Review of "Comparative Study Parallel Join Algorithms for MapReduce environment"

Abdur Rafay Saleem
University of Central Florida
ab464825@ucf.edu

***Abstract*—This is a review of the paper "Comparative Study Parallel Join Algorithms for MapReduce environment" by A. Pigul. It comprises the following three sections: an explanation of the research problems and the technical contributions of the paper, an introduction of the proposed techniques, and a critical analysis of the effectiveness of these techniques.**

I. RESEARCH PROBLEM

Data-intensive applications use several types of database architectures and techniques to make processing and analytical decisions fast. They need techniques that involve storing the data in a smart way to process and output the data in the shortest time possible. MapReduce is a popular one along with some of the others mentioned in the paper like parallel DBMS, columnar storage, or a hybrid system based on their combinations.

Performing join operations is in the nature of such data-analytics systems no matter what technique they use. Their workloads are mostly heavy select operations that involve reading millions of tuples of data and aggregating them together for analysis. However, join operations are not supported by MapReduce and the current methods to overcome that limitation aren't that effective. Having un-optimized joins can hurt the speed of response, slow down almost every other operation, impact decision-making and reduce effectiveness of the application. Finding a technique that handles such operations in a quick and optimized way can not only improve result speeds, but also balance workloads across processing nodes to make the system usage-optimal. This paper suggests a variety of different techniques and optimizations that can be done to make join operations performant in a MapReduce environment.

Technically, the author has contributed greatly by going deep in exploration for each of the techniques. The author shares their algorithms, gives a background to their discovery, provides a detailed analysis of their I/O costs, and identifies the advantages and disadvantages to employing them. He conducts an experiment that measures the execution times of the program using the various techniques and makes them subject to a comparative analysis based on several factors.

II. TECHNIQUES

The techniques considered in this paper are called two-way joins and can be divided into Reduce-side join or Map-side join.

A. Reduce-Side Join

This algorithm performs data pre-processing in the Map phase and joins in the Reduce phase. It has three variants: (a) General, (b) Optimized, and (c) Hybrid Hadoop. (a) The general uses tags to read data from multiple sources in the map phase and combines them by their key in the reduce phase using a nested loop join. However, it is limited by the memory requirements and is sensitive to data skew since it should have sufficient memory to hold all the same key records. (b) This uses sorting and grouping of tags to improve on (a) which reduces buffering to only one of the input sets. However, it suffers the same limitations of data transfer overhead and skew. (c) This combines the joins on Map and Reduce side by only processing one source's input in Map phase, reading the second source from distributed storages in Reduce side and joining them directly. This requires pre-

partitioning but solves the problem of sending additional source data to the Reduce side. The main objective of this technique is to reduce the data transmission from Map to Reduce phase by filtering original data using semi joins. Overall, due to the loading of key values in memory, it faces the challenge of memory overflow and data skew.

B. Map-Side Join

This technique doesn't have a Reduce side and performs the join operation in the Map phase. It has two groups: (a) partition join and (b) in-memory join. (a) This join pre-partitions data into the same number of parts using the same partitioner, and joins the matching parts in the Map phase using the default join. However, it is also susceptible to data skew. In cases of partitions being sorted it can use merge join to load partitions on-demand and solve the memory overflow and skew problem. (b) This variant sends the smaller datasets as a whole and larger dataset in partitions to the mappers. The smaller dataset can be read either from a distributed file system or cache. This is more resistant to data skew because of the fixed number of tuples being read but it requires the smaller one to fit into memory completely. There are three methods to solve the memory fit requirement: (1) JDBM-based map join, which does automatic swapping of the hash table between memory and disk and uses HTree instead of hashMap. (2) Multi-phase map join, which partitions the smaller dataset into memory-fittable parts and re-runs the join for each of them. However, this doubles the execution time and causes repeated I/Os to the second dataset not in memory. (3) Reversed map join, which loads the pre-partitioned larger dataset into the hash table and joins the second one from a file line by line. This join is usually optimized with a combination of a distributed cache since it requires repeated reading of the data from the file. All of the variants have their pros and cons and choice depends on the task's requirements.

C. Semi Join

This technique pre-processes the data to remove the tuples that don't participate in the join by using selection or bitwise filters. This reduces the data transmission over the network and size needed for join. However, since this technique requires an additional step, it is only beneficial if there are low-selectivity keys. This technique has three different implementations: (a) Semi-join using Bloom-filter. It is a two stage job, where the first one selects keys from one set and adds them to the bit array (Bloom-filter) in Map phase and combines many filters into one in the Reduce phase. The second stage only has the Map stage which uses this filter on the second dataset. It has the benefit of being compact and accuracy depends on the size of the bitmap. However, it needs a balance between the filter size and false-positive responses and can degrade in performance over large datasets. (b) Semi-join using selection, which extracts unique keys into a hash table and uses it to filter the second set. The first stage uses Map-side for key extraction and Reduce-side for combining into one file. The second stage only involves the Map phase for filtering. However, this implementation faces the challenge of needing a very large in-memory hash table. (c) Adaptive semi-join, which filters the original data on the fly in one stage. In the Map phase it reads the keys from both sources and tags them to identify which source it comes from. Next, the Reduce phase selects the keys with different tags and joins them. This approach can improve performance and solve memory overflow in case of low key-selectivity. However, it suffers from additional network transmission required for transfer of source information in form of tags.

D. Range Partitioners

Earlier techniques, except In-Memory join, suffer from data skew which can be solved by replacing the hash partitioners with the range partitioners. It basically aims to distribute data evenly across nodes before the MapReduce job. This balances the workload on every processing node and reduces the I/O cost being too skewed towards one of them, thus making parallelization faster. It has two types: (a) Simple Range-based Partitioner, (b) Virtual Processor Partitioner. (a) This method randomly selects and splits join keys from the dataset and constructs a range vector before the start of the job. The size equal to the square root of the number of tuples in the dataset is considered optimal

for the vector. The split parts are equal to the number of Reduce phases. The vector is applied to distribute the keys evenly to the nodes. A key is distributed to all nodes to save memory. In cases where it maps to multiple nodes, a node is selected randomly. (b) It follows the same idea, but uses a larger partition vector of size being a multiple of the number of tasks. It loads the nodes more uniformly and scatters keys on more of them than the previous approach. Overall, this is a good optimization that can be combined with any of the techniques above to overcome their limitations of data skew.

E. Distributed Cache

Oftentimes the tasks in nodes can require repeated access to certain files and this requires transmitting data all the way from the global file system. This optimization helps store the regularly used files nearby to all the worker nodes and allows faster access if multiple tasks require the same files. This reduces the need for inter-node communication, network transmission overhead and speeds up the parallel processing. However, having a distributed cache is not always a good strategy because it requires additional effort to maintain and may not cause significant improvements in smaller datasets or algorithms that don't require frequent reading of files. Moreover, having a too large cache requires more memory which means a reduced memory available to the MR program. Another way to achieve the same performance improvement without the cache is to increase replication of files locally to increase the chances of the locally saved copies being read. However, this also is not always ideal since this can use extra storage and if the nature of the data is such that it keeps updating frequently it can easily become complex to sync the changes with the copies. Therefore, important considerations need to be kept in mind to adjust the caching strategy accordingly.

III. CRITICAL ANALYSIS

The paper proposed some very practical and effective techniques to improve the parallel joins' processing times in MapReduce environments. It provides steps of the algorithms of each of the techniques and explains their theoretical idea with great details. The possible limitations of memory overflow and data skew are identified correctly and variations of each technique are provided to address them. However, the techniques only address the situation of two-way joins and in reality a data-intensive application can have much more complex joins like multi-way join or non-equi join.

A multi-way join is a scenario involving joins of more than two datasets. Parallelizing them becomes increasingly complex as the number of datasets involved increases. Managing the load and identifying keys across all datasets becomes more challenging with such joins. The techniques discussed in the paper do not directly address this complexity. These techniques would require further adaptation or combination to effectively handle multi argument joins.

Secondly, the techniques described in the paper are primarily designed for equi-joins (where datasets are joined based on equality conditions). In practice, non-equi joins (where datasets are joined based on equality conditions like less than or greater than) are also prevalent in many applications. Non-equi joins present unique challenges that cannot be easily addressed by hashing or sorting keys, which form the foundation of the techniques discussed in the paper. For example in a Reduce-Side join, the reducer receives all the values for a key and performs the join. This works suitably for equi- joins. However, when the join condition is not equality, it may involve checking a range of keys. This adds complexity to the process as it filters more data, requires more memory, and involves more complex comparisons. With their current state, such scenarios are more likely to lead to memory overflows and slower execution times.

Moreover, the experiments mentioned in the paper were performed on virtual machines which may differ in performance compared to real clusters. To effectively analyze the proposed cost models of the technique it is vital to run the experiments on a real-world cluster containing more than three nodes and datasets of variable sizes.

The paper does a good job of introducing techniques and optimizations that can cater towards improving two-way parallel joins in a map-reduce environment. The best technique depends on the task's specific requirements, the use case, the nature of the data and the computational resources available at hand.