to speed up calculations. We found that syntax optimization and vectorization improved overall processing time by reducing task deserialization and reducing the amount of

data processed.

To evaluate how effective these optimization techniques were, we ran several tests to get the runtime of the optimizations individually and compiled together. The results showed that a combination of these techniques was effective on our dataset as it was of significant size. By using a combination of optimization techniques, we could significantly reduce processing time and improve overall pipeline performance.

In conclusion, our analysis shows how various optimization techniques are specific on the task you are working with. Learning to use many different optimization techniques will prove useful in future projects as we are more aware on the applications of the individual optimization techniques. By identifying underlying issues and knowing what type of data we are dealing with, we can improve performance on both small and large datasets.

6 APPENDIX

REFERENCES

[1] Muhammad Asif Abbasi. *Learning apache spark 2.* Packt Publishing Ltd, 2017.
[2] Michael J Mior and Kenneth Salem. Respark: Automatic caching for iterative applications in apache spark. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 331–340. IEEE, 2020.
[3] Bill Chambers and Matei Zaharia. *Spark: The definitive guide: Big data processing made simple.* " O'Reilly Media, Inc.", 2018.
[4] BigDataThoughts. Spark performance optimization part1 | how to do performance optimization in spark. https://www.youtube.com/watch?v=NvziNVQSlqk&t=776s&ab_channel=BigDataThoughts, 2016.
[5] Apache Spark. Sql performance tuning. https://spark.apache.org/docs/latest/sql-performance-tuning.html, 2021. [Online; accessed 4-May-2023].
[6] Rafał Leszko. Vectorized operations in apache spark sql. https://www.waitingforcode.com/apache-spark-sql/vectorized-operations-apache-spark-sql/read, 2019. Accessed on May 3, 2023.
[7] Vijayakumar Subramaniyaswamy, V Vijayakumar, R Logesh, and V Indragandhi. Unstructured data analysis on big data using map reduce. *Procedia Computer Science*, 50:456–465, 2015.
[8] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. Hpcache: Memory-efficient olap through proportional caching. In *Data Management on New Hardware*, pages 1–9. 2022.
[9] Romeo Kienzler. *Mastering Apache Spark 2. x.* Packt Publishing Ltd, 2017.
[10] Holden Karau and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark.* " O'Reilly Media, Inc.", 2017.