e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:05/Issue:10/October-2023 Impact Factor- 7.868 www.irjmets.com

OPTIMIZE LARGE-SCALE DATA PROCESSING VIA SPARK TUNING

Apurva Kumar*1, Shilpa Priyadarshini*2

*1Principal Software Engineer, Walmart Labs, Sunnyvale, CA, USA.

*2Principal Product Manager, Silicon Valley Bank, Santa Clara, CA, USA.

DOI : https://www.doi.org/10.56726/IRJMETS45567

ABSTRACT

Apache Spark is a leading open-source data processing engine used for batch processing, machine learning, stream processing, and large-scale SQL (structured query language). It has been designed to make big data processing quicker and easier. Since its inception, Spark has gained huge popularity as a big data processing framework and is extensively used by different industries and businesses that are dealing with large volumes of data. This paper will exhibit actionable solutions to maximize our chances of reducing computation time by optimizing Spark jobs. The strategy lays out different run stages, wherein each run stage builds upon the previous and improves the computation time by making new enhancements and recommendations.

Keywords: Spark, Big Data, Batch Processing, Spark Optimization.

I. INTRODUCTION

Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Scala, Java, Python, and R, and an optimized engine that supports general computation graphs for data analysis. Spark is designed to be fast, scalable, and fault-tolerant, and it can be used to process both batch and streaming data. One of the key features of Spark is its in-memory processing capability. Spark can store data in memory across a cluster of machines, which allows it to perform computations much faster than traditional disk-based systems. Spark also provides a variety of fault tolerance mechanisms, so that data processing can continue even if some of the machines in the cluster fail.

Spark can be used for a wide range of large-scale data processing tasks, including:

- Batch processing: process large datasets in batch mode, such as for data warehousing and machine learning.
- Streaming processing: process streaming data, such as sensor data and social media feeds.
- Interactive data analysis: perform interactive data analysis on large datasets, such as for ad-hoc queries and data visualization.

This paper focuses on how we can leverage different optimization techniques in order to reduce the computation time it takes to run the job end to end.

II. METHODOLOGY

The strategy lays out different run stages, wherein each run stage builds upon the previous, and improves the computation time by making new enhancements and recommendations. The underlying dataset here is based on flight data [1]. The challenge was to perform this computation on extremely high data volumes with better performance, and at a lower cost. The recommendations made here are with YARN & HDFS cluster but can be applied to other infrastructures as well. Six run stages of spark optimization. We start with an overview depicting the various run stages and the improved run times in each optimization technique. The first five runs took place on Spark 2.4 and the last one was implemented on Spark 3.1.4.

III. MODELING AND ANALYSIS

1. Optimization: Serialization, Parquet File Format, and Broadcasting

1.1 Serialization

Serialization helps in converting objects into streams of bytes and vice versa. When we work on any type of computation, our data gets converted into bytes and are transferred over the network. If we transfer less data across the network, the time taken for the job to be executed decreases accordingly. Spark provides two types of Serializations, Java and Kryo.

1.1.1 Java Serialization

- Provided by default, which can work with any class that extends java.io.Serializable

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:05/Issue:10/October-2023 Impact Factor- 7.868 www.irjmets.com

- Flexible but quite slow, and leads to large, serialized formats for many classes

1.1.2 Kryo Serialization

- Faster and compact compared to Java Serialization
- Requires registering the classes in advance for best performance

1.2 Parquet file format

Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

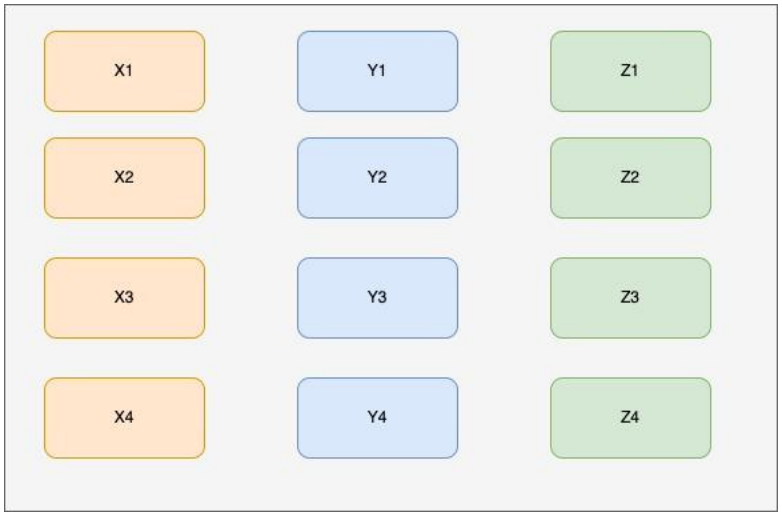

Figure 1: A sample dataset in a tabular format

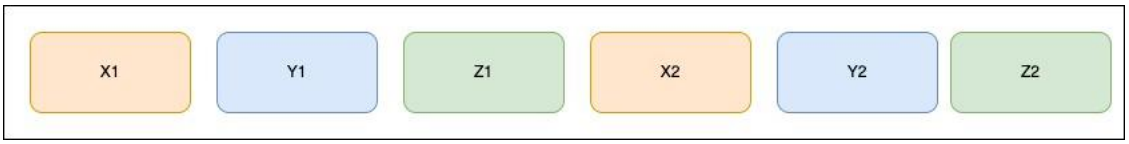

Figure 2: Row-oriented storage format saves the data row-wise

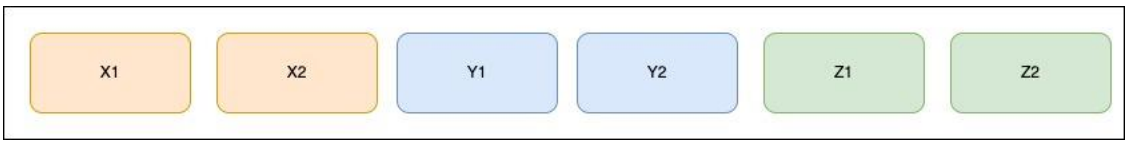

Figure 3: Column-oriented storage format saves the data column-wise

Columnar formats are attractive in terms of both file size and query performance. File sizes are usually smaller than row-oriented equivalents as the values from one column are stored next to each other. Additionally, query performance is better as a query engine can skip columns that are not needed.

1.3 Broadcasting

Joining two tables is a routine operation in Spark. Usually, a large amount of data is exchanged over the network between the executing nodes. This exchange can cause network latency. Spark offers several join strategies to optimize this operation. One of them is Broadcast Hash Join. If one of the tables is small enough (the default is 10MB, but could go up to 40MB), the smaller table can be broadcasted to each executor in the cluster, and shuffle operations can be avoided. Broadcast Hash Join happens in 2 phases, Broadcast, and Hash Join.

- Broadcast phase: Small dataset is broadcasted to all executors
- Hash Join phase: Small dataset is hashed in all the executors and joined with the partitioned big dataset

Here are some things to note about Broadcasting:

1. The broadcast relation should completely fit into the memory of each executor as well as in the driver, because the latter starts the data transfer.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:05/Issue:10/October-2023 Impact Factor- 7.868 www.irjmets.com

2. When the size of the broadcasted data is big, you would get Out Of Memory exception.
3. Broadcasting only works for equi ('=') joins.
4. Broadcasting works for all join types (inner, left, right) except full outer joins.
5. Spark deploys this join strategy when the size of one of the join relations is less than the threshold values (10 MB default). The Spark property which defines this threshold is spark. sql. auto Broadcast Join Threshold (configurable).

2. Optimization 2: Breaking the Lineage

In our use case, we have complex computations which involve iterative and recursive algorithms that are performed on large volumes of data. Each time we apply transformations on a DataFrame, the query plan grows. When this query plan becomes huge, the performance decreases dramatically, which results in bottlenecks. It is not advisable to chain a lot of transformations in a lineage, especially when you need to process a huge volume of data with minimum resources. We tested the following three methods to break the lineage:

2.1 Checkpoint

Checkpointing is a process of truncating the execution plan and saving it to a reliable distributed (HDFS) or local file system. It's a feature of Spark to that is specifically useful for highly iterative data algorithms. Checkpoint files can be used in subsequent job runs or driver programs. Checkpointing can be eager or lazy, as per the eager flag of checkpoint operator. The former is the default and happens immediately when requested, while the latter only happens when an action is executed.

2.2 Local Checkpoint

This works similarly to a checkpoint, but the data is not transferred to HDFS, and is instead saved to the executor's local filesystem. If an executor is killed during processing, the data will be lost, and Spark will not be able to recreate this Data Frame from the DAG (Directed Acyclic Graph).

2.3 Writing data to HDFS in parquet

When we checkpoint the RDD/Data Frame, it is serialized and stored in HDFS. It doesn't store it in parquet format, parquet provides efficient data storage. Breaking the lineage by writing to HDFS in parquet gave us the best performance of the above three. It would also be good to note that caching offers an alternative to increase performance without breaking the lineage.

3. Optimization: Right Shuffle Partition

Choosing the right shuffle partition number helps in job performance. Partitioning decides the degree of parallelism in a job, as there is a one-to-one correlation between a task and a partition (each task processes one partition)[4].

The ideal size of each partition is around 100–200MB. The smaller partitions increase the number of parallel running jobs, which can improve performance, but too small of a partition will result in overhead and increase the GC time. Larger partitions will decrease the number of jobs running in parallel and will also leave some cores idle, which will increase the processing time. In case of a shuffle, Spark recommends [2] below:

1. If intermediate data is too large, then we should increase shuffle partitions to make partitions smaller.
2. In case of idle cores during job runs, increasing shuffle partitions helps in job performance.
3. If intermediate partitions are small (in KBs), then decreasing shuffle partitions helps
4. For a cluster with huge capacity, the number of partitions should be 1x to 4x of the number of cores to gain optimized performance. For instance, with a data of 40GB and 200 cores, set the shuffle partition to 200 or 400.
5. For a cluster with limited capacity, shuffle partitions can be set to Input Data Size / Partition Size (100–200MB per partition). The best-case scenario would be to set the shuffle partition to be a multiple of the number of cores to achieve maximum parallelism, depending on cluster capacity.

4. Optimization: Code

For this run we focused on making two changes at the code level.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:05/Issue:10/October-2023 Impact Factor- 7.868 www.irjmets.com

4.1. Replace join and aggregations with Window functions

In most of our computations, we had to perform aggregation on specified columns. The result was to be stored as a new column. In this instance, this operation consists of aggregation followed by join. The more optimized option here would be to use Window functions. By replacing our join and aggregation with Window functions in our code, we found significant improvement.

4.2. Replace with Column with Select

Every operation on a Data Frame results in a new Data Frame. In cases when we need to call with Column repeatedly, it's better to have a single Data Frame. So, instead of using:

```
df.withColumn( colName = "amount", col( colName = "amount").cast(DecimalType(28, 8)))
df.withColumn( colName = "weighted_amount",col( colName = "amount")* .2)
df.withColumn( colName = "weighted_neg",col( colName = "weighted_amount")* -1)
```

use:

```
df.select(col( colName = "amount").cast(DecimalType(28, 8)).as( alias = "amount"),
  (col( colName = "amount") * .2).alias( alias = "weighted_amount"), (col( colName = "weighted_amount") * -1)
    .alias( alias = "weighted_neg"))
```

5. Optimization: Speculative Execution

Apache Spark has speculative execution feature to handle the slow tasks in a stage due to environment issues such as a slow network. If one task is running slowly in a stage, the Spark driver can launch a speculation task for it on a different host. Between the regular task and its speculation task, the Spark system will take the result from the first successfully completed task and kill the slower one. In the case of long running jobs (where some tasks are slower than the others) — which can be identified from monitoring the time taken via Spark UI, enabling speculation would help. If spark. speculation is set to true, then slow running tasks are identified based on the median computed by taking the completion time of other tasks. After identifying the slow running jobs, speculative tasks are initiated on the other nodes to complete the job.

6. Optimization: Enabling AQE (Adaptive Query execution)

For this run, we enabled AQE, which is the main feature of Spark 3.0. AQE can be enabled by setting SQL config spark. sql. adaptive. enabled to true (false is the default in Spark 3.0). In Spark 3.0, the AQE framework is shipped with three features:

6.1 Dynamically coalescing shuffle partitions

It simplifies or even avoids tuning the number of shuffle partitions. Users can set a relatively large number of shuffle partitions at the beginning, and AQE can then combine adjacent small partitions into larger ones at runtime.

6.2 Dynamically switching join strategies

It partially avoids executing suboptimal plans due to missing statistics and/or size misestimation. This adaptive optimization automatically converts sort-merge join to broadcast-hash join at runtime, further simplifying tuning and improving performance.

6.3 Dynamically optimizing skew joins

It is another critical performance enhancement, as skew joins can lead to an extreme imbalance of work and severely downgrading performance. After AQE detects any skew from the shuffle file statistics, it can split the skew partitions into smaller ones and join them with the corresponding partitions from the other side.

IV. RESULTS AND DISCUSSION

As we can see the total time it took post each optimization technique helped improve the runtime from 2.5 hours without any optimization to roughly 35 minutes leading to ~78% improvements. Hence using spark tuning we can do large-scale data processing much faster without scaling the underlying compute infrastructure.

e-ISSN: 2582-5208
International Research Journal of Modernization in Engineering Technology and Science
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
Volume:05/Issue:10/October-2023 Impact Factor- 7.868 www.irjmets.com

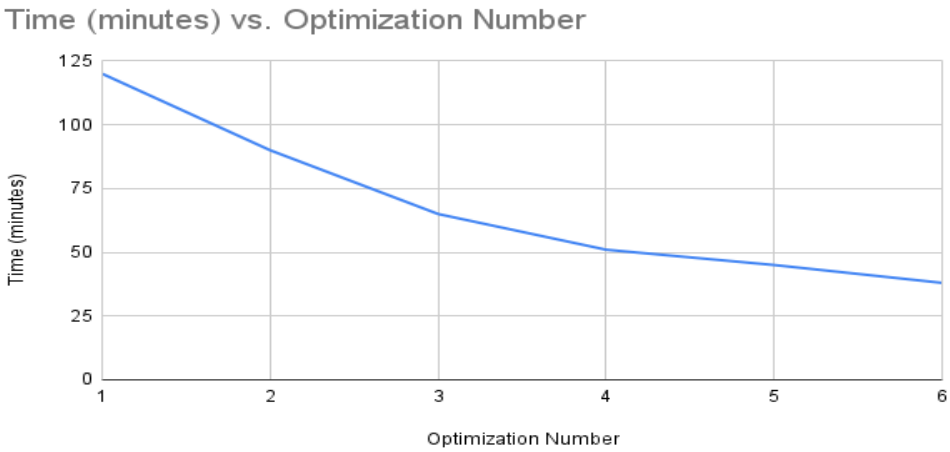

Figure 4. Time taken to run the job vs each optimization step 1-6 discussed in Section III

V. CONCLUSION

Spark tuning is critical in order to achieve maximum performance for a running job. This work deals with configuring Spark applications in an efficient manner. By tuning Spark applications, we achieved significant performance improvements (~78%) without scaling the underlying hardware. We evaluated the effectiveness of our proposal by dividing into 6 stages of optimizations and also provided the detailed principles behind our approach. Our results showcased that with an informed approach an existing large-scale workload can be computed in significantly less time using proper tuning of spark parameters.

VI. REFERENCES

[1] https://openflights.org/

[2] https://spark.apache.org/docs/latest/sql-performance-tuning.html

[3] https://www.databricks.com/blog/2020/06/18/introducing-apache-spark-3-0-now-available-in-databricks-runtime-7-0.html

[4] A methodology for spark parameter tuning. A Gounaris, J Torres - Big data research, 2018 - Elsevier