GPU Offloading in Spark Rapids: Investigating Compression, Batching, Filtering, and Reordering

Nikhil Pavan Kanaka
kanaka.3@osu.edu

***Abstract*—This project aims to optimize data movement between the CPU and GPU in Spark by investigating various techniques and evaluating their effectiveness under different scenarios. The primary objective is to identify the optimal configuration of compression, batching, data filtering, and reordering techniques to minimize the total data transfer time while maintaining performance, based on the specific use case. To achieve this objective, research will be conducted through simulation and experimentation, starting with a comprehensive literature review to identify the most promising techniques. The RAPIDS API will be utilized to implement and evaluate these techniques, and experiments will be conducted using various data sets and workloads to provide a comprehensive evaluation. The expected outcome of the project is to provide valuable insights into optimizing data transfers.**

***Index Terms*—Spark, GPU, Offloading, Rapids**

I. Introduction

Apache Spark [4] is an open-source distributed computing system designed to process and analyze large-scale data sets. It provides a unified analytics engine that supports various data processing workloads. One of the key features that sets Spark apart is its in-memory data processing capability. Unlike traditional disk-based processing frameworks, Spark leverages the power of main memory to store and manipulate data in a distributed fashion. By keeping the data in memory, Spark significantly reduces the data access latency, leading to substantial performance improvements. In recent years, Spark has embraced GPU acceleration to further enhance its performance and scalability. By leveraging the computational power of Graphics Processing Units, Spark can accelerate computations and achieve significant speedups for data-intensive workloads. The integration of GPU acceleration libraries, such as RAPIDS, into Spark has opened up new opportunities for processing big data in main memory and on GPUs. Spark RAPIDS [1] is a collaboration between Apache Spark and NVIDIA that aims to provide high-performance in-memory data processing and analytics on GPUs. It combines the distributed computing capabilities of Spark with the power of GPU acceleration, enabling efficient and scalable processing of big data workloads. Spark RAPIDS' performance superiority over traditional CPU-based Spark implementations can be attributed to its ability to leverage GPU parallelism, enhanced memory bandwidth, optimized algorithms, and seamless integration with Spark. By leveraging the parallel processing capabilities of GPUs and minimizing data movement overhead, Spark RAPIDS aims to achieve faster and more efficient data processing and analytics. Data transfers between the CPU and GPU in Spark RAPIDS occur through a bus, such as PCIe or NVLink, with the CPU sending data to the GPU for processing and receiving the results back. However, these transfers can lead to long transfer times and decreased performance due to the large amounts of data involved. Improving the performance of Spark RAPIDS requires addressing the input/output (IO) bottleneck, which refers to the data transfer between the CPU and GPU. The IO bottleneck can significantly impact performance by causing delays, adding overhead, and hindering system efficiency. Therefore, it is crucial to minimize data movement between the CPU and GPU. To tackle the IO bottleneck, various techniques are employed in Spark RAPIDS, including compression, batching, data filtering, and reordering. Compression reduces data size before the transfer, minimizing transfer time and CPU overhead. Batching combines multiple transfers into a single larger transfer to reduce overhead. Data filtering eliminates unnecessary data prior to transfer, reducing the amount of data transferred. Data reordering optimizes data placement to minimize CPU-GPU movement. By leveraging these techniques, Spark RAPIDS aims to minimize total data transfer time while maintaining high performance.

A. Compression

Compression techniques [2] are employed to reduce the size of the data before transferring it from the CPU to the GPU. By compressing the data, the amount of data to be transferred is reduced, which in turn minimizes the transfer time and CPU overhead. Spark RAPIDS utilizes various compression algorithms to compress the data before it is sent to the GPU. Upon arrival at the GPU, the compressed data is decompressed for further processing.

1) Snappy: Fast and efficient compression algorithm optimized for speed. It achieves compression by replacing repeated sequences of bytes with shorter representations. Snappy offers good compression ratios and fast decompression speeds, making it suitable for scenarios where a balance between compression efficiency and speed is required.

2) Gzip: Widely used compression algorithm that provides high compression ratios. It utilizes the DEFLATE algorithm, which combines LZ77-based compression with Huffman coding. Gzip achieves higher compression ratios than Snappy but can have slower compression and decompression speeds.

3) LZO: Compression algorithm designed for high-speed data compression and decompression. It emphasizes decompression speed over compression ratios. LZO achieves fast

speeds by using a dictionary-based compression approach combined with byte-level encoding.

4) Brotli: Modern compression algorithm developed by Google. It offers high compression ratios and is particularly effective for compressing text-based data. Brotli utilizes a combination of LZ77-based compression, context modeling, and Huffman coding techniques.

5) LZ4: Compression algorithm optimized for high-speed compression and decompression. It provides fast compression and decompression speeds, making it suitable for scenarios where real-time or low-latency processing is required. LZ4 achieves compression by using a sliding window and dictionary-based approach.

6) Zstd: Compression algorithm developed by Facebook. It offers high compression ratios and provides a wide range of compression settings to balance between speed and efficiency. Zstd leverages a combination of entropy coding, dictionary compression, and pattern matching techniques.

B. Batching

Batching involves combining multiple data transfers into a single, larger transfer. Instead of performing separate transfers for each data item, Spark groups multiple data transfers together to create a batch. By batching the data transfers, the overhead associated with each individual transfer is significantly reduced, resulting in improved transfer efficiency.

1) Batch Processing: In batch processing, data is processed in fixed-sized batches or partitions. Spark's core processing engine supports batch processing by default. Each batch is treated as a separate unit of work, and the data is processed in parallel across multiple nodes or cores.

2) Micro-batching: Micro-batching is a technique used in Spark Structured Streaming, which combines the benefits of both batch and streaming processing. Data is divided into small, fixed-sized micro-batches, and each micro-batch is processed as a batch job. This approach allows for near-real-time processing with low-latency and fault-tolerance characteristics. Micro-batching provides a compromise between true real-time streaming and batch processing, enabling processing guarantees and easier integration with existing batch workflows.

3) Windowed Batching: Windowed batching is used in time-series analysis and event-based processing. It involves dividing the data into fixed-size or time-based windows. Spark provides window functions and operations that allow you to define the size and sliding behavior of the windows. By processing data within windows, you can perform batched operations on a window of data rather than individual records. This technique enables computations over specific time periods or data segments, such as aggregations or calculations within a sliding window.

4) Adaptive Batching: Adaptive batching is a dynamic batching technique where the batch size is adjusted based on run-time conditions or workload characteristics. Although not explicitly provided as a built-in feature in Spark, adaptive batching can be implemented using custom logic within the application. For example, you can dynamically adjust the batch size based on the available system resources, input data rate, or computational complexity. This approach allows for optimizing the balance between throughput and latency based on the current workload.

C. Data Filtering

Data filtering aims to eliminate unnecessary data before sending it to the GPU. Spark provides APIs that allow users to filter out irrelevant or redundant data before transferring it. By applying filtering operations such as filter, select, or drop, the amount of data that needs to be transferred between the CPU and GPU can be reduced. This reduction in data size leads to faster transfer times and improved overall performance.

1) Batch Processing: In batch processing, Spark provides a rich set of filtering capabilities. The most commonly used technique is the `filter()` transformation, which allows you to specify a predicate function that defines the filtering condition. The `filter()` transformation applies the predicate to each record in the data-set and retains only the records that satisfy the condition. Spark's filtering capabilities include various comparison operations, logical operations, string operations, and more. These operations can be combined to create complex filtering conditions for processing batch data.

2) Structured Streaming: In Spark Structured Streaming, filtering is an integral part of the streaming data processing pipeline. Similar to batch processing, you can use the `filter()` transformation to apply filtering conditions on the streaming data. This allows you to select or exclude specific records based on criteria defined by the predicate. The filtering conditions can be static or dynamic, depending on the requirements of the streaming application. Filtering in Structured Streaming can be performed on a continuous stream of data, enabling real-time data reduction and extraction.

3) Windowed Filtering: Windowed filtering is a technique used for processing data within specific time-based windows. Spark provides window operations, such as `window()` or `rangeBetween()`, which allow you to define time-based or row-based windows for filtering. These operations enable you to filter data within a sliding window, tumbling window, or custom-defined window. Windowed filtering is particularly useful for time-series analysis, event-based processing, or any scenario where data needs to be filtered within specific time intervals or data segments.

4) Partition-Level Filtering: Partition-level filtering is a technique used to optimize data processing by selectively reading only relevant data partitions. Spark's partition pruning feature leverages filtering conditions to determine which partitions of data need to be accessed during the processing. By evaluating the filtering condition against partition metadata or statistics, Spark can skip reading unnecessary data partitions, thereby reducing disk I/O and improving performance. Partition-level filtering is especially beneficial in scenarios where the data is partitioned based on specific criteria, such as date, location, or any other relevant attribute.

D. Data Reordering

Data reordering techniques optimize the order of data transfers between the CPU and GPU to minimize data movement. By rearranging the data before transferring it to the GPU, Spark aims to reduce unnecessary data shuffling and movement during processing. Techniques such as sorting, partitioning, or shuffling can be employed to optimize the data layout, ensuring that the data is transferred and processed efficiently on the GPU. This minimizes the time spent on data movement and enhances the overall transfer time.

1) Batch Processing: In batch processing, data reordering techniques focus on optimizing the data layout to improve data locality and minimize data shuffling during parallel processing. Techniques like "`partitioning`" or "`sorting`" can be applied to group related data together, ensuring that data residing on the same partition is processed together. Partitioning can be based on key-value pairs, range-based partitions, or custom partitioning functions. Sorting the data based on a specific attribute can also help in certain scenarios, such as optimizing join operations.

2) Streaming and Structured Streaming: In streaming applications, data reordering is often employed in windowed operations. The data is partitioned or sorted based on the window boundaries to ensure that all the relevant data for a particular window is processed together. This technique reduces the need for shuffling data across partitions during window computations, improving overall efficiency. Additionally, Spark provides operations like re-partition and coalesce that can be used to redistribute data based on specific criteria, facilitating better data locality and reducing unnecessary data movement.

3) Graph Processing: In graph processing, data reordering techniques are used to optimize the graph structure and facilitate efficient graph traversals and computations. Techniques like "`graph partitioning`" aim to divide the graph into smaller sub-graphs or partitions to enable parallel processing. Partitioning is typically based on node properties or graph topology, ensuring that related nodes are grouped together. This improves locality and minimizes communication overhead during graph algorithms like PageRank or connected component analysis.

4) Machine Learning: Data reordering in machine learning applications focuses on optimizing the data layout for better training and prediction performance. Techniques such as "`feature reordering`" or "`data sampling`" can be employed. Feature reordering arranges the features in a specific order based on their importance or correlation, enabling faster convergence of machine learning algorithms. Data sampling techniques reorganize the data to achieve a better balance between classes or improve data distribution, leading to more representative training sets.

By leveraging compression, batching, data filtering, and data reordering techniques, Spark maximizes the efficiency of CPU-GPU data transfers. These techniques collectively help reduce data size, minimize transfer overhead, eliminate irrelevant data, and optimize the order of data transfers, resulting in improved performance and faster data processing on the GPU.

II. Compression

Compression algorithms play a vital role in reducing data movement costs. Spark's MapReduce [3] supports various compression and decompression algorithms called codecs. These algorithms can be splittable or non-splittable. Splittable algorithms split files into compressed and uncompressed data blocks of fixed size, allowing individual decompression. Non-splittable algorithms require serial decompression, resulting in longer decompression time.

The choice of compression algorithm depends on factors like data quality, codec schemas, data type, and application type. The compression ratio, which is the ratio of compressed data to uncompressed data size, determines the degree of data size reduction and I/O usage. Lower compression ratios indicate less memory and I/O usage. Compression can be implemented at different stages, including input data, intermediate Map output data, and Reduce output data. Intermediate compression of the map output reduces network usage during the Mapreduce shuffle step. All nodes begin to communicate with each other and collect the map output as the phase reduces input. If the input or intermediate output of the map phase is compressed, the framework chooses a decompression algorithm before processing according to the file extension, refer to Table I.

TABLE I
A summary of compression formats

Compression format	Splittable
gzip	No
bzip2	Yes
snappy	Yes (container file formats)
Lzo	Yes (indexing algorithm)
lz4	Yes (4MC library)
zstandard	Yes (4MC library)

The compression codecs used in the data storage are all lossless, ensuring data integrity. The gzip and deflate codecs employ the deflate algorithm, which combines lz77 and Huffman Coding. The lz77 algorithm replaces duplicate bit positions based on their previous occurrences. Gzip and deflate differ in the Huffman encoding phase. The splittable compression bzip2 codec utilizes the Burrows-Wheeler text compression and Huffman coding algorithms. It compresses data blocks independently and allows parallel compression. Snappy, a fast compression library, incorporates concepts from lz77. Snappy blocks are non-splittable, but the files within them can be split. The lzo compression algorithm is a variation of lz77 and involves finding matches, writing unmatched literal data, determining match lengths, and writing match tokens. The lz4 algorithm represents compressed data files as LZ4 sequences containing tokens, literal lengths, offsets, and match lengths. Zstandard, is an lz77-based algorithm that supports dictionaries, employs finite-state entropy coding and Huffman coding for entropy coding steps.

1) Evaluation with TPC-DS: To evaluate and compare the performance of different compression techniques in Spark, the TPC-DS benchmarking approach is utilized. The TPC-DS benchmark is a widely recognized standard for testing the performance of big data processing systems. It consists of a set of queries that simulate real-world business intelligence and decision support workloads. For the benchmarking process, shuffle intensive queries are selected from the TPC-DS benchmark suite. These queries involve a significant amount of data shuffling, which can be particularly challenging for compression techniques. By focusing on shuffle intensive queries, the benchmarking aims to assess the effectiveness of compression techniques in scenarios where data movement and transfer play a crucial role. To ensure comprehensive evaluation, the benchmarking is performed for different combinations of datasets. The datasets used in the benchmarking process vary in size and complexity, reflecting realistic data processing scenarios. By running the benchmarking on diverse datasets, the impact of data size and computational complexity on the performance of compression techniques can be observed. The benchmarking results are presented using a graph 1that illustrates the average timelines. The timelines are represented on a normalized scale ranging from 0 to 10. Higher values on the graph indicate higher compression ratios and faster compression speeds. This representation enables a visual comparison of the different compression techniques based on their performance characteristics.

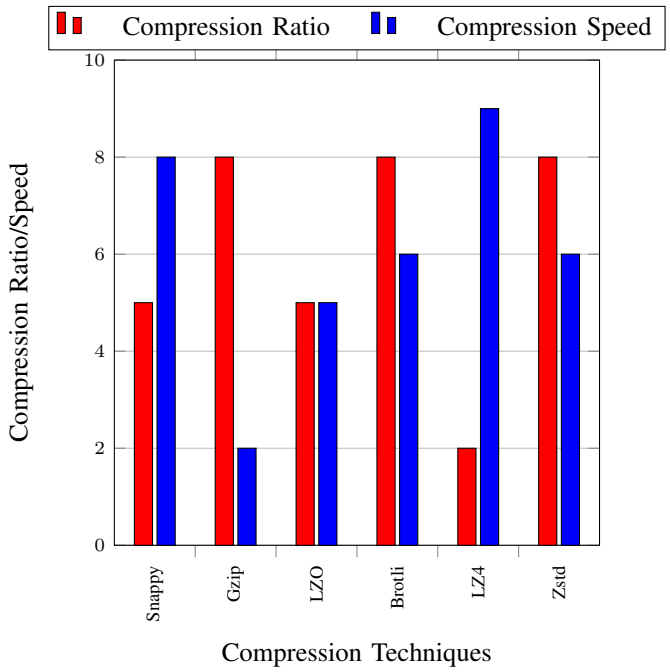

Fig. 1. Comparison of Compression Techniques

Based on the results, Snappy and LZO demonstrate a moderate compression ratio and fast compression speed. They provide a good balance between compression efficiency and speed. Gzip and Brotli achieve a high compression ratio but at the cost of slower compression speeds. LZ4, on the other hand, achieves a low compression ratio but operates at a very fast speed. Zstd achieves a high compression ratio while maintaining a balanced compression speed.

These results highlight the trade-offs between compression ratio and compression speed. Depending on the specific requirements of a given application, different compression techniques can be selected to optimize either compression efficiency or processing speed. It's important to note that the suitability of a compression technique depends on factors such as the nature of the data, available system resources, and specific use case requirements. Therefore, it is recommended to carefully evaluate the trade-offs and choose the compression technique that best aligns with the desired balance of compression ratio and speed for a particular scenario.

III. Batching

Modifying parameters associated with batching in Spark, such as the batch size or trigger interval, yields diverse impacts on the performance and behavior of your Spark application.

A. Batch Size

A larger batch size can lead to increased memory consumption as more data is processed together. It can also result in higher processing latency because a larger amount of data needs to be processed before results are available. However, it may improve throughput if the system has enough resources to handle larger batches efficiently.

A smaller batch size reduces memory usage and can result in lower processing latency as smaller amounts of data are processed at a time. It may be beneficial for applications with strict low-latency requirements. However, frequent batch generation can add overhead due to increased scheduling and coordination.

B. Min Partitions

Setting a higher number of partitions can increase parallelism and distribute the workload across more resources. It can improve processing efficiency, especially when working with large datasets. However, it may result in higher memory consumption and increased scheduling overhead. Setting a lower number of partitions reduces memory usage and scheduling overhead. It can be beneficial for small datasets or applications with limited resources. However, it may limit the level of parallelism and processing efficiency, especially for large-scale data processing.

From Fig.2 (where batch size and execution time and represented on a normalized scale), When the min partitions value is low, represented by the blue line, the execution time initially decreases as the batch size increases. This is because smaller batch sizes allow for faster processing of each partition. However, after a certain point, further increasing the batch size leads to diminishing returns, and the execution time starts to plateau or even increase slightly. This is because larger batch sizes result in increased overhead and resource consumption.

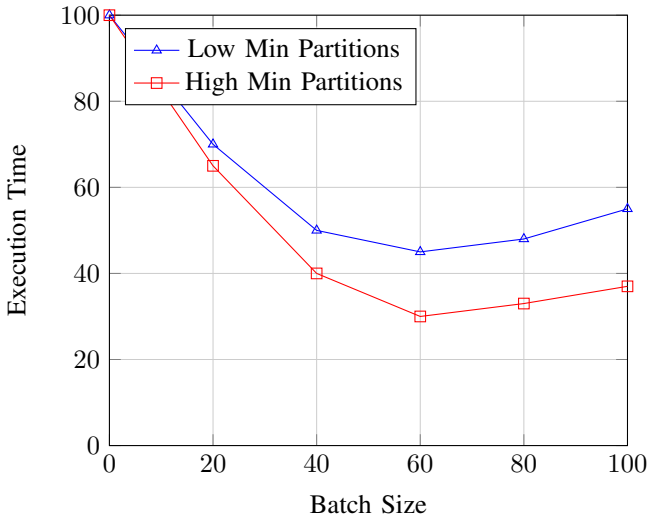

Fig. 2. Trade-off between Batch Size and Min Partitions

Comparing the low min partitions and the high min partitions, we can see that the red line consistently performs better in terms of execution time. This is because higher min partitions allow for better parallelism and workload distribution across the available resources, leading to more efficient processing.

C. Structured Streaming

In Structured Streaming, there are specific parameters that are applicable and used for configuring the behavior of jobs.

1) Trigger Interval: A longer trigger interval means fewer micro-batches are generated over time. This reduces the frequency of processing and may lead to higher processing latency. It can be useful when dealing with slower data streams or when there is no strict real-time processing requirement. A shorter trigger interval increases the frequency of batch generation and reduces processing latency. It allows for near-real-time processing with low-latency characteristics. However, it may put more pressure on system resources and increase scheduling overhead.

2) Max Records Per Trigger: Allowing a higher number of records per trigger can increase the batch size and potentially improve processing efficiency. It reduces the overhead of initiating new micro-batches. However, it may increase processing latency as more data needs to be processed before results are available. Limiting the number of records per trigger reduces the batch size and can lead to lower processing latency. It can be beneficial for applications that require faster processing or have memory constraints. However, frequent micro-batch generation may introduce additional scheduling overhead.

3) Max Files Per Trigger: Allowing a higher number of files per trigger increases the batch size and reduces the overhead of processing individual files. It can improve processing efficiency, especially when dealing with large files. However, it may increase processing latency as more data needs to be processed before results are available. Limiting the number of files per trigger reduces the batch size and can lead to lower processing latency. It can be useful when dealing with small files or applications that require faster processing. However, it may introduce additional overhead due to more frequent micro-batch generation.

The impact of configuring these parameters to low or high values depends on the specific workload, data characteristics, and system resources. It's important to consider the trade-offs between latency, throughput, and resource utilization when adjusting these parameters. Additionally, it's worth noting that configuring these parameters too low or too high can lead to resource contention, out-of-memory errors, or reduced processing efficiency. Therefore, it's essential to monitor system resources, such as memory and CPU utilization, and adjust the parameters accordingly to ensure optimal performance and stability of your Spark application.

IV. Data Filtering

Data filtering plays a crucial role in optimizing data processing by eliminating irrelevant or redundant data. Parameters such as filter conditions and operations like select or drop enable users to define specific criteria for data filtering.

A. Predicate Pushdown

Allows Spark to push filtering predicates to the underlying data sources, such as Parquet or ORC files. This means that filtering operations can be performed directly within the data source, reducing the amount of data that needs to be read into Spark. Predicate pushdown depends on the capabilities of the underlying data source. Not all data sources may support predicate pushdown, so it might not always be applicable. Additionally, pushing down predicates may result in increased complexity in terms of managing and optimizing queries, especially when working with multiple data sources.

B. Filter Pushdown

Specifically applies to Parquet and ORC files in Spark. It enables Spark to push filtering predicates down to the data source during the data read process, reducing the amount of data that needs to be read into Spark for further processing. Filter pushdown is similar to predicate pushdown, where it depends on the capabilities and compatibility of the underlying data source. Not all data sources may support filter pushdown, and enabling it may require additional configuration or setup steps.

C. Partition Pruning

Technique that leverages filtering conditions to skip unnecessary data partitions during query execution. Spark evaluates the filtering conditions against partition metadata or statistics to determine which partitions need to be accessed, reducing the amount of data read from disk and improving query performance. Partition pruning works well when data is properly partitioned and statistics are collected and up-to-date. However, if partition metadata or statistics are not accurate or unavailable, partition pruning may not be effective.

Additionally, partition pruning adds an extra overhead during query planning and execution, as Spark needs to evaluate and process partition information.

D. Data Source Optimizations

Configuring specific properties or options provided by the underlying data source to optimize filtering operations. This could include options like `pushdownPredicate` in JDBC connections or specific configuration properties for other data sources. Some data sources may provide extensive optimization options, while others may have limited support. It requires understanding the data source's capabilities and experimenting with different configurations to achieve the desired performance improvements.

E. Caching

Caching allows you to persist a DataFrame or RDD in memory, which can improve the performance of repetitive or iterative operations. When a cached dataset is accessed, Spark can read the data directly from memory instead of recomputing it, resulting in faster processing. Caching requires sufficient memory resources to store the cached data. If the dataset is too large to fit in memory, caching may lead to increased memory pressure and potential performance degradation.

When multiple operations are performed on the same intermediate data, caching allows for quick retrieval of the next result. If the intermediate data is not cached or a failure occurs on a node holding the cache, Spark reconstructs the data using lineage, which records the procedure for generating the data. The reconstruction task only instantiates the partition of the lost cache.

Intermediate data can be explicitly stored in memory or disk. When stored on disk, the data is serialized, with the default serialization method being Java serialization. However, Java serialization can be slow and result in large serialization formats for many classes. Alternatively, the faster Kryo serialization can be chosen, but it requires registering the classes to be serialized in the user's program.

The storage level determines whether the data is stored in memory-only, disk-only, or a combination of both. By default, RDDs are cached in memory-only, while dataFrames are cached in memory and disk. The Executor's memory is divided into ExecutionMemory and StorageMemory by the MemoryManager. ExecutionMemory is used for processing intermediate data, while StorageMemory, managed by the BlockManager, stores the intermediate data partitions as blocks. This means that StorageMemory is utilized when caching intermediate data in memory.

Several storage levels can be specified for intermediate data caching, each with its own behavior:

1) NONE (NOCACHE): Does not cache intermediate data at all. The data is discarded immediately after use, requiring its regeneration when referenced again.

2) MEMORY ONLY: Holds the intermediate data cache in memory-only. When caching is performed, the BlockManager reserves a required amount of memory in StorageMemory through the MemoryStore interface and saves each partition of the intermediate data as a block. There is also a variant called MEMORY ONLY SER, which serializes the data to occupy less memory space.

3) OFF HEAP: Stores the intermediate data cache in off-heap memory. The partitions are serialized in memory by the BlockManager and stored in off-heap memory via the MemoryStore interface. This approach reduces the impact of garbage collection.

4) DISK ONLY: Stores the intermediate data cache on a local disk. The BlockManager serializes the partitions in memory and stores them using the DiskStore interface.

5) MEMORY AND DISK: Combines memory and disk for caching intermediate data. It attempts to use memory primarily and stores the data as blocks in StorageMemory. However, if the data does not fit in memory, the BlockManager moves some old blocks to disk, following the Least Recently Used (LRU) algorithm, to free up space for new blocks. When accessing the intermediate data stored on disk, the BlockManager checks the availability of space in StorageMemory, and if possible, it reads the blocks from disk and puts them back into StorageMemory. The key advantage of this approach is that it efficiently utilizes both memory and disk without requiring users to explicitly choose one. By leveraging intermediate data caching and selecting an appropriate storage level, Spark can enhance performance, reduce recomputation, and provide fault tolerance in case of failures.

Points to consider while selecting the caching strategy and storage level based on specific requirements [5]:

1) Performance degradation occurs when using disk in intermediate data caching, but the performance of disk is less likely to be a bottleneck due to the buffer cache of the operating system. Serialization is often the main bottleneck, especially with RDDs and the standard Java serialization.
2) When using memory-only caching, RDDs are faster than DataFrames because RDDs skip the serialization process, while DataFrames perform encoding. However, when storing the cache on disk or off-heap memory, encoding of DataFrames is faster than serialization of RDDs.
3) Disk performance becomes evident when many tasks are executed in parallel on each worker node, simultaneously performing caching operations. HDDs with larger capacity are often more cost-effective in such scenarios. Improvements in Spark's serialization performance are highly anticipated.
4) DataFrame requires more execution memory than RDD for encoding and decoding operations, as observed in a benchmark failure due to out-of-memory (OOM) error in the same environment.
5) Memory and disk mode, which selectively uses both memory and disk, may have poor performance compared to disk-only caching when memory capacity is insufficient. However, the issue can be mitigated by suppressing re-caching. An opportunity exists to develop

a more intelligent algorithm that returns blocks to memory when more free memory becomes available during computation.

6) Although off-heap memory reduces the impact of garbage collection (GC), it does not necessarily outperform disk-only caching due to serialization overhead. Further investigation is required to assess the influence of GC algorithms and parameters.

These conclusions highlight the importance of carefully selecting the caching strategy and storage level based on specific requirements, available resources, and the characteristics of the data being processed.

V. Conclusion

Compression, Batching, Filtering, and Reordering are crucial techniques in Spark that contribute to improved performance and efficient data processing. The best configuration for a Spark job depends on the specific requirements and characteristics of the data and application. It is important to consider factors such as data size, processing needs, hardware capabilities, and trade-offs between latency and throughput. Experimentation and tuning based on the specific use case are necessary to determine the optimal configuration for achieving the desired performance and efficiency goals.

VI. Future Work

By conducting in-depth analyses and experiments, researchers and practitioners can uncover the most effective combinations of parameters and strategies for each technique, taking into account various factors such as data characteristics, workload patterns, and resource constraints. This knowledge can contribute to the development of guidelines and best practices for Spark job configuration, enabling users to achieve the highest possible performance in their data processing tasks. Additionally, advancements in Spark itself, along with the continuous evolution of hardware and distributed computing technologies, may provide new opportunities for refining and fine-tuning these techniques to achieve even better performance gains in the future.

References

[1] https://nvidia.github.io/spark-rapids/.

[2] https://spark.apache.org/docs/2.4.3/sql-data-sources-parquet.html.

[3] Mohd Rehan Ghazi and Durgaprasad Gangodkar. Hadoop, mapreduce and hdfs: a developers perspective. *Procedia Computer Science*, 48:45–50, 2015.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[5] Kaihui Zhang, Yusuke Tanimura, Hidemoto Nakada, and Hirotaka Ogawa. Understanding and improving disk-based intermediate data caching in spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2508–2517. IEEE, 2017.