International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 05 | May 2020 www.irjet.net p-ISSN: 2395-0072

Parameter Tuning of Apache Spark based Applications for Performance Enhancement

Saipraveen PN[1], Nagaraja GS[2]

[1] *UG Student, Dept. of Computer Science and Engineering, RV College of Engineering, Bangalore, India*
[2] *Professor and Associate Dean, Dept. of Computer Science and Engineering, RV College of Engineering, Blr, India*

Abstract - *In the recent past, Apache Spark has become the most popular Big Data Analytics Framework, having taken on Apache Hive based on MapReduce due to the edge offered by Spark's in-memory computation. The key obstacles for Big Data Analytics are operating on tremendous volumes of data, managing wide variations in data and high-speed data processing. Spark provides default configurations which have been evaluated for low capability hardware and is not the most optimal solution for specific types of data and the computations performed on them. Fine grain control through the various performance parameters is essential to leverage maximum capabilities of Spark. Parameter tuning of available hardware resources for Spark applications takes the highest precedence to achieve optimal performance. However, there lacks an in-depth understanding on the impact of these performance parameters. This paper discusses in detail the various parameters such as number of executors, memory persistence levels, caching, broadcasting, serialization, compression, repartitioning and network parameters that can be tuned to enhance the efficiency of Spark applications tailored to the data being handled and the execution environment.*

Key Words: **Apache Hive, Apache Spark, Big Data, MapReduce, Parameter Tuning, Performance Optimization**

1.INTRODUCTION

In today's competitive world it is imperative for businesses of all scales to understand their customers' needs and trends in order to help serve them better. Data driven decision making is very crucial since every business today is inclined towards a growth beyond boundaries. It has been projected that, in 2020, information obtained per second from each individual person could approximate to 1.7 Mega Bytes aggregating to a volume of approximately 44 Zetta Bytes or 44 trillion Giga Bytes and by 2025, 463 Exa-Bytes of data would be generated every 24 hours [1]. Hence, Big Data Analytics plays a pivotal role in its overall growth and market penetration. With advancements in technology and tremendous increase in computational capabilities contributed by High Performance Distributed Computing, businesses are now capable of processing mammoth amounts of data being ingested at exponential rates. Huge expanse in mobile phones, IoT devices and other internet services are key sources of raw data which are churned to form sense out of it.

The growth and revenue of the business depends on critical Key Performance Indicators (KPIs) such as metrics (sales, stocking), analytics (product sales trend), forecasts (supply demand analysis), decision support systems etc. to name a few. The mentioned KPIs are driven through data, hence having the most optimized data processing capabilities is the need of the hour.

Big data Analytics has its challenges, processing of enormous amounts of data being produced at high rate and offering end-users reliable real-time results. Two approaches in evaluating or processing the data and overcome the limitation of latency are batch processing and stream processing. Batch processing works in a store-and-process fashion, but for real-time event detection it is important to process the data on data streams [2].

Technologies have been developed to handle such large amount of data in parallel. In 2004, Google Inc, published a paper titled "Google's MapReduce". A Map phase splits queries and circulates it crosswise over numerous nodes that handle in parallel, the outcomes are consolidated by the Reduce phase [3]. Apache Hive is a data warehouse tool implemented on top of MapReduce, it takes SQL queries which are operated on distributed data in the file system like a SQL database. Hive can also connect through a shell client or ODBC and execute SQL queries on the data stored on the Hadoop cluster. Hive makes it easy to port form SQL-based applications to Hadoop. Apache Spark is a unified cluster computing engine designed to be fast and general-purpose, it is a computational engine at the core and takes responsibility of scheduling, distributing and monitoring applications across multiple worker machines, or a computing cluster. Spark extends the widely adopted MapReduce model, on top of either yarn or Mesos to run queries on data and efficiently supporting more types of computations including interactive queries and stream processing.

Spark Architecture Fig. 1 illustrates the various processing modules of Spark, such as Spark Core which consists of fault tolerant and immutable Resilient Distributed Datasets (RDDs), APIs for performing SQL like operations on RDDs, schedulers for assigning tasks to various executor systems and memory management components. Spark SQL allows writing of SQL Queries against data or use SQL like functions to transform datasets in Spark. Spark Streaming enables

real-time input data processing. Spark MLlib provides an entire library of machine learning and data mining tools to run on datasets in Spark and GraphX library for analysis and processing of graph-based data structures [3]. Apache Spark facilitates two special kinds of operations namely, Trans-formations and Actions. Transformations perform a lazy evaluation where a dependency graph is built on the RDDs during transformations until an action is triggered, during which spark figures out the shortest path across the dependencies by working backwards. Actions return RDDs of DataFrames to the driver program on completion of computations.

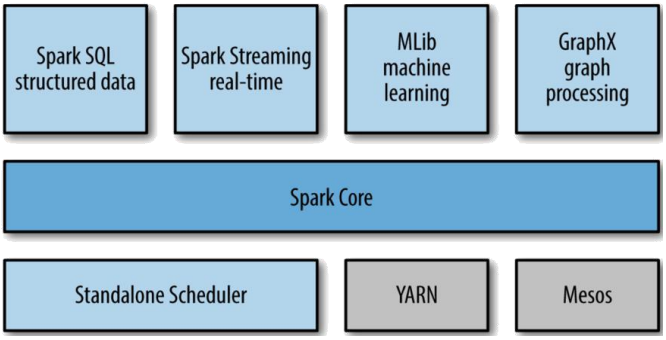

Fig -1: Components of Apache Spark

2. OVERVIEW AND PROBLEM STATEMENT

The Apache Spark application considered for assessing the performance of parameter tuning is an implementation of a Real-time Twitter Streaming System. Twitter has over 330 million monthly active users and generates over 500 million tweets per day. The streaming System utilizes developer APIs and other streaming systems to access, process and store real-time tweets.

The experimental study was performed on a Hadoop cluster with 4 worker nodes running Apache Spark 2.4, each worker node is allocated with 128 GB of RAM and 64 processor cores. The Experiment section will cover the impact of tuning performance parameters like number of executors, memory persistence levels, caching, broadcasting, serialization, com-pression, repartitioning and network parameters etc. on the streaming system. The execution times is the metric taken into consideration for evaluating the performance, it has been tabulated in Table 2 to derive insights and conclude the analysis.

3. EXPERIMENTATION

The Real-time Twitter Streaming System takes Access Token, Access Token Secret, Consumer Key (API Key) and Consumer Secret (API Secret) as arguments for the developer API to access online tweets. Apache Flume offers an architecture for easy and robust streaming of data flows, tweets are fetched every 90 seconds. Various Spark transformation operations are performed on the contents of the tweets.

On execution of the streaming application with the default configurations provided by Spark, the results for over 40,000 tweets were provided in 97 seconds. Further subsections indicate the performance improvements on tuning of various performance parameters.

3.1 Number of Executors

Choosing the optimal number of executors based on the available cores, number of nodes in the cluster, volume of data and available executor memory gives a significant performance improvement. The Spark application can be configured through the properties such as 'spark.driver .cores', 'spark.driver.memory', 'spark.executor-.cores', 'spark. executor.memory' and 'spark.executor.-instances'. Fig. 2 depicts the Spark hierarchy to understand the various processes involved in the execution of a Spark application. A strategy for optimising the number of executors has been given in [4] whose steps are as follows:

A. Calculation of Available Cores

The number of available cores are first identified which is 64 cores per node and 256 total available cores for processing. Hadoop or Yarn daemons executing in the background require one core which has to be excluded from the available cores resulting in 252 remaining cores considering 4 worker nodes.

B. Calculation of Number of Executors per Node

The thumb rule to select the number of cores per executor is 5 cores per executor. With this assumption, the number of executors to be set is 252/5 i.e. 51 executors. The cluster manager requires one executor for maintaining metadata of the submitted application. Finally leaving 50 executors in the cluster hence 'spark.executor.instances' is set to 12.

C. Calculation of Memory per Executor

Each cluster node is equipped with 128GB of RAM. The total memory has to be distributed to the 12 executors per node Hence, the executor memory is calculated to be 10GB, 'spark.executor.memory' is set accordingly.

D. Repartitioning of Data

Repartitioning is done on data for faster access, the number of partitions are chosen to be a multiple of the total number of available executors in the node. Incorporation of the above steps helped to set the optimal hardware resources for the Twitter Streaming System. The results indicated in Table 2 shows a phenomenal reduction in the execution time by about 71%.

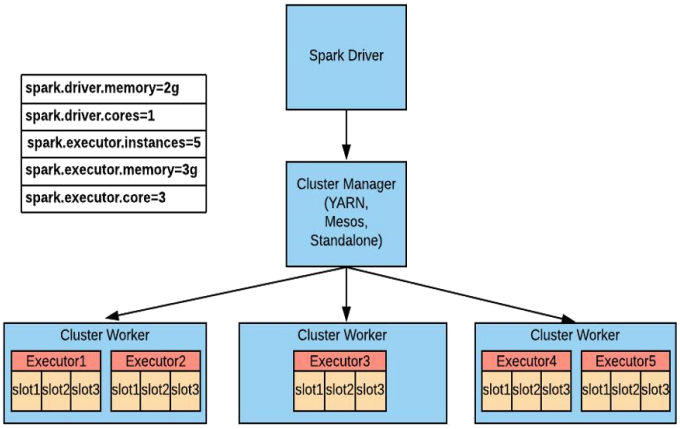

Fig -2: Apache Spark Hierarchy

3.2 Caching of DataFrames

Caching is an optimization strategy where incremental and dynamic computations are performed. Caching techniques provided by Spark allow the saving of partial intermediate results in either memory or can persist in solid storages for subsequent reuse based on the available memory. In Spark, memory allocation comes into one of two classifications: execution and storage. Execution memory corresponds to computation in shuffle, join, sort, and aggregation operations, whereas storage memory corresponds to caching and distribution of internal data across the cluster. 'spark.memory.fraction' expresses the size of the unified memory region as a fraction of the JVM heap space which is occupies 60% memory by default, to persist RDDs. The rest 40% memory is reserved for objects created during execution, user data structures and Spark internal metadata. 'spark.memory.storageFraction' expresses the size of minimum storage space as a fraction of the unified memory region which is 0.5 by default [5]. Cached blocks in the minimum storage space are immune to being evicted during execution.

The Twitter Streaming System generated and reused four data frames for several transformations before completion of the job. To assess the impact of caching, all data frames were cached in memory subsequently on creation and execution times were compared.

Observations from Table 2 indicate that the application executed approximately 42% faster after caching and reuse of intermediate results. Additionally, a dynamic memory management algorithm for Apache Spark can be utilised to handle factors that influence memory management such as optimal storage level, spark memory ratio selections, heavy memory burden and garbage collection overheads. Size of data to be cached and available memory size can be considered to determine the optimal storage categories namely, MEMORY AND DISK, MEM-ORY ONLY SER MEMORY AND DISK SER and MEM-ORY ONLY [6].

3.3 Adaptive Serialization

Run-time statistics such as JVM heap memory usage, partition size, cached output size, I/O speed and serialization performance can be collected dynamically and stored in round-robin structures for selection of system storage categories listed in the previous subsection. Serialization though Java and Kryo algorithms are supported by Spark, the default serialization configuration is changed from one RDD to one block to gain flexibility. In cases where RDDs have a greater number of blocks, different serialization algorithms can be used on different blocks in a single RDD. This dynamic serialization strategy yields up to a speedup of 2 when compared to the unoptimized Spark execution [7].

3.4 Broadcasting of Variables

Broadcast variables get shared throughout all the nodes in the cluster. Broadcast join is an effective strategy of optimization in Spark to maximize the efficiency of join operations. For a map join, one of the join components may be materialized to prevent the sorting and shuffling step which is a computationally expensive operation. Shuffling incurs data migration with also impacts bandwidth, but shuffling can be useful at times where data skewness has to be resolved. Data locality is how close data is to the process. Locality has an impact during broadcasting, based on the locality the following parameters can be set viz. PROCESS LOCAL, NODE LOCAL, NO PREF, RACK LOCAL and ANY. The broadcasted values are sent as a file to all mappers and joined with the other table during its read operation, boosting efficiency [8]. While performing a join operation, the Twitter Streaming application broadcasts the smaller data frame to all nodes in the cluster. Observations from Table 2 indicate that the application executed approximately 16% faster due to broadcasting of the smaller table.

3.5 Partitioning

One of the essential optimization strategies for enhancing Apache Spark's efficiency is to increase the degree of parallelism for the tasks to be carried out. Data partitioning governs how to access all hardware resources while executing a job and is an important concept for achieving parallelism in Apache Spark. By default, Spark reads data from neighbouring cluster nodes into a Resilient Distributed Dataset (RDD). Transformations on chunks of data can be automated once partitions are built. Given the configuration of clusters and the task requirements, the level of partitions in Spark must be taken with caution. It is crucial to pick the correct amount of partitions required. Increasing to a large number of partitions implies that every partition has rather less or no records. Far too few partitions result in handling larger collection of data. For a given job, an ideal amount of partitions should be determined depending on the number of available processor cores and the usable RAM for each executor. The Driver process of Spark will execute one

parallel job for each partition in the RDD, until the available number of cores. In the experimental scenario, the RDDs can have 256 partitions at the most since there are 256 cores present in the cluster. Spark performs a shuffled hash join by default, the smaller data frame is broadcasted to minimises shuffling followed by the join operation.

Different numbers of partitions along with equal corresponding number of cores are evaluated for execution time. In the unoptimized run, 16 partitions were allocated to a single core. Subsequent runs consider double the partitions of the previous steps along with broadcast join as indicated in Table 1 for increased number of partitions and avoiding of shuffles.

From Table 2 it can be inferred that a 69% reduction in the execution time could be achieved. The results indicate that on increasing the number of partitions and cores using 'total-executor-cores' from 16 to the highest possible cap of 256 there was no major difference in the execution times.

Table -1: Performance Evaluation on Partitioning

Performance Parameter	No. of Partitions	No. of Cores	Execution Time (sec)
Default	Default	Default	97
Default	16	16	206
Broadcast Join	16	16	30
Broadcast Join	32	32	32
Broadcast Join	64	64	31
Broadcast Join	128	128	28
Broadcast Join	256	256	28

3.6 Garbage Collection

JVM (Java Virtual Machine) Garbage Collector (GC) evicts old objects to make room for new ones, by tracing through all user created Java objects and identify the objects without any reference. Cost of garbage collection is proportional to the number of Java objects, frequent run-time execution of the JVM GC indicates insufficient memory for computation. The memory used for caching of RDDs has to be reduced to lower GC overhead. Alternatively, if GC is not frequently executed the memory used for caching of RDDs can be increased. The memory fraction for caching can be set to spark.storage.memoryFraction in the Spark properties file. Statistics on frequency of garbage collection and time spent by GC can be obtained by adding 'verbose:gc -XX: + PrintGC Details -XX: +PrintGCTimeStamps' to the Java options.

4. RESULTS

The Experimentation section analysed the execution time considering tuning parameters such as Number of Executors, Caching of DataFrames, Broadcasting of variables, Partitioning and other miscellaneous optimizations at an independent level and observed maximum reduction in the application execution time by performing resource optimizations. However, the efficiency can be further enhanced by combining of multiple tuning parameters discussed in the previous sections. In the Twitter Streaming System, combination with caching, optimization of number of executors, Broadcasting and partitioning indicated in Table 2 resulting in 88.6% reduction in execution time on the Twitter Streaming Service application.

Table -2: Performance Evaluation After Tuning Individual Parameters

Performance Parameter	Execution Time (sec)	Performance Improvement %
Default Configuration	97	0
Broadcasting of Variables	81	16.5
Caching of DataFrames	57	41.2
Partitioning	30	69
Hardware Resource Optimisation	23	76.2
Combination of Parameters	13	86.6

5. CONCLUSION AND FUTURE WORK

It can be concluded from this study that certain parameters could be tuned in the Twitter Streaming System to achieve maximum efficiency in terms of execution time of the Spark applications and emphasizing the significance of performance tuning in Spark based applications as opposed to MapReduce applications. The results give a clear indication that the suggested optimization techniques could achieve a phenomenal 86.6% reduction in the execution time on tuning of parameters. Although Spark can perform much faster due to its in-memory computations and execution plans, it can be concluded that unless the user does not have an understanding of the properties of the data being handled, fine tuning cannot be performed and tuning is very much essential to leverage the best performance of Apache Spark. This study took up the performance tuning in the cluster with the standalone cluster manager. Further study can include the performance of various cluster managers like Yarn and Mesos which are prevalent scenarios in real-time business use-cases. Additionally, using Scala or Python as the language for Spark applications has been a debatable topic amongst developers, it can be further explored to identify the scenarios suitable for either of the languages.

REFERENCES

[1] Seedscientific, "Big Stats and Facts About Big Data", https://seedscientific.com/how-much-data-is-created-every-day/

[2] D. Puthal, S. Nepal, R. Ranjan and J. Chen, "A Secure Big Data Stream Analytics Framework for Disaster Management on the Cloud," 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International

Conference on Data Science and Systems (HPCC/ SmartCity/DSS), Sydney, NSW, 2016, pp. 1218-1225.

[3] Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia "Learning Spark" O'Reilly Media, Inc.

[4] P. Ramprasad, "Understanding Resource Allocation configurations for a Spark application," http://site.clairvoyantsoft.com/understanding-resource-allocation-configurations-Spark-application/

[5] Memory Management Overview https://spark.apache.org /docs/latest/tuning.html

[6] "S. Chae and T. Chung, DSMM: A Dynamic Setting for Memory Management in Apache Spark, 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Madison, WI, USA, 2019, pp. 143-144."

[7] Y. Zhao, F. Hu and H. Chen, "An adaptive tuning strategy on spark based on in-memory computation characteristics," 2016 18th International Conference on Advanced Communication Technology (ICACT), Pyeongchang, 2016, pp. 484-488.

[8] "Broadcast Join with Spark" https://henning.kropponline.de/2016/12/11/broadcast-join-with-Spark/