International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V71I5P103

Volume 71 Issue 5, 10-14, May 2023

Original Article

Performance Tuning and Optimization of Apache Spark Applications

Anish Ninan

Hitachi Solutions America LTD, 100 Spectrum Center Dr # 350, Irvine, CA, USA.

Received: 16 March 2023 Revised: 26 April 2023 Accepted: 09 May 2023 Published: 20 May 2023

Abstract *- Apache Spark has emerged as a powerful and widely used distributed data processing engine for big data analytics. However, achieving optimal performance in Spark applications can be challenging due to the complex nature of distributed computing and the myriad of configuration parameters involved. This paper presents a comprehensive study of performance tuning and optimization techniques for Apache Spark applications, with the goal of enabling users to maximize resource utilization, minimize execution time, and improve overall application efficiency.*

We begin by providing an overview of Apache Spark's architecture, including its data structures, core components, and execution model. This foundation allows us to explore the impact of various factors on Spark application performance, such as data partitioning, data serialization, and caching strategies. We then discuss critical performance-related parameters, including executor configuration, memory management, and garbage collection settings.

Next, we delve into advanced optimization techniques, such as adaptive query execution, dynamic allocation, and data locality. We demonstrate the effectiveness of these techniques through a series of experiments and benchmarks using real-world datasets and workloads. Additionally, we introduce tools and best practices for monitoring and profiling Spark applications, allowing users to identify and address performance bottlenecks.

By providing a comprehensive understanding of performance tuning and optimization for Apache Spark applications, this paper aims to empower users to harness the full potential of this powerful data processing engine, unlock new insights from their big data workloads and most importantly, save on costs!

Keywords *- Apache spark, Big data, AI, ML, Data Engineering, Performance tuning.*

1. Introduction

Apache Spark has emerged as a prominent distributed data processing engine for big data analytics, addressing several drawbacks of the Hadoop ecosystem, such as limited iterative processing capabilities and tight coupling of compute and storage resources. By offering fault-tolerant, parallelized processing capabilities and support for various data sources and APIs, Spark enables the separation of computing and storage, leading to improved scalability and flexibility. However, obtaining optimal performance in Spark applications remains challenging due to the intricate nature of distributed computing and the numerous configuration parameters involved. This paper presents an in-depth study of performance tuning and optimization techniques for Apache Spark applications, aiming to enhance efficiency, reduce execution time, and maximize resource utilization. By providing a detailed understanding of these techniques, we seek to empower users to unlock the full potential of Apache Spark and uncover valuable insights from their big data workloads.

2. Literature Review

Several studies have investigated the impact of different cluster configurations and resource allocation strategies on Spark performance. For example, Cheng et al. (2017) evaluated the effects of CPU, memory, and network bandwidth allocation on Spark job completion time and resource utilization. They found that over-provisioning resources beyond the optimal level can degrade performance due to contention and interference. Similarly, Li et al. (2018) analyzed the performance trade-offs between using large vs small Spark executor instances and recommended using a combination of both for different types of workloads.

3. What is Apache Spark?

Apache Spark is an open-source big data processing framework designed to process large volumes of data in a distributed and fault-tolerant manner. It was developed at the University of California, Berkeley, and later donated to the Apache Software Foundation, where it is now one of the top-level Apache projects. Spark provides an interface for programming in various languages like Scala, Python, Java, and R.

Spark is known for its ability to process large volumes of data quickly and efficiently, making it ideal for processing big data workloads. It achieves this by running computations in memory and using a distributed processing architecture that allows it to process data in parallel across many nodes in a cluster. Spark provides a wide range of

tools and libraries, including SQL, machine learning, graph processing, and streaming, making it a versatile platform for big data processing. Spark has a vibrant and active community of developers, contributing to its growth and development as a leading big data processing framework. In fact, 80% of the Fortune 500 use Apache Spark for their high-demanding data processing applications [4].

3.1. How does Spark Work?

Apache Spark can work on a single machine or deploy under the clustered computing architecture. What makes it so unique is that it uses in-memory caching and optimized execution to process queries against data of any size [5]. This significantly reduces the need for disk I/O and speeds up iterative algorithms. In contrast, Hadoop's MapReduce relies heavily on disk-based storage, leading to slower processing times. [6]. It's important to note that Spark is not a data storage solution but performs computations on Spark Java Virtual Machines (JVMs). [7].

Spark applications process data using Resilient Distributed Datasets (RDDs), DataFrames, and Datasets, with RDDs being the most fundamental data structure. Spark's execution model involves transforming data through a sequence of narrow and wide transformations, followed by actions that materialize the results. RDDs and DataFrames are read-only data collections that can be partitioned across a subset of Spark cluster machines and form the main working component [8]. This capability makes Spark one of the most simplistic Big Data processing engines on the market today. In addition, by providing a set of transformations and actions as operations, Spark offers a simple programming model that you can be used to build Big Data applications in familiar languages [9]. Spark supports various programming languages like Scala, Java, Python, SQL, and R. So, Data Engineers, scientists and developers have multiple ways to develop data applications based on the platform's architecture and processing resilience.

The other beauty of Spark is that it ships with its own unified API libraries like *Spark SQL* for structured streaming, *Spark MLlib* for Machine learning, *Spark Streaming* and *GraphX* for analytics.

Parallel processing in Spark allows you to execute concurrent workloads under one engine without the need for separate clusters for each [10]. Spark puts focus on its fast, parallel computation engine without having to prioritize storage [9].

3.2. Cluster Architecture

Driver is the leader and coordinates the activity of Executors. It is used for operations that require consolidating data (collect, toPandas) or coordinating executors (Kafka streaming). The driver can sometimes be the bottleneck of your job.

On the other hand, Executors are followers and receive instructions from the driver. They oversee the execution of individual tasks within each Spark job and have scalable computing power that enables them to process large amounts of data efficiently.

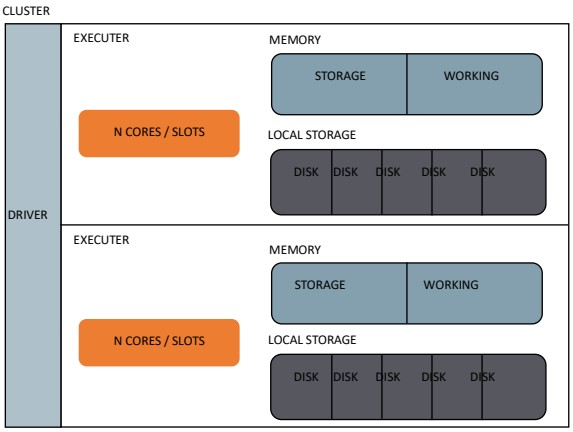

Fig. 1 Spark Architecture

4. Basic Performance Factors

It is possible to tune and optimize Spark applications to run faster to be able to achieve the results you want. There are several techniques to employ to maximize performance;

4.1. Data Partitioning

Data partitioning is the process of dividing a dataset into smaller, non-overlapping chunks called partitions. Each partition represents a subset of the data, and these partitions are distributed across the nodes of a Spark cluster. Data partitioning impacts the parallelism of a Spark application. Tasks are distributed evenly across the cluster by effectively partitioning data, reducing data skew and minimizing network overhead. The partitioning strategy should consider data size, key distribution, and the nature of operations performed.

4.2. Data Serialization

Serialization plays a vital role in Spark applications, affecting both performance and memory usage. While Spark uses Kryo serialization by default, users can customize serialization settings or implement custom serializers for specific data types to reduce overhead and improve performance. This library offers more optimization and performance potential compared to the Java serialization method. It is ten times faster than Java serialization as it serializes objects more quickly.

4.3. Shuffling

Shuffling in Spark is the process of redistributing data across partitions in a distributed computing environment. It typically occurs during operations that require data reorganization, such as joins, groupBy, reduceByKey, and repartition operations.

Shuffling can be a performance bottleneck in Spark applications due to the following reasons:

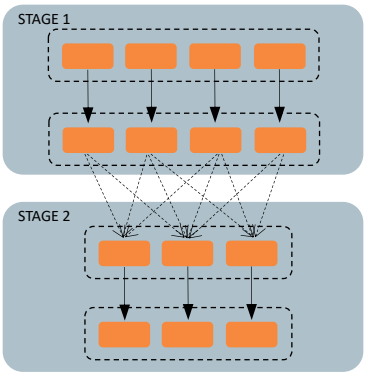

Fig 2. Shuffling

4.3.1. Data Movement

Shuffling involves transferring data across the network between executor nodes. The more data that needs to be moved, the longer it takes for the operation to complete.

4.3.2. I/O Overhead

As data is shuffled, it is written to disk and then read back into memory by the executor nodes. This disk I/O operation can be time-consuming, especially when dealing with large volumes of data.

4.3.3. Serialization and Deserialization

Shuffling requires data serialization and deserialization, which can add additional overhead to the operation.

4.3.4. Garbage Collection

Shuffling may generate many short-lived objects, leading to increased garbage collection pressure, which can impact overall performance.

4.4. Caching

Caching is a powerful technique to optimize iterative workloads or applications that reuse intermediate results. Persisting data in memory or disk can significantly reduce recomputation time and accelerate processing. However, caching should be used judiciously, as it can lead to memory pressure and garbage collection issues.

4.5. Data Skew

Data skew in Spark refers to uneven data distribution across partitions in a distributed computing environment. In other words, it occurs when some partitions have a significantly larger amount of data than others. Data skew can lead to performance bottlenecks and inefficient resource utilization in Spark applications, as it impacts the following aspects:

4.5.1. Parallelism

Ideally, data should be evenly distributed across all partitions, enabling tasks to be processed concurrently and efficiently. Data skew causes some tasks to take much longer to complete due to their larger partition sizes, while other tasks finish quickly, leading to an imbalance in task execution times.

4.5.2. Resource Utilization

When data is skewed, some executor nodes may be overloaded, processing more data than other nodes in the cluster. This can result in wasted resources, as underutilized nodes remain idle while waiting for the overloaded nodes to complete their tasks.

4.5.3. Network Overhead

Data skew can also cause increased network overhead during operations that involve shuffling, such as joins, groupBy, or reduceByKey. Skewed data may require more data movement across nodes, leading to higher network latency and slower query execution times.

To mitigate the impact of data skew in Spark, you can employ the following techniques:

4.5.4. Custom Partitioning

Use custom partitioning strategies, such as range partitioning or hash partitioning with a suitable key, to ensure even data distribution across partitions.

4.5.5. Increase Parallelism

Increase the number of partitions to reduce the amount of data per partition and promote better load balancing.

4.5.6. Salting

For operations like joins, introduce a random salt key to the skewed data by appending a random number to the join key. This helps redistribute the data more evenly across partitions, reducing the impact of skew.

4.5.7. Adaptive Query Execution

Enable Adaptive Query Execution (AQE) in Spark, which dynamically adjusts the execution plan based on runtime statistics. AQE can optimize operations like joins and aggregations to handle skewed data more efficiently.

4.5.8. Broadcasting

In the case of joining a large DataFrame with a small one, consider using broadcast joins. This will replicate the smaller DataFrame to all worker nodes, avoiding the need for shuffling and reducing the impact of data skew.

5. Advanced Optimization Techniques

5.1. Memory Usage Optimization

Spark uses *static allocation* and *dynamic allocation* of resources to applications to be able to run efficiently. The static allocation works in such a way that each application is satisfactorily assigned an appropriate size of resources on the cluster and reserves them for the duration as long as the SparkContext keeps running [17]. On the other hand, dynamic resource allocation can escalate the capability of the static allocation by automatically adding and removing executors of the Spark application as needed, based on a set of heuristics for estimated resource requirements.

Spark applications work by using in-memory caching [18]. So, efficient memory management is critical to achieving maximum application performance.

5.2. Adaptive Query Execution

Adaptive Query Execution (AQE) is an advanced optimization technique that dynamically adjusts query plans based on runtime statistics. AQE can optimize join strategies, repartition data, and adjust the degree of parallelism, resulting in significant performance improvements.

To enable AQE in Spark, you need to set the spark.sql.adaptive.enabled configuration property to true:

AQE introduces several optimizations, including:

5.2.1. Coalesce Shuffle Partitions

During operations like joins or aggregations, AQE can automatically coalesce shuffle partitions based on runtime statistics to reduce the number of output partitions. This optimization can minimize the overhead of small shuffle partitions and improve the parallelism of subsequent stages.

5.2.2. Skew Join Optimization

AQE can detect and handle skewed data in join operations by splitting the skewed partition into smaller, more balanced partitions. This technique ensures better load balancing and parallelism, reducing the impact of data skew on query performance.

5.2.3. Dynamic Partition Pruning

AQE can improve join performance by pruning unnecessary partitions in the fact table based on runtime filter values from the dimension table. This optimization can significantly reduce the amount of data read and processed during joint operations.

6. Spark Configurations

1. "spark.sql.shuffle.partitions" is a configuration parameter in Spark SQL that determines the number of partitions to use when shuffling data during query execution. The number of partitions can impact the performance of Spark SQL queries. Setting the value too high can lead to excessive memory usage while setting it too low can result in slow query execution times. As a best practice, the value of spark.sql.shuffle.partitions should be set based on the size of the data being shuffled and the available cluster resources to ensure optimal performance.
2. "spark.executor.memory" is a configuration setting that specifies the amount of memory allocated to each executor (worker node) in a Spark cluster. This setting defines the maximum amount of memory an executor can use to store data and execute tasks. If the value is set too low, the executor may run out of memory, leading to slower performance. On the other hand, if the value is set too high, it may lead to unnecessary memory usage and limit the number of executors that can run concurrently.
3. "spark.driver.memory" is a configuration setting in Apache Spark that specifies the amount of memory to allocate to the driver program. The driver program is the main program that controls the execution of a Spark application and runs on the driver node in the Spark cluster. This setting controls the amount of memory allocated to the driver JVM process. If the amount of memory allocated to the driver process is too low, it can cause out-of-memory errors and slow down the application. On the other hand, if the amount of memory allocated is too high, it can lead to inefficient resource usage and slow down other applications running on the same cluster. The value of spark.driver.memory is typically specified in gigabytes (e.g., 4g for four gigabytes).
4. "spark.executor.cores" is a configuration setting that specifies the number of CPU cores that each executor can use for processing tasks. The value of spark.executor.cores depends on several factors, such as the number of cores available on each worker node, the amount of memory allocated to each executor, and the nature of the workload being processed. Increasing the number of cores per executor can improve the parallelism and performance of the Spark application, but it can also increase the memory footprint of each executor and may result in increased contention for resources.

7. Conclusion

In conclusion, this technical paper has explored various optimization techniques for Apache Spark applications, emphasizing the importance of understanding and effectively addressing the challenges associated with distributed data processing. The paper has covered key aspects such as data partitioning, data shuffling, data skew, and Adaptive Query Execution, providing insights into their impact on performance, resource utilization, and scalability. Additionally, the paper also provided real-world spark configurations developers can set to optimize spark applications. Optimizing Spark applications can be challenging and requires specialized knowledge and expertise. It's important for developers to understand the various trade-offs involved in Spark optimization, such as the increased complexity, resource requirements, debugging challenges, memory management, and limited support for real-time data processing. Ultimately, the benefits of Spark optimization in terms of faster execution times, improved performance, and lower costs justify the effort and investment required to optimize Spark applications. As the demand for big data processing continues to grow and cost concerns increase, Spark optimization will become an increasingly important skill for developers and organizations looking to unlock the full potential of big data analytics.

References

[1] IBM, The First Multi-Core, 1GHz Processor. [Online]. Available: https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4

[2] Lawrence Livermore National Laboratory, Introduction to Parallel Computing Tutorial, 2022. [Online]. Available: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

[3] B. Chambers, and M. Zaharia, Spark: The Definitive Guide, 1005 Gravenstein Highway North, Sebastopol, CA: O'Reilly Media, Inc., 2018.
[4] Karthikeyan Rajendran, Speed Dialer: How AT&T Rings Up New Opportunities with Data Science, Nvidia, 2022. [Online]. Available: https://blogs.nvidia.com/blog/2022/03/22/att-data-science-rapids /
[5] AWS, What is Apache Spark?, AWS, 2022. [Online]. Available: https://aws.amazon.com/big-data/what-is-spark/#:~:text=Apache%20Spark%20is%20an%20open,against%20data%20of%20any%20size.
[6] IBM Cloud Education, Hadoop vs Spark: What's the Difference?, IBM, 2021. [Online]. Available: https://www.ibm.com/cloud/blog/hadoop-vs-spark
[7] Holden Karau, and Rachel Warren, *How Spark Works,* High Performance Spark, 2017, pp. 7-22.
[8] Apache Team, RDD Programming Guide, Apache, 2022. [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html
[9] Jules S. Damji et al., *Learning Spark*, 2nd Edition, O'Reilly Media, Inc., 2020. [Google Scholar] [Publisher Link]
[10] Institute of Computer Science, University of Tartu, Parallel Computing, Databricks. [Online]. Available: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/6908168003362015/2853703854010541/4620261684428706/latest.html
[11] Oracle Team, Interface Serializable, Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html
Aoache Team, Spark Configuration. [Online]. Available: https://spark.apache.org/docs/latest/configuration.html#spark-properties
[12] Azure Team, How to Improve Performance with Bucketing, Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/kb/data/bucketing
[13] Azure Team, Bucketing Example in Slack, Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/_static/notebooks/bucketing-example.html
[14] Vithal S., Apache Spark SQL Bucketing Support – Explanation, DW Geek, 2020. [Online]. Available: https://dwgeek.com/apache-spark-sql-bucketing-support-explanation.html
[15] Jun Guo, Bucketing 2.0: Improve Spark SQL Performance by Removing Shuffle, Databricks & Bytedance, 2020. [Online]. Available: https://www.databricks.com/session_na20/bucketing-2-0-improve-spark-sql-performance-by-removing-shuffle
[16] Siddharth Ghosh, Partitioning vs Bucketing — In Apache Spark, A Medium Corporation, 2022. [Online]. Available: https://medium.com/@ghoshsiddharth25/partitioning-vs-bucketing-in-apache-spark-a37b342082e4
[17] Holden Karau, and Rachel Warren, *Resource Allocation Across Applications*, High Performance Spark, 1005 Gravenstein Highway North, Sebastopol, CA 95472, O'Reilly Media, Inc., p. 20, 2017. [Publisher Link]
[18] Apache Team, Performance Tuning, Apache, 2022.[Online]. Available: https://spark.apache.org/docs/latest/sql-performance-tuning.html
[19] Apache Team, Tuning Spark, Apache, 2022. [Online]. Available: https://spark.apache.org/docs/latest/tuning.html