Materializing Intermediate Results for in-Memory Data Processing

Karim Maatouk, Ledia Isaj

Abstract— **Query optimization and view materialization have been studied in the database research community. The era of big data brings new opportunities, as well as new challenges. Challenges that exist in traditional view materialization also apply in the context of Spark. However, data processing paradigms should be taken into consideration to extend performance optimization algorithms. Spark provides caching capabilities in a non-automated fashion. Determining what to cache might be a difficult task since there can be complex dependencies between tasks. Moreover, cache misses occurring not only because an RDD is not explicitly cached by the developer, but also because it was cached and later evicted due to LRU policy. Deciding what to cache can be an NP-hard problem considering the storage capacity, set of jobs to be executed, size of the result. What is more, the sequence of jobs within a given time and their frequency might be priori unknown. Therefore, adaptive solutions are needed to decide what to cache and adapt in time. In this article we are going to discuss the in-memory data processing in Spark, study Spark's caching mechanism, point out the challenges, and explore different solutions so far.**

Keywords— **Intermediate Data Overlapping, Cache Optimization, Spark, Multi-query optimization**

I. Introduction

The era of big data includes masses of unstructured data that need more real-time analysis. Big data brings new opportunities for discovering new values, helps us to gain an in-depth understanding of the hidden values, but also brings new challenges. [9] Efforts are always put to gain more performance and reduce costs regarding different parameters. Materializing intermediate results aims to optimize data-intensive flow (DIF) execution. Parts of the flow are materialized to avoid costly recomputations. Data processing paradigms should be taken into consideration to extend performance optimization algorithms. Our goal is to focus on the in-memory data processing in Spark, study Spark's caching mechanism, point out the challenges, and explore different solutions so far.

Efficient cache management plays a crucial role in in-memory data analytics. Existing caching policies are based on historical information (e.g., frequency and recency). Spark, the BlockManager uses Least Recently Used (LRU) approach to determine which data block to evict when memory capacity is full. Jobs executed in such distributed environments often have significant computational overlaps and exploiting these overlaps would result in a huge performance gain. Moreover, Spark offers caching capabilities in a non-automated fashion. Storage strategies are chosen by the user, and the expected performance is full of uncertainties. Poor storage strategies not only lower the efficiency of the program but also lead to errors. When useless intermediate results are kept, it is likely to waste memory and degrade the executing speed. Meanwhile, this situation would miss the useful intermediate results which will be reused in the subsequent computations. [4] [5] [6]

For these reasons, we are motivated to explore the solutions introduced for caching decisions and memory performance improvement. In the first section, we will give a brief introduction. In the second,

a short overview of how Apache Spark works is given, including in-memory data processing and caching mechanisms. Moreover, some of the current challenges that were in the focus of this paper were introduced. Later we discuss traditional materialization in the context of Spark. In section four we explore different solutions to tackle different challenges and give a quick comparison between them. In section five, we are going to explain the low-level materialization solution as an alternative approach. In section six, we give an overview of multi-query optimization. In the end, we summarized what was the focus of this paper, issues tacked, and solutions for them, regarding materializing intermediate results in Spark.

II. Apache Spark

Apache Spark is an analytics engine that is used for large-scale data processing. It provides an innovative model of parallel computing and is based on lazily distributed datasets, called RDDs. [1] Each RDD is characterized by five main properties :

- RDD is an abstraction in Spark and it contains a list of partitions, which are distributed on different nodes.
- A function for computing each split.
- A list of dependencies on other RDDs. The DAGScheduler forms a Directed Acyclic Graph (DAG) of stages for each job according to the lineage.
- Optionally, a partitioner for key-value RDDs.
- Optionally, a list of preferred locations to compute each split. [14] [6]

All operations in Spark are based on RDDs. There are two kinds of operation: transformation operations (e.g., map, join, groupBy, and filter) and action operations (e.g., reduce, count, collect, and save). During the transformation operation process, RDD is transformed into different RDDs, and this process is not executed until its action operation happened. Sparks makes it abstract for the programmers, but knowing the implementation helps to write more performant code. In the lazy paradigm of Spark, the application does not do anything until an action is called. At this moment, the Spark Scheduler builds the execution graph and launches the Spark job. Each step consists of stages, and each of them is formed by a collection of tasks that represent each parallel computation performed on the executors. RDD dependencies are used to build a Directed Acyclic Graph (a DAG) for each Spark job. The DAG forms a graph of stages for each job, allocated the location for each task to run, TaskScheduler gets this information, and is responsible for running these tasks and creates a graph with dependencies between partitions. So Spark keeps track of how the RDD is partitioned and makes sure not to partition the same RDD by the same partitioner twice. [2] [3]

A. Apache Spark in-memory data processing and caching

The lazy evaluation also enables Spark to combine operations that do not need communication with the driver. Comparing Spark with Map Reduce, Spark is much more performant in use cases that involve repeated computations. Performance is highly increased due to in-memory persistence. Rather than going back and forth to the disk, Spark has the option of keeping the data in memory on the executors. There are three options for memory management:in-memory as deserialized data, in-memory as serialized data, and on disk. Storing the data in memory as deserialized Java objects are the fastest way since it reduces serialization time. However, might not be so efficient since the data is stored as objects. You can also serialize the data into a stream of bytes. This approach might be a bit slower, but is often more memory efficient. Lastly, RDDs that have very large partitions, can be written to disk. On

the one hand, this strategy is slower for repeated computations. On the other hand, it can be more fault-tolerant for long sequences of transformations. The programmer decides if it is needed to reuse the RDD, and there exist three primary operations for this: cache, persist, and checkpoint. Caching and persisting are more used to avoid recomputation. A checkpoint is used to prevent recomputations due to failures. The saved RDD stays in memory throughout the Spark application unless you define the unpersist function. If the executors run out of memory, Spark uses Least Recently Used (LRU) caching to determine which partition to flush [2], [3].

B. Caching Challenges

It is normal to exist computational overlaps in Spark jobs. The tasks would be much more performant if proper caching mechanisms are applied to avoid computational overlaps. Spark provides caching capabilities in a non-automated fashion. Therefore it is the responsibility of the developer to handle this. However, to determine what to cache might be a difficult task, since there can be complex dependencies between tasks. Moreover, as explored in the previous section, cache misses occurring not only because an RDD is not explicitly cached by the developer, but also because it was cached and later evicted due to LRU policy. So to determine what to cache can be very difficult. It is an NP-hard problem considering the storage capacity, set of jobs to be executed, size of the result. What is more, the sequence of jobs within a given time and their frequency might be priori unknown. Therefore, adaptive solutions are needed to decide what to cache and adapt in time.[4]

III. Traditional Materialization in the context of Spark

Query optimization and View Materialization have been studied extensively in the database research community and the challenges and innovation in this field can also be extended into the Spark framework by mapping the concept of the database to Spark's Resilient Data Sets (RDDs). This section introduces a survey of View Materialization and its challenges and reflects on it in the context of Apache Spark. Chirkova and Yang define view materialization as "a natural embodiment of the ideas of precomputation and caching in databases." They continue, "Instead of computing a query from scratch from base data, a database system can use results that have already been computed, stored, and maintained." [10] Likewise, data processing in Spark makes use of similar concepts of materialization or persistence of data to speed up operations and optimize their performance. Yang et al. [11] define four factors that affect choosing a suitable materialization plan which includes:

- Frequency of query access
- Frequency of updates
- Cost of reformulation from the materialized view
- Cost of maintenance of the materialized view

These set of factors can be mapped in the spark framework as follows:

- Frequency of execution of an operation on an RDD
- Frequency of update of the materialized operation
- Cost of processing operation from a materialized one
- Cost of maintenance of the materialized RDD

Materialization is available in Spark through cache and persist operations which can save to either memory or disk or a combination of both. The

options to materialize in Spark are available through high-level programming API to developers. Challenges that exist in traditional view materialization also apply in the context of Spark such as "How to analyze and handle more complicated queries such as a query with aggregation functions." [11] In addition to how to choose which intermediate results to materialize so that the overall cost is minimized[B]. Yang et al. [11] motivate the need for a global materialized view and multiple query processing by the benefit of a shared intermediate result for multiple queries. Likewise, the concepts, again, can be projected onto the Spark Framework motivating the need for shared materialized RDDs to speed up the processing if multiple operations.

IV. Solutions introduced for caching decisions and memory performance improvement

A. Adaptive algorithms for Intermediate Data Caching Optimization for Multi-Stage and Parallel Big Data Frameworks

Yang, Zhengyu, et al proposed an optimization framework and introduced an adaptive cache algorithm to optimize caching mechanism. According to their simulation, the proposed algorithm could improve the performance by reducing 12 % of the total work to recompute RDDs.

1) An Adaptive Algorithm with Optimality Guarantees: First, they introduced an adaptive algorithm that converges to caching decisions without any prior knowledge of job arrival rates λ_G. They partitioned time into periods of equal length to collect statistics for different RDDs. They also keep as state information the marginals $y_v \epsilon [0,1]^{|v|}$. Each y_v captures the probability that node $v \in V$ is cached. When the period ends, the state vector is adapted $y = [y_v]_v \epsilon_V \epsilon [0,1]^{|v|}$ and the contents of the cache are reshuffled. This algorithms projects gradient ascent over a concave function L, keeping each time a fractional $y \in [0.1]^{|V|}$ that captures the probability with which each RDD should be placed in the cache (V are the nodes in the DAG graph). The information from executed jobs is used to estimate the gradient $\Delta L(y)$, and the probabilities y are adapted. Based on these probabilities, they constructed a randomized placement x that satisfies the capacity constraint.

2) A Heuristic Adaptive Algorithm: A Heuristic Adaptive Algorithm This algorithm suggests that computation should be cached if it is requested often, caching it reduces computation significantly and has a small size. For each job, the algorithm keeps a moving average of the request rate of individual nodes and the cost if not cached. The jobs with a high value of this average are placed in the cache. This algorithm is adaptive, when iterating RDDs the temporal cost is calculated and stored. [4]

B. Least Reference Count (LRC)

Yu, Yinghao, et al. introduced the Least Reference Count approach that evicts the cached data blocks whose reference count is the smallest, where the reference count is the number of dependent child blocks that have not been computed yet. The solution they provided was a pluggable cache manager in Spark. According to their experimental results, LRC achieves the same performance with 40% of cache space compared to LRU. When using the same cache space, the reduced runtime of the application was up to 60%. There are other advantages to this solution. Firstly, it can timely detect inactive data blocks with zero reference count. Moreover, reference count can serve as an indicator of the probability of future data access. Lastly, reference count can be accurately tracked at runtime with minimal overhead, making LRC a lightweight solution for all DAG-based systems. [5]

C. Weighted replacement algorithm

Duan, Mingxing, et al proposed another heuristic approach. A selection algorithm was introduced to choose the most valuable RDDs according to DAG before tasks are executed. This speeds up iterative computations. The algorithm first checks whether an RDD has already been in memory. Secondly, a reasonable RDD is selected according to its number of uses, while the sizes of the whole RDD partitions should be less than the free memory which is used to cache partitions. However, if a lot of RDDs are chosen and the memory capacity is full, Spark uses the LRU replacement algorithm, which only considered if the RDDs are recently used or not. They also proposed the weight replacement (WR) algorithm, which takes into consideration the partitions computation cost, the number of uses for partitions, and the sizes of the partitions. If different partitions have the same computing cost while the frequency of use of one of them is higher, then its weight is the largest. Considering these factors, reasonable partitions are selected to be replaced, which may speed up the process of Spark computation. According to their experiment results, Spark with the WR algorithm shows better performance. [6]

D. Fairride: Near-optimal, fair cache sharing

Pu, Qifan, et al. tackled the problem of cache allocation in a multi-user environment. They argue that most cache management algorithms like LRU and LFU focus on global efficiency between multiple users that access shared files. They showed that with data sharing it is impossible to find an allocation policy that provides isolation-guarantee, strategy-proofness, and Pareto-efficiency simultaneously. They proposed FairRide as a new policy that provides isolation-guarantee (so a user gets better performance than on isolated cache) and strategy proofness (so users are not incentivized to cheat), by blocking access from cheating users. Also, Pareto-efficiency is near-optimal. LRU provides Pareto efficiency, but not isolation guarantee and strategy proofness. FairRide can outperform previous policies when users cheat. Based on the appealing properties and relatively small overhead, they believe that FairRide can be a practical policy for real-world cloud environments. [7]

E. Piglet

Hagedorn, Stefan, and Kai-Uwe Sattler presented an approach for a cost-based decision model to speed up the execution of dataflow programs. This was achieved by merging jobs and reusing intermediate results across multiple executions of the same or different jobs. Recycling results are useful when computation power is expensive or limited, but storage is cheap and available to a large extent. Therefore the problem of cache replacement was not addressed in their solution. The model was implemented in a Pig-to-Spark compiler Piglet which injects profiling code into the submitted jobs and rewrites them to materialize intermediate results or reuse existing results. They showed that this does not add any significant overhead to execution time, jobs benefit from reusing existing results, and different strategies for choosing which intermediate result to materialize are needed. [8]

F. SparkCruise

Roy, Abhishek, et al. provided SparkCruise, a computation reuse system that automatically selects the most valuable common computations to materialize based on the past query workload. It aims to automatically materialize these common computations as part of query processing, and subsequently reuse them in future queries. No modifications are made to the Spark code. It is presented as plug-and-play with Apache Spark using just the configurations. [13]

G. Solutions Comparison

Different solutions have been provided to utilize materializing intermediate results in Spark, tackling different issues. Choosing the best caching approaching is an NP problem, and most of the solutions provided are heuristic approaches. However, they show a significant gain in performance. The heuristic adaptive approach provided an adaptive solution, taking into consideration the frequency of the computation, the computational cost, and the size of the result. Weighted replacement algorithm provides a selection algorithm based on DAG, frequency, computational cost, and size and provides a replacement algorithm that considers memory capacity. Least Reference Count also provides a lightweight solution for all DAG-based systems. Fairride provides a solution to tackle the problem of cache allocation in a multi-user environment, rather than global efficiency, which makes good use in practical scenarios. Piglet provides a cost-based model and speeds up execution by merging jobs and reusing intermediate results. They do not consider the memory size and cache replacement issue. SparkCruise is another solution that automates the selection of the most valuable computations to materialize and is presented as plug-and-play in Spark.

V. Low-level materialization : An alternative approach

The reuse of intermediate results for in-memory data processing has been researched with the aim to optimize in-memory data processing for data-intensive flows. Dusrun et al. assert that existing solutions to the reuse of intermediates require the materialization of intermediary data of each operator.[12] They add, "inserting such materialization operations into a query plan not only incurs additional execution costs but also often eliminates important cache- and register-locality opportunities."[12] In Revisiting Reuse in Main Memory Database Systems, Dusrun et al. suggest a new reuse model for intermediates which makes use of caching mechanisms to store materialized results during query processing and make them reusable for subsequent operations. The research employs an architecture for storing those intermediates using hash tables and focuses on tackling the reuse problem at a low-level. The paper presents a new main memory database system HashStash that makes use of internal data structures, hash tables, to answer future queries and operations. Traditional reuse models depend heavily on the overlap between operations, and a materialization that results in minimal or no overlaps with subsequent operations incurs extra costs and overhead. Dusrun et al. add, "In the worst case, if the overlap is low, then the extra cost caused by materialization operations might even result in overall performance degradation for analytical workloads." [12] Dusrun et al. suggest that their reuse model eliminates the overhead cost of materialization as it makes use of internal data structures that are already materialized by operations that are pipeline breakers; therefore, even in the case there is no overlap between the materialized data and the subsequent operations, there will be no additional cost.[12] The HashStash reuse model extends two types of reuse models :

- Single query reuse which devises the best reuse plan benefiting from cached internal data structures for a single query and is based on three components: i) caching hash tables along with their statistics ii) reuse optimizer which matches operations with hash tables to be reused according to a cost model iii) garbage collector to remove unused hash tables
- Multi-query reuse which devises the best reuse plan for multiple queries executed concurrently and makes use of previous works on shared

plans extending them to reuse of hash tables and internal data structures and generate an optimized shared plan. [12]

The HashStash reuse model uses a reuse-aware query optimizer and offers to interfaces for the previous reuse models:

- Query-at-a-time interface where the input is a single query and the output is an "optimized reuse aware execution plan." [12] The intuition is to create hash tables that bring about benefits for future operations even if the materialization is expensive if it reduces the cost of subsequent queries. The optimizer is extended several operations reuse types: "exact-, subsuming-, partial-, and overlapping-reuse," adding that "this is different from the existing approaches,..., which only support the exact-reuse." The paper argues that the leveraging exact reuse model only eliminates usage of subplans to construct a query plan because the materialization will be used only in the case there is a hash table containing the exact tuples required by the operation or query. However, combining with other reuse scenarios such as subsuming, partial, and overlapping reuse allows for more optimal reuse of the intermediate results stored as hash tables, however, in some cases this might lead to query approximation. Dusrun et al. add "this might lead to false positives." [12] Dusrun et al. describe the process of reusing the intermediates according to the proposed HashStash reuse model. The optimizers initially generate orders to perform an operation and receives a list of reuse candidate cached hash tables. After generating an optimal reuse plan, the information is forwarded to a hash table manager. Lastly, the optimizer sends the plan to an executor to be executed. After execution, the hash table manager is instructed to release the hash tables and make them available for garbage collection.
- Query-Batch Interface where several queries are executed concurrently and therefore can share one common execution plan. Dusrun et al. refer to it as a "reuse-aware shared plan." [12] Dusrun et al. present a shared query plan which takes advantage of the previously introduced reuse aware plan and applies it to query batches.

Dusrun et al. suggest that their novel reuse aware model generates a 5.3 times performance gain compared to a no-reuse model for average workloads. The main differentiation of this novel reuse-aware model form tradition materialized views, Dusrun et al. add, is that internal data structures, hash tables, are used directly to speed up subsequent queries, in contrast to creating new views in traditional approaches. Therefore, the overhead is removed with HashStash and it is more optimized for main-memory data processing.[12] Since the reuse-aware model proposed by Dusrun et al. [12] leverages mainly the main memory and opts to optimize that performance, it makes the research ideal for usage in the context of Apache Spark which leverages mainly main memory. Moreover, it can optimize the caching mechanism in Spark and automate it to bring about an optimized plan for reusing cached or persisted RDDs not only for one operation but also for subsequent operations by externalizing the cached RDDs. Ideally, it will support all four types of reuse scenarios: exact, subsuming, partial, and overlapping. Currently, the research on reusable intermediate results that are focused on the Spark framework solely is scarce.

VI. Multi-query Optimization

Multiple query optimization has been studied in many research works. In In-memory Caching for Multi-query Optimization of Data-intensive Scalable Computing Workloads [15], Pietro et al. present

a model for data-intensive computing frameworks that combines memory caching and multi-query optimization. The idea is to transform several queries or a batch of them into one query and avoid costly repetitive computations. Pietro et al. define multi-query optimization as aiming to "to find similarities among a set of queries and uses a variety of techniques to avoid redundant work during query execution." [15] The work sheds light on Multi-query optimization in the context of Apache Spark Framework. The research makes use of spark operator to materialize the data in memory. The model proposes starts by finding commonalities between queries and areas in which can be shared by multiple queries. After doing that, multiple sharing plans are generated and materialized in RAM, and the best cost-efficient plans are selected. Finally, a global query plan is rewritten. Pietro et al. suggest that up to an "80% reduction in query runtime," is achieved, they add, "when compared to a setup with no work-sharing." Another work sheds light on Multi-query optimization [12], where Dursun et al. propose a reuse aware shared plan which takes as an input multiple concurrent queries and generated one shared plan. Dursin et al. extend multi-query optimization to leverage the use of internal data structures, hash tables (described in section V). [12] The research does not project the findings into Apache Spark's context.

VII. Conclusions

Efficient cache management plays a crucial role in in-memory data analytics. Existing caching policies are based on historical information. Spark uses the Least Recently Used (LRU) approach to determine which data block to evict when memory capacity is full. Moreover, jobs often have significant computational overlaps, and exploiting these overlaps would result in a huge performance gain. Spark offers caching capabilities in a non-automated fashion. Storage strategies are chosen by the user, and the expected performance is full of uncertainties. Different solutions have been provided to automate this process and utilize materializing intermediate results in Spark, and they have tackled different issues, for instance, DAG dependency, cache replacement, frequency of computation, computational cost, size of the intermediate result, multi-user environment, adaptive over time. Choosing the best caching approach is an NP problem, and most of the solutions provided are heuristic ones. However, they show a significant gain in performance.

References

[1] Apache Spark, *"Apache Spark: Lightning-fast unified analytics engine"*, accessed: 1/7/2020

[2] Karau, Holden, and Rachel Warren. *"High performance Spark: best practices for scaling and optimizing Apache Spark. "* O'Reilly Media, Inc.", 2017.

[3] Zaharia, Matei. *An architecture for fast and general data processing on large clusters.* Association for Computing Machinery and Morgan & Claypool, 2016.

[4] Yang, Zhengyu, et al. *"Intermediate data caching optimization for multi-stage and parallel big data frameworks."* 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018.

[5] Yu, Yinghao, et al. *"LRC: Dependency-aware cache management for data analytics clusters."* IEEE INFOCOM 2017-IEEE Conference on Computer Communications. IEEE, 2017.

[6] Duan, Mingxing, et al. *"Selection and replacement algorithms for memory performance improvement in Spark."* Concurrency and Computation: Practice and Experience 28.8 (2016): 2473-2486.

[7] Pu, Qifan, et al. *"Fairride: Near-optimal, fair cache sharing."* 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). 2016.

[8] Hagedorn, Stefan, and Kai-Uwe Sattler *"Cost-based sharing and recycling of (intermediate) results in dataflow programs."* European Conference on Advances in Databases and Information Systems. Springer, Cham, 2018.

[9] Chen, Min, Shiwen Mao, and Yunhao Liu. *"Big data: A survey."* Mobile networks and applications 19.2 (2014): 171-209.

[10] Rada Chirkova and Jun Yang. 2012. Materialized Views. Now Publishers Inc., Hanover, MA, USA.

[11] Yang, Jian & Karlapalem, Kamalakar & Li, Qing. (1997). *Tackling the challenges of materialized view design in datawarehousing environment.* 32-41. 10.1109/RIDE.1997.583695.

[12] K. Dursun, C. Binnig, U. Çetintemel, and T. Kraska. *Revisiting Reuse in Main Memory Database Systems.* In SIGMOD, pages 1275–1289, 2017.

[13] Roy, Abhishek, et al. *"Sparkcruise: Handsfree computation reuse in spark."* Proceedings of the VLDB Endowment 12.12 (2019): 1850-1853.

[14] Zaharia, Matei, et al. *"Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing."* Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). 2012.

[15] Pietro Michiardi, Damiano Carra, and Sara Migliorini.*"Cache-based multi-query optimization for data-intensive scalable computing frameworks."* arXiv preprint arXiv:1805.08650, 2018.