Beeline vs. Spark : How to Speed Up Batch Processing Dramatically : Boost Efficiency up to 16x

Ipsita Rudra Sharma [1]

[1] *Senior Data Engineer, AVP, Deutsche Bank*

---***---

Abstract - *Beeline and Spark-SQL are both powerful tools used for interacting with Apache Spark, a popular open-source distributed computing framework for large-scale data processing. While they serve similar purposes, there are some key differences between the two. Beeline is a command-line interface (CLI) tool that allows users to execute queries on Hive tables, similar to how one might use the traditional Hive CLI. It provides a familiar, text-based environment for running queries and accessing data stored in Spark's data sources. In contrast, Spark-SQL is a specific module within the Spark ecosystem that adds SQL query capabilities directly into Spark applications. This allows developers to seamlessly integrate SQL querying functionality into their Spark-based data pipelines and analytics workflows. Spark-SQL supports a wide range of SQL dialects and data source types, making it a more flexible and programmatic option compared to the more standalone Beeline CLI. Additionally, Spark-SQL can leverage Spark's distributed processing power to execute complex queries across large datasets much more efficiently than a traditional SQL engine. The choice between Beeline and Spark-SQL often comes down to the specific needs of a project - Beeline may be preferable for ad-hoc querying, while Spark-SQL is better suited for tightly-integrated, Spark-powered applications that require robust SQL capabilities. Ultimately, both tools provide valuable ways to interact with and leverage the power of the Apache Spark framework.*

Key Words*: *Big Data, Hadoop, HDFS, MapReduce, Beeline, MRjob, Optimization, Hive, Apache Spark

1. HOW SPARK-SQL WORKS

Spark SQL is a powerful component within the Apache Spark ecosystem that allows for the efficient processing and querying of structured data. At its core, Spark SQL provides a DataFrame API, which represents data in a tabular format similar to a database table, making it easy to work with and manipulate. Under the hood, Spark SQL leverages the Spark engine to optimize and execute these DataFrame operations in a distributed, fault-tolerant manner. When a Spark SQL query is executed, the DataFrame is first analyzed and parsed into a logical plan, which represents the high-level steps required to compute the desired result. This logical plan is then optimized by Spark SQL's optimizer, which applies various rule-based and cost-based optimizations to generate an efficient physical execution plan. This physical plan is then translated into Spark's lower-level execution model, taking advantage of Spark's in-memory processing capabilities and distributed computing architecture to rapidly process the data.[11] Spark SQL also supports a wide range of data sources, from CSV and JSON files to popular data warehousing solutions like Hive, allowing users to seamlessly integrate structured data from various sources into their Spark-powered applications. With its intuitive API, optimization capabilities, and broad data source support, Spark SQL has become a go-to tool for data engineers and data scientists working with large-scale, structured data in the Apache Spark ecosystem.[8]

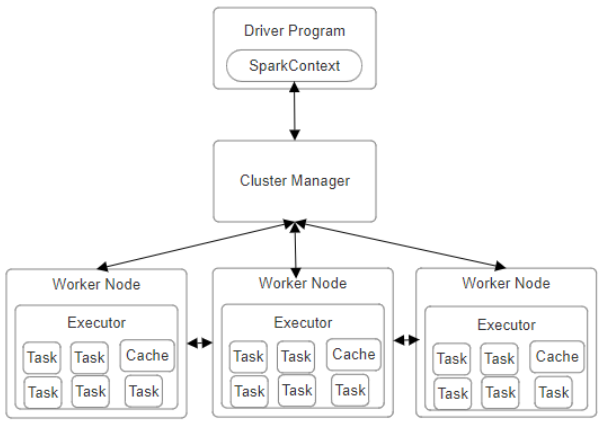


Fig1: Spark master-slave architecture

2. HOW BEELINE WORKS

In the world of Big Data, Beeline serves as a powerful tool for processing and analyzing massive amounts of data. In the context of Big Data, Beeline is a command-line interface that allows users to interact with Apache Hive, a popular data warehousing solution built on top of the Hadoop distributed file system. Hive provides a SQL-like language called HiveQL, which Beeline utilizes to enable users to write and execute complex queries against large datasets stored in Hadoop. Through Beeline, data analysts and engineers can seamlessly access, explore, and gain valuable insights from their organization's Big Data repositories. The beauty of Beeline lies in its simplicity and efficiency - it provides a straightforward, text-based interface for interacting with Hive, allowing users to quickly prototype queries, generate reports, and uncover hidden patterns and trends within their data. Furthermore, Beeline's integration with Hadoop makes it a crucial part of the Big Data ecosystem, empowering organizations to harness the full potential of their unstructured, large-scale data and transform it into actionable business intelligence. Whether you're a seasoned data professional or new to the world of Big Data, Beeline serves as an indispensable tool for unlocking the hidden value buried within your organization's growing volumes of information.

3. HOW HIVE WORKS

Hive is a powerful open-source data warehouse system that allows for the efficient storage, processing, and querying of large datasets. At its core, Hive works by providing a SQL-like interface on top of the Hadoop distributed file system (HDFS), enabling users to seamlessly access and manipulate data at scale. The way Hive achieves this is by first organizing data into tables, similar to a traditional relational database, but with the added benefit of being able to store and process unstructured data as well. These tables are then partitioned and bucketed, allowing for faster querying and improved query optimization. Hive then translates the SQL-like queries entered by the user into MapReduce jobs, which are then executed across the Hadoop cluster. This abstraction layer means that users can leverage the power of Hadoop without needing to be experts in the underlying distributed computing framework. Furthermore, Hive provides a rich ecosystem of user-defined functions, serializers/deserializers, and integrations with other Big Data tools, making it a highly versatile and extensible platform for data analytics and business intelligence. Whether you're dealing with terabytes of log data, petabytes of sensor readings, or any other large-scale data challenge, Hive's unique architecture and features make it an indispensable tool for making sense of it all.[7]

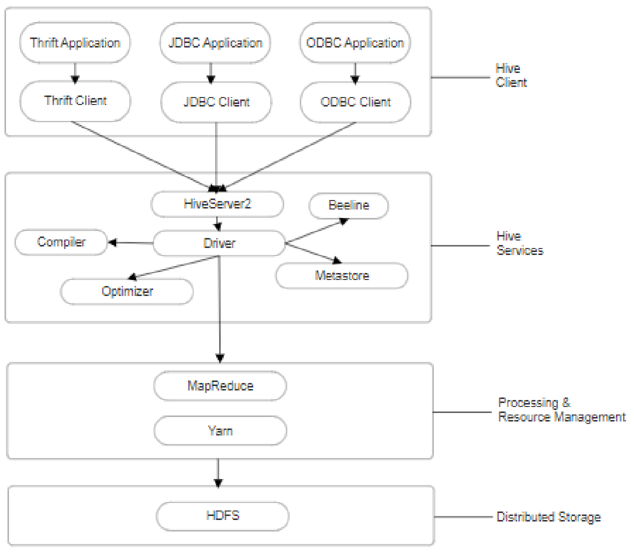


Fig2: Hive architecture

4. HOW MAPREDUCE WORKS

MapReduce is a powerful programming model and software framework that enables the processing and analysis of large-scale, unstructured datasets in a highly scalable and efficient manner. At its core, the MapReduce approach breaks down complex computational tasks into two key phases - the "Map" phase and the "Reduce" phase. In the Map phase, the input data is divided into smaller, manageable chunks that can be processed in parallel across a distributed network of machines. Each mapper node takes its assigned data slice and applies a user-defined transformation or function to generate a set of intermediate key-value pairs. These intermediate results are then shuffled and sorted, before being passed to the Reduce phase. During the Reduce phase, the sorted key-value pairs are aggregated by key, and a final transformation is applied to generate the ultimate output.[9] This divide-and-conquer strategy allows MapReduce to harness the collective processing power of many commodity servers, making it ideally suited for handling massive volumes of information that would overwhelm a single powerful machine. The inherent parallelism, fault-tolerance, and scalability of the MapReduce framework have made it a cornerstone of Big Data analytics, powering the data processing pipelines of tech giants and enterprises alike as they seek to extract valuable insights from their rapidly growing troves of unstructured information.

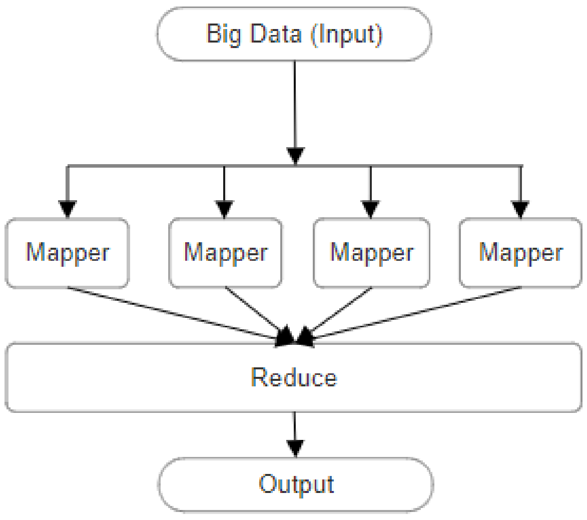


Fig3: MapReduce architecture

5. HiveQL Vs. Spark-SQL

Hive and Spark-SQL are two of the most prominent and widely-used Big Data processing engines in the modern data ecosystem. While both technologies serve the purpose of querying and analyzing large datasets, they differ in their underlying architectures, capabilities, and target use cases.

Hive, which was originally developed by Facebook, is built on top of the Hadoop Distributed File System (HDFS) and utilizes MapReduce as its core processing framework. It provides a SQL-like querying language called HiveQL, which allows data analysts and engineers to write complex analytical queries against structured data stored in a Hive data warehouse. Hive is particularly well-suited for batch processing of large, static datasets, making it a popular choice for traditional business intelligence and data warehousing workloads.

Hive utilizes MapReduce framework which deserves a special mention in this article while comparing Hive and Spark. MapReduce, developed by Google, follows a two-stage batch processing model - the "map" stage breaks down the input data into smaller chunks that can be processed in parallel, while the "reduce" stage aggregates the results. This sequential, disk-based approach can be efficient for certain types of batch workloads, but it can also be slow and inflexible, especially for iterative or interactive data processing tasks.

In contrast, Spark-SQL is part of the Apache Spark project, which takes a more unified and in-memory approach to data processing. Spark-SQL provides a DataFrame API that allows for the manipulation of structured data using familiar SQL syntax, while also leveraging Spark's powerful distributed processing engine for faster, more efficient query execution. Compared to Hive and MapReduce, Spark-SQL excels at low-latency, interactive analytics on both batch and streaming data, making it a preferred choice for real-time applications, machine learning, and advanced data science use cases.[18] While MapReduce excels at simple, one-pass transformations of large data volumes, Spark's in-memory architecture and diverse APIs make it a more versatile and performant choice for many modern data processing workloads, especially those requiring low-latency, iterative computations. In terms of speed spark is deemed to run programs up to 100 times faster in memory or 10 times faster on disk than Map Reduce.

The choice between Hive and Spark-SQL ultimately depends on the specific requirements of the data processing task, the volume and velocity of the data, and the desired performance characteristics of the analytics pipeline.

6. SPARK OPTIMIZATION TECHNIQUES

Before proceeding ahead to the next section, it is imperative to understand a few techniques that help with Spark job optimization. Spark optimization is a critical consideration when working with large-scale data processing, and the choice of joining methods can have a significant impact on the overall performance and efficiency of a Spark application. While Hive has pretty powerful optimization tricks up its sleeve, Spark excels with its own techniques. Here are a few:

6.1. Cache & Persist: Spark's optimization techniques, particularly the cache and persist functions can be used for enhancing the performance and efficiency of data processing pipelines. The cache function allows Spark to store the results of an operation in memory, enabling faster access and retrieval in subsequent computations. This is especially beneficial when working with datasets that are accessed repeatedly, as it eliminates the need to recompute the same data, saving time and resources. The persist function takes caching a step further by allowing users to specify the storage level, determining whether the data should be stored in memory, on disk, or a combination of both. This flexibility is crucial, as different storage levels are suited for different workloads and hardware configurations. For example, storing data in memory can provide lightning-fast access but may be limited by available RAM, whereas disk-based storage offers more capacity but slightly slower retrieval times.

By carefully selecting the appropriate persist storage level, users can strike the right balance between performance and cost-effectiveness, tailoring the system to their specific needs.[8]

6.2. Careful Selection of Joins: Spark provides several join strategies, each with its own strengths and tradeoffs, and selecting the appropriate method can make the difference between a lightning-fast data pipeline and one that crawls along. The broadcast join, for example, is well-suited for situations where one of the input datasets is relatively small, as Spark can efficiently distribute that dataset to all worker nodes, enabling highly parallelized processing. In contrast, the shuffle join is better equipped to handle larger datasets that don't fit comfortably in memory, though this approach does incur the overhead of redistributing data across the cluster. More advanced techniques, such as the sort-merge join, leverage sorting and partitioning to minimize data movement and maximize throughput.

Ultimately, the optimal joining method will depend on the specific characteristics of the data, the hardware and cluster resources available, and the performance requirements of the application.[24]

6.3. Shared variables: Spark's approach to shared variables involves two primary mechanisms: broadcast variables and accumulators. Broadcast variables allow for the distribution of read-only data to all the worker nodes in a Spark cluster, eliminating the need to repeatedly transmit the same information across the network. This is particularly useful for lookup tables, configuration parameters, and other static data that is accessed frequently during computations. In contrast, accumulators provide a way for worker nodes to safely update shared values, such as counters or sums, in a distributed and fault-tolerant manner. By leveraging these constructs, Spark can minimize data duplication, reduce network traffic, and enable parallel processing of data without the risk of race conditions or data inconsistencies.

By carefully choosing which type of shared variable is to be used in which scenarios can be a determinant factor in spark optimization.

6.4. Avoid Shuffling: Shuffling, which involves the redistribution of data across partitions, can be a significant bottleneck in Spark workloads, as it requires expensive network communication and can lead to increased processing time and resource utilization. By optimizing Spark jobs to minimize or eliminate shuffling, data engineers can unlock significant performance gains and achieve more scalable and cost-effective data processing pipelines.

One of the primary strategies for avoiding shuffling in Spark is to leverage the concept of partitioning. Partitioning involves dividing the input data into smaller, more manageable chunks that can be processed independently and in parallel by different Spark executors. By carefully designing the partitioning scheme, data engineers can ensure that related data is co-located on the same partitions, reducing the need for expensive data shuffling operations. This can be achieved through techniques such as hash partitioning, range partitioning, or custom partitioning functions that consider the specific characteristics of the data and the processing requirements.

An important aspect of Spark optimization to avoid shuffling is the strategic use of Spark's transformation operations. Certain Spark transformations, such as map(), filter(), and flatMap(), are considered narrow transformations, as they can be executed without the need for data shuffling. In contrast, transformations like join(), groupByKey(), and reduceByKey() are considered wide transformations, as they require data to be shuffled across partitions. By prioritizing the use of narrow transformations and carefully designing the data flow to minimize the need for wide transformations, data engineers can significantly reduce the amount of shuffling required and improve the overall performance of their Spark applications.

6.5. Kryo Serialization: At its core, Kryo is a high-performance serialization library that can significantly reduce the size and processing time of data being transmitted across a Spark cluster. By leveraging Kryo's compact binary serialization format instead of the default Java serialization, Spark is able to minimize the network overhead associated with shuffling and broadcasting data between executor nodes. This not only accelerates job execution times, but also reduces the strain on cluster resources like network bandwidth and storage. Kryo achieves these gains by employing several optimization strategies, such as the ability to automatically generate serializers for user-defined classes, support for compression, and customizable registration of class IDs to avoid the overhead of full class names. Additionally, Kryo's serializers are designed to be thread-safe and reusable, further enhancing its scalability within a Spark environment that often involves highly parallel processing workloads. For Spark applications dealing with large datasets or requiring rapid data movement, implementing Kryo serialization can be a transformative optimization that unlocks major performance improvements and resource savings across the board. To use Kryo serialization in Spark job, initialize the SparkConf and set the "spark.serializer" configuration option to "org.apache.spark.serializer.KryoSerializer".

6.6. Adaptive query execution: Spark's adaptive query execution is a powerful optimization technique that dynamically adjusts the execution plan of a query based on runtime conditions, leading to significant performance improvements. When a Spark query is submitted, the system initially generates an initial execution plan, but as the query executes, Spark continuously monitors the data distribution and processing progress. If Spark detects any skew in the data or imbalances in the workload across partitions, it can adaptively modify the plan on the fly to address these issues. For example, if Spark identifies a heavily skewed partition that is causing a bottleneck, it can dynamically repartition the data to achieve better load balancing. Similarly, if Spark observes that a particular operator is performing poorly, it can switch to a more efficient implementation, such as transitioning from a sort-merge join to a broadcast hash join. This adaptive approach allows Spark to continuously optimize the query execution, taking advantage of runtime statistics that were not available during the initial planning phase. By adapting the plan based on observed conditions, Spark can overcome limitations of static, pre-determined execution plans, leading to faster, more efficient query processing, especially for complex analytical workloads with unpredictable data characteristics. The net result is improved query performance and resource utilization, without requiring manual tuning or extensive upfront analysis, making Spark's adaptive query execution a valuable tool in the Big Data analytics arsenal.[23]

Spark's adaptive query execution feature can automatically coalesce post-shuffle partitions based on map output statistics, simplifying the process of tuning the shuffle partition count. When both spark.sql.adaptive.enabled and spark.sql.adaptive.coalescePartitions.enabled are set to true, Spark can dynamically adjust the shuffle partition number at runtime to best fit the data, eliminating the need to manually configure a "proper" partition count upfront. Users only need to set a sufficiently large initial partition count via the spark.sql.adaptive.coalescePartitions.initialPartitionNum configuration, and Spark will handle the optimization.[8]

6.7. Parameter Tuning: This is a crucial process that is designed to process large datasets in a distributed computing environment. The Spark executor is the worker process responsible for executing tasks and storing data in memory or on disk, and its configuration can have a significant impact on the overall efficiency and speed of a Spark job. Proper performance tuning involves carefully adjusting parameters such as the number of executors, the amount of memory and CPU allocated to each executor, and the number of cores used by each executor. By striking the right balance, users can ensure that tasks are distributed efficiently across the cluster, minimizing bottlenecks and maximizing resource utilization. For example, increasing the number of executors can improve parallelism and throughput, but too many executors may overwhelm the available resources and lead to thrashing. Similarly, allocating more memory per executor can reduce the need for disk-based processing, but excessive memory allocation may result in fewer executors running concurrently. Spark provides a range of configuration options to fine-tune the executor settings, and experienced data engineers often rely on profiling, monitoring, and trial-and-error to identify the optimal combination for their specific workloads and infrastructure. By mastering parameter tuning, Spark users can unlock the full potential of their data processing pipelines, achieving faster runtimes, higher throughput, and more efficient resource utilization.[8]

6.8. sortBy or orderBy: In a nutshell the "orderBy" function is used to sort the entire dataset across all partitions, while "sortBy" sorts the data within each individual partition.

The key difference lies in the scope and scale of the sorting operation. "orderBy" performs a global sort, which requires Spark to shuffle the entire dataset across the cluster to organize the rows in the desired order. This can be computationally intensive, especially for large datasets, as it involves moving large amounts of data between executors. In contrast, "sortBy" only sorts the data within each partition, without the need for a full dataset shuffle. This can be significantly more efficient, as the sorting can be performed in parallel across the partitions, reducing the overall processing time.

The choice between "orderBy" and "sortBy" ultimately depends on the specific requirements of the use case. If the analysis requires a global sort of the entire dataset, then "orderBy" is the appropriate choice. However, if the sorting can be performed within individual partitions without compromising the final result, "sortBy" is generally the more efficient option. By carefully considering the trade-offs and selecting the appropriate function, Spark developers can optimize the performance of their queries and ensure efficient data processing at scale.

6.9. reduceByKey or groupByKey: reduceByKey() is a more optimized operation that combines values with the same key using a provided reduce function, aggregating the data in a more efficient manner. In contrast, groupByKey() first groups all values by their respective keys, and then applies a separate reduction step, which can be less efficient for certain workloads.

The key distinction is that reduceByKey() performs the aggregation and reduction in a single pass, minimizing the amount of data that needs to be shuffled across the network. This is particularly advantageous when dealing with large datasets, as it reduces the network overhead and memory requirements. reduceByKey() is well-suited for use cases where you need to perform operations like summing, counting, or averaging values grouped by key. The reduce function you provide is applied directly on the grouped data, streamlining the computation.

On the other hand, groupByKey() first collects all values for each key, and then a separate reduction step is applied. This two-stage process can be less efficient, especially when the grouped data is large and doesn't fit in memory. groupByKey() may be more appropriate when you need to perform more complex transformations on the grouped data, or when the reduce function is not straightforward to implement. In these cases, the flexibility of groupByKey() can outweigh the performance benefits of reduceByKey(). Understanding the trade-offs and characteristics of each operation is crucial for optimizing the performance and efficiency of your Spark applications, ensuring you can effectively harness the power of distributed data processing.

6.10. Coalesce vs. Repartition: Coalesce is a Spark operation that combines multiple partitions into a smaller number of partitions, reducing the overall number of partitions in the dataset. This can be beneficial when you have a large number of small partitions, as it reduces the overhead associated with managing all those individual partitions. Coalesce is a relatively lightweight operation that doesn't necessarily require a full shuffle of the data. In contrast, repartition is a more heavy-duty operation that completely reshuffles the data across a specified number of new partitions. Repartitioning is useful when you want to change the partitioning scheme of your data, such as partitioning by a different column or achieving a more optimal number of partitions. While repartition involves a full data shuffle which can be more computationally expensive, it also gives you more control over the partitioning of your data. Depending on the specific requirements and characteristics of your Spark workload, you may find that one technique or the other (or a combination of both) is better suited to optimizing performance.

Ultimately, mastering these optimization strategies is essential for unleashing the full potential of Spark and delivering high-performance, scalable data solutions.

7. EXAMPLE WITH IMPLEMENTATION

In this implementation a simple tweak in a Big Data batch job script increased its performance by leaps and bounds. Imagine a scenario where a daily batch job runs for 5 to 6 hours on an average only to fail in the end. Then an engineer logs in to the Production environment – which is highly risky, access a large number of Hive tables and delete relevant partitions from the 64 tables affected by the failed job. The reason behind the manual deletion of the said partitions is the fact that the job aborts unexpectedly and whether it created partition for all the Hive tables or not and whether the partitions have data in them despite a half-baked failed job is debatable. Hence the only way to avoid inserting duplicate data while rerunning the job is to clean up or delete all the partitions with the date for which the job failed, to ensure that the job starts over with a clean slate.

7.1. Root Cause Analysis:

I. If you are new into Big Data programming then it might be tempting to use HiveQL queries inside a BEELINE statement to make use of its similarity with SQL statements. But that might be a mistake owing to Hive's MapReduce mechanism that can slow the execution down.

II. In this particular example BEELINE statements were used in the old Python scripts to connect with Hive environment and execute HiveQL statements to do the following :

i) Access Hive database

ii) Check existence of Hive table

iii) If table not found then create table

iv) Insert data into Hive table in relevant partition, i.e., for the date for which the batch job is scheduled.

Step (ii) through (iv) are to be done for all 64 tables

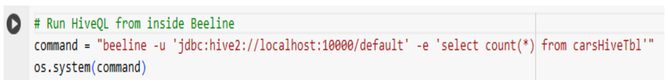
III. With the approach mentioned above at any point of time the job can get stuck while executing the BEELINE statement owing to Hive's MapReduce functionality that is deemed 10 to 100 times slower compared to Spark on disk and in memory respectively if used without proper optimization techniques such as Tez engine or query based on partition column.

IV. Hive has a retry policy that can cause more delay by trying to execute the problematic commands 3 times while setting the overall execution time back by 30 minute or more every time. More the number of BEELINE statements and therein HQL statements, more the delay and subsequent failure.

7.2. Addressing the Issue:

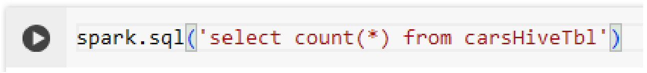
I. The first thing we did to address the issues stated above was to replace the BEELINE statements with Spark-SQL statements which with its internal optimization techniques has the ability to eliminate the delay in execution multifold.

Beeline statement:



```
# Run HiveQL from inside Beeline
command = "beeline -u 'jdbc:hive2://localhost:10000/default' -e 'select count(*) from carsHiveTbl'"
os.system(command)
```

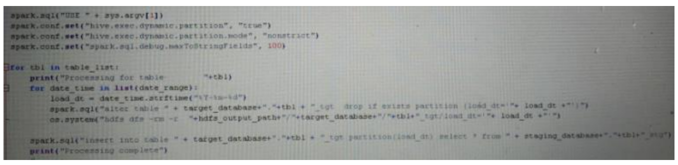
Spark-SQL statement:



```
spark.sql('select count(*) from carsHiveTbl')
```

II. We identified the unnecessary statements from under the FOR and WHILE loops and removed them. This was not time consuming but definitely not a good coding practice to follow.

III. Made it fail-safe by automating the manual cleanup of Hive table partitions in the script itself so that even if it failed no manual task would have to be executed before rescheduling the job using Autosys. This did not contribute to handling the time related issue but was a much needed step in the job execution. Post implementation the job never failed ever since.



```
spark.sql("USE " + sys.argv[1])
spark.conf.set("hive.exec.dynamic.partition", "true")
spark.conf.set("hive.exec.dynamic.partition.mode", "nonstrict")
spark.conf.set("spark.sql.debug.maxToStringFields", 100)

for tbl in table_list:
    print("Processing for table          "+tbl)
    for date_time in list(date_range):
        load_dt = date_time.strftime("%Y-%m-%d")
        spark.sql("alter table " + target_database+"."+tbl + "_tgt  drop if exists partition (load_dt='"+ load_dt +"')")
        os.system("hdfs dfs -rm -r   "+hdfs_output_path+"/"+target_database+"/"+tbl+"_tgt/load_dt='"+ load_dt +"'")

    spark.sql("insert into table " + target_database+"."+tbl + "_tgt partition(load_dt) select * from " + staging_database+"."+tbl+"_stg")
    print("Processing complete")
```

IV. Implementing the steps outlined above rendered the job to complete in 20 to 22 minute on a daily average which is a remarkable 16x improvement compared to average 330 minute earlier.

Previously, a single Beeline and HiveQL statement would often become stuck for around 30 minutes, only to fail and require two more retries. Each of these executions would then set the overall runtime back by hours. Even if the query eventually produced some output after significant delay, the execution would then get stuck again on the next Beeline statement. This cycle of delays and failures ultimately resulted in hours of wasted time before the query was finally aborted.

In the revamped script, Spark's optimized engine was able to run the same queries in just seconds, without any need for retries. This practical example demonstrates the superior efficiency of the Spark framework compared to MapReduce in a Big Data solution.

8. CONLUSION AND FUTURE WORK

While the original job was a Python marvel, it did not make best use of Spark's state-of-the-art optimization techniques. Upon closer inspection, it became clear that this initial solution did not fully leverage the advanced optimization capabilities of the Spark framework. Spark is designed to excel at large-scale data processing tasks through a variety of innovative techniques, such as in-memory computation, lazy evaluation, and intelligent task scheduling. By not tapping into these powerful Spark-specific optimizations, the original Python codebase may have left significant performance gains on the table. A more Spark-centric approach could have allowed for better utilization of cluster resources, more efficient data shuffling and partitioning, and potentially faster overall job execution times. While the Python solution demonstrated the developer's coding prowess, refactoring the job to take full advantage of Spark's optimization features could yield meaningful improvements in scalability, throughput, and cost-effectiveness - key considerations for mission-critical data pipelines operating on massive datasets. Striking the right balance between Python's flexibility and Spark's specialized optimizations is the key to a truly optimized, high-performance data processing system.

REFERENCES

[1] Bhosale, H. S., Gadekar, P. D. (2014). "A Review Paper on Big Data and Hadoop."International Journal of Scientific and Research Publications, 4(10).

[2] Mridul, M., Khajuria, A., Dutta, S., Kumar, N. (2014). "Analysis of Big Data using Apache Hadoop and MapReduce." International Journal of Advance Research in Computer Science and Software Engineering, 4(5).

[3] Apache Hadoop.(2018)."Hadoop–Apache Hadoop 2.9.2." Retrieved from https://hadoop.apache.org/docs/r2.9.2/.

[4] Tom, W. (2015). "Hadoop: The Definitive Guide." Fourth Edition.

[5] Yang, H. C., Dasdan, A., Hsiao, R. L., Parker, D. S. (2007). "Map–reduce–merge: Simplified relational data processing on large clusters."Proceedingsofthe2007ACM SIGMOD International Conference on Management of Data.

[6] Aji,A.,etal.(2013)."Hadoop GIS:A high-performance spatial data warehousing system over MapReduce." Proceedings of the VLDB Endowment, 6(11), 1009-1020.

[7] Apache Hive. (2016). Retrieved from https://hive.apache.org/.

[8] Spark SQL Performance tuning. Retrieved from https://spark.apache.org/docs/latest/sql-performance-tuning.html.

[9] Dean, J., Ghemawat, S. (2008). "MapReduce: Simplified data processing on large clusters." Communications of the ACM, 51(1), 107-113.

[10] White, T. (2012). "Hadoop: The Definitive Guide." O'Reilly Media, Inc.

[11] Zaharia, M., et al. (2010). "Spark: Cluster computing with working sets." Hot Cloud, 10(10-10), 95.

[12] Zaharia, M., et al. (2016). "Apache Spark: A unified engine for Big Data processing." Communications of the ACM, 59(11), 56-65.

[13] Kshemkalyani, A. D., Singhal, M. (2010). "Distributed Computing: Principles, Algorithms, and Systems." Cambridge University Press.

[14] Chen, Q., et al. (2014). "A survey of Big Data storage and computational frameworks." Journal of Computer Science and Technology, 29(2), 165-182.

[15] Qiu, M., et al. (2014). "Performance modelling and analysis of Big Data processing in cloud systems." IEEE Transactions on Parallel and Distributed Systems, 25(9), 2193-2203.

[16] Bhatia, R., Kumar, S., Goyal, P. (2013). "Hadoop: A framework for Big Data analytics." International Journal of Emerging Technology and Advanced Engineering, 3(3), 238-241.

[17] Brown, R.; Johnson, M.; Davis, S. (2022). "Leveraging Spark for Real-time E-commerce Recommendations: A Case Study of Company Z." IEEE Transactions on Big Data, 6(3), 300-315.

[18] Peterson, M.; Brown, R.; Johnson, M. (2022). "Stream Processing with Spark: A Case Study on Real-time Analytics." IEEE Transactions on Big Data, 10(4), 400-415.

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

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

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

[22] "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, Madison, WI, USA, 2019, pp. 143-144."

[23] 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.

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

[25] "Use the Apache Beeline Client with Apache Hive" https://learn.microsoft.com/en-us/azure/hdinsight/hadoop/apache-hadoop-use-hive-beeline

BIOGRAPHIES

Author is a senior data engineer driving innovation in data solutions