Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

An analysis of two-way equi-join algorithms under MapReduce

Amer F. Al-Badarneh *, Salahaldeen Atef Rababa

Computer Information Systems Department, Jordan University of Science and Technology, P.O. Box 3030, Irbid 22110, Jordan

ARTICLE INFO

Article history:
Received 21 January 2020
Revised 21 April 2020
Accepted 10 May 2020
Available online 15 May 2020

Keywords:
Big data
Join algorithms
MapReduce
Equi-Join
Hadoop

ABSTRACT

Large-scale datasets collected from heterogeneous sources often require a join operation to extract valuable information. MapReduce is an efficient programming model for large-scale data processing. However, it has some limitations in processing heterogeneous datasets. In this study, we review the state-of-the-art strategies for joining two datasets based on an equi-join condition and provide a detail implementation for each strategy. We also present an in-depth analysis of the join strategies and discuss their feasibilities and limitations to assist the reader in selecting the most efficient strategy. Concluding, we outline interesting directions for future join processing systems.

Contents

1. Introduction

The advancement of many technological trends, such as smart devices, Internet of Things (IoT), cloud computing services, web-based services, and social networks, have contributed to the massive amount of data being generated every day at unprecedented rate. Such technologies have led to the emergence of the era of

* Corresponding author.
E-mail addresses: amerb@just.edu.jo (A.F. Al-Badarneh), sarababa13@cit.just.edu.jo (S.A. Rababa).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

Big Data, the era of processing Gigabytes, Terabytes, or even Petabytes of data. What is worth noting is that Big Data analytics has become one of the hottest topics in the field of computer science. In this newly challenging world, data is everywhere, and the driving forces are access to ever-increasing volumes of data and our ever-increasing technological capabilities to mine that data for commercial insights (Marr, 2015). According to the International Data Corporation (IDC) report (Reinsel et al., 2018), the volume of data we created in 2018 reached about 19 Zettabytes (Zetta = 10^{21}) and it is expected in 2025 that the volume of data we create and copy will reach to 163 Zettabytes.

Prior to big data revolution, companies were using traditional database management systems to store and analyze their data. In the context of Big Data, such systems go through poor performance and limited storage capacity. This is due to the characteristics of Big Data and the lack of scalability and flexibility of these systems. Various frameworks have been developed by industry and academia to overcome the limitations of the traditional database management systems for processing large-scale data. Among these are: Google MapReduce (Dean and Ghemawat, 2008), Yahoo PNUTS (Cooper et al., 2008), Microsoft SCOPE (Chaiken et al., 2008), and Apache Spark (Zaharia et al., 2016). These frameworks combine an infrastructure of commodity machines that can scale up to millions of servers and can store databases up to Exabytes (Exa = 10^{18}) of volume. Moreover, such frameworks with their powerful failure handling, can process a large amount of data in a reasonable time in order to extract valuable information.

MapReduce (Dean and Ghemawat, 2008) is an efficient programming model that runs on a shared-nothing cluster for processing large-scale data. It is initially designed for processing and generating large-scale data in terms of two functions *map* and *reduce*. In fact, MapReduce is one of the most popular frameworks in the field of Big Data technologies. This is because of the salient features that it offers, which include simplicity of programming, high scalability, fault-tolerance, and flexibility. However, despite its features, MapReduce has some limitations to perform analytical operations that require processing of heterogeneous datasets. One of the essential such operations is the join, which combines related records from two or more datasets. The main problem with the join operation in MapReduce is the large amount of irrelevant records to the join that are transferred from the map phase to the reduce phase through the network connection.

In this study, we primary focus on the two-way equi-join algorithms in MapReduce context, which are the most common type of join conditions. Multi-way joins can be implemented using a sequence of two-way joins. Therefore, the join strategies analyzed in this paper can be further extended to multi-way joins context. The main aim of this study is to provide a reference for researchers investigating equi-join approaches in MapReduce and inspecting future join optimizations. Additionally, our analytical comparison of the state-of-the-art equi-joins assists the readers in adopting the most efficient join approach according to their application.

The rest of the paper is organized as follows: Section 2 provides an overview of the MapReduce framework and briefly reviews the related works on analyzing join algorithms in MapReduce. Section 3 introduces the detail implementation of equi-join algorithms in MapReduce environment. Section 4 presents an analysis of the algorithms and summarizes their advantages and disadvantages. Finally, we conclude and discuss future work in Section 5.

2. Background & related work

In this section, we provide an overview of the MapReduce framework and briefly reviews the related works on analyzing join algorithms in MapReduce.

2.1. MapReduce overview

MapReduce is an efficient programming model for processing massive datasets. The simplicity of MapReduce framework lies in processing large-scale data as simple units of work, which are *map* and *reduce* functions. The programmer requires only to concentrate on the logic of these simple interfaces and leaves the complexity of task parallelization, load balancing, failure handling, and data distribution to the framework. To create a MapReduce job, the programmer needs only to define the logic within *map* and *reduce* functions. Then, the framework runs multiple instances of these functions in parallel on a cluster of commodity machines. When a MapReduce job is submitted, the following sequence of steps occur (Dean and Ghemawat, 2008), as illustrated in Fig. 1:

a) MapReduce divides the input data into *M* fixed-size blocks called input splits and replicates them three times across the cluster. Then, the *M* splits are passed to the participating nodes in the cluster. Subsequently, the MapReduce framework starts up many copies of the program on these nodes.
b) The master node of the cluster picks idle workers and assigns a map task or a reduce task for each one. Note that the map tasks need to finish their executions before any reduce task can begin, because the output of the map tasks are eventually fed to the reduce tasks.
c) Each worker assigned a map task iterates through each record of the input split. It parses a key/value pair out of each record and applies a user defined *map* function to produce another key/value pair, called the intermediate key/value pair. Then, the output of the map task is written to the worker's memory buffer.
d) The content of the memory buffer is periodically written to the local disk of the worker node and partitioned into *R* regions by the partitioning function. The locations of *R* regions are then passed to the master node, which in turn forwards these locations to the reduce workers.
e) Each reduce worker reads its preserved buffered partition from the local disks of the map workers using remote proce-

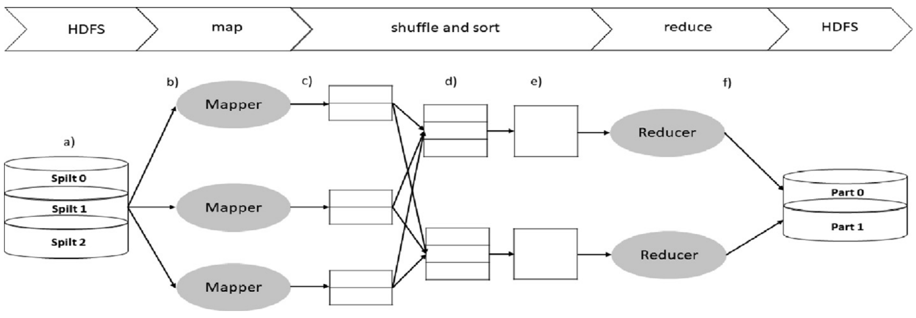


Fig. 1. MapReduce program dataflow.

dure calls. When all intermediate data is copied to the memory of the reduce worker, it begins to sort the data by key in order to group all occurrences of the same key together. In case the intermediate data does not fit in the memory of the reduce worker, an external sort is required.

f) Then each reduce worker iterates through each key of the sorted intermediate data and applies a user defined *reduce* function. In other words, the reduce worker processes a set of intermediate key/value pairs to produce a reduced set of key/value pairs. Finally, the output of the *reduce* function is appended to the final output file of the reduce task.

2.2. Related work

The join operation is one of the most essential, costly, and frequently used operations in relational database query processing (Mishra and Eich, 1992: Graefe, 1993). Basically, it is an important operation that combines records from two or more datasets based on a predefined condition. Many studies have been conducted to discuss the techniques, performance, and optimizations of this operation for large-scale data in MapReduce environment (Afrati and Ullman, 2010: Zhang et al., 2012: Bruno et al., 2014: Gavagsaz et al., 2019). Blanas et al. (2010) provided detail implementations of several equi-join algorithms for log processing in MapReduce. Specifically, repartition join, broadcast join, semi-join and per-split semi-join were introduced in the context of MapReduce. The authors conducted experimental evaluation of the mentioned join algorithms on a 100-node Hadoop cluster and proposed several preprocessing techniques to further improve the join performance. However, this study only considered five join algorithms out of the equi-joins context in MapReduce.

The authors of (Lee et al., 2012a,b) characterized various technical aspects of the MapReduce framework and discussed its pros and cons. Moreover, MapReduce optimizations were reviewed, and several parallel data processing issues were discussed, such as the join operator. They roughly classified join strategies in MapReduce into two groups: Map-side joins and Reduce-side joins and briefly explained these strategies. Doulkeridis and NØrvåg (2014) broadened the range of query processing aspects in the context of MapReduce framework and provided an in-depth analysis of MapReduce limitations. They addressed different types of join conditions, including equi-join, and briefly presented several methods under each category. An analysis of the join algorithms was also introduced based on different phases that improve the join performance.

Phan et al. (2016) provided a systematic approach to compare several filter-based equi-join algorithms and underlined the efficacy of early removal of redundant data in terms of I/O, CPU and communication costs. They theoretically and experimentally examined the impact of extending the equi-join algorithms with filters in MapReduce and characterized the situations where filters are extremely efficient. In our work, we broaden the range of equi-join algorithms in MapReduce and provide an in-depth analysis of the algorithms.

3. Equi-Join algorithms in MapReduce

In this section, we provide the detail implementations of the state-of-the-art equi-join algorithms in MapReduce and address their feasibilities and limitations. We classify the equi-join algorithms into four categories: Standard Equi-Joins in MapReduce, Filter-Based Joins, Skew-Insensitive Joins, and MapReduce Variants.

3.1. Standard equi-joins in MapReduce

Standard Join algorithms in MapReduce can be roughly classified based on which phase the join operation is performed, either in the map phase or in the reduce phase, aka map-side joins and reduce-side joins, respectively. Map-side joins produce the join result in the map phase, since there are no intermediate records sent from mappers to reducers. In contrast, the reduce-side joins send a large amount of intermediate records and produce the join result in the reduce phase. There is a tradeoff between the performance and feasibility of these join algorithms, the better the performance the lesser the feasibility. In this subsection, we present the state-of-the-art map-side and reduce-side joins algorithms.

3.1.1. Map-Side joins

Hadoop framework provides a default implementation for the map-side join (White, 2015), also known as Map-Merge Join (Lee et al., 2012a,b). The map-merge join consists of three MapReduce jobs where the first and the second jobs are run as full MapReduce jobs. They partition and sort the input datasets into an equal number of partitions based on the join key, such that all records having the same join key are residing in the same partition. Although this sounds as a strict requirement, the output of a MapReduce job fits its description. Therefore, the map-merge join could benefit from the outputs of processing the input datasets from previous MapReduce jobs that had the same number of reducers, the same join keys, and the same partitioning function. The map-merge join performs the join operation in the third job, which is a map-only job. Each mapper in the third job deals with one partition from one of the input datasets and loads the counterpart partition of the other dataset into an in-memory hash table. Then, the *map* function performs the merge join and writes the join result to Hadoop Distributed File System (HDFS). Fig. 2 shows an example of a join operation using map-merge join. As depicted in the figure, partitioning and sorting the input datasets make the third job implemented easily, and therefore, improve the join performance. However, if the input datasets are not partitioned and sorted in advance, then the first and the second MapReduce jobs degrade the overall performance of the algorithm.

Map-Side Partition Merge Join (Pigul, 2012) is an improvement of the map-merge join. Its main objective is to avoid memory overflow using on-demand reading of the second dataset. However, it inherits the same drawbacks of its ancestor algorithm, in addition to the overhead of discrete reading. The map-merge join and the map-side partition merge join are suited for large input datasets. On the contrary, if at least one of the input datasets is small enough to fit in the node's cache memory, then the Fragment Replicate Join (Andreas, 2010: Gates, 2011) becomes the optimal solution. For instance, consider two datasets R and L, a reference table and a log table, respectively. In most datasets join applications, the size of the reference table is much smaller than the size of the log table, i.e., $|R| \ll |L|$. The fragment replicate join is run as a map-only job. It replicates the smaller dataset R to all mappers of the larger dataset L and performs the join operation in the map phase. Fig. 3 shows an example of join processing of R and L using fragment replicate join. Firstly, Hadoop framework distributes R to all map tasks of L. Secondly, each map task of L reads the distributed R and builds an in-memory hash table using the join key during the *setup* function, just before the *map* function starts. Finally, in the *map* function, each map task iterates through each record of L and probes the hash table for the record's key, then in case of a match, it writes the join result to the HDFS.

The fragment replicate join algorithm is an efficient map-side join algorithm. However, it can only be performed in case that the size of one of the input datasets is small enough to fit in a node's memory. To address this problem, several approaches have

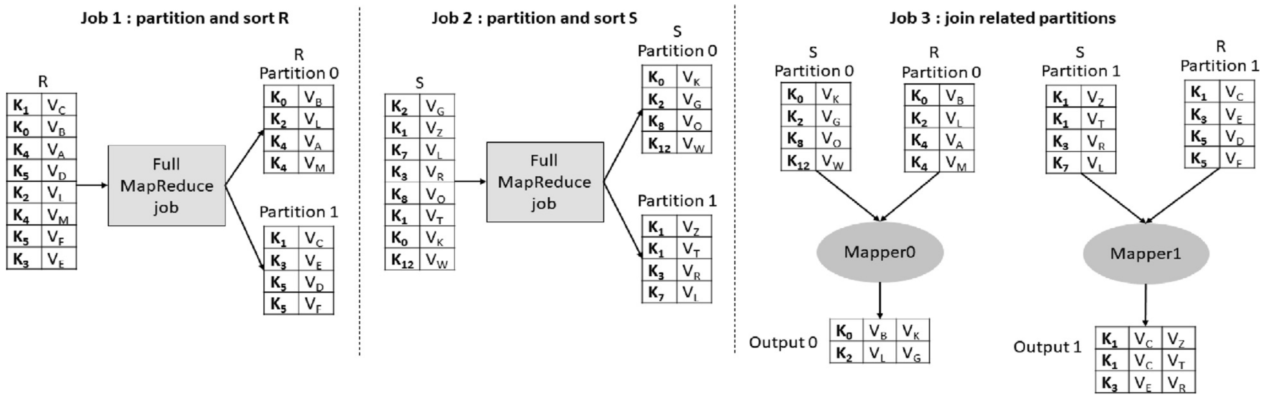

Job 1 : partition and sort R
Job 2 : partition and sort S
Job 3 : join related partitions
Full MapReduce job
Partition 0
Partition 1
Mapper0
Mapper1
Output 0
Output 1

Fig. 2. Map-merge join implementation.

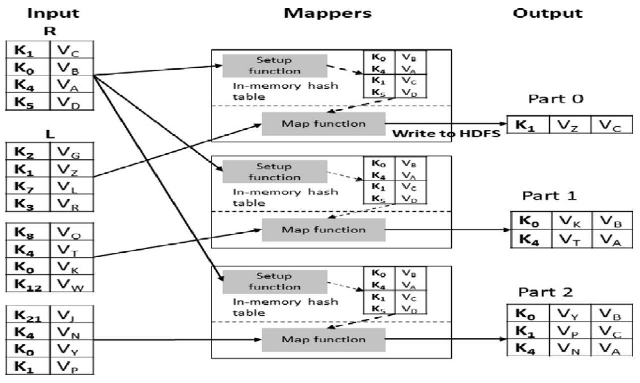


Fig. 3. Fragment replicate join implementation.

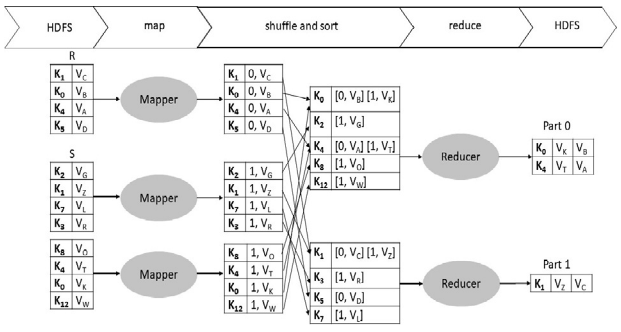


Fig. 4. Standard repartition join implementation.

been introduced, some of which are Reversed Map Join (Luo and Dong, 2010) and Broadcast Join (Blanas et al., 2010). The reversed map join builds a hash table for each input split of L in the *map* function, then, it iterates over each record of R in the *cleanup* function and probes the hash table for the record's key. Finally, it writes the join result to HDFS. The broadcast join combines the implementations of the fragment replicate join and the reversed map join. If the size of R is larger than the size of an input split of L, then the reversed map join is performed. Otherwise, the fragment replicate join is performed. Although the reversed map join and the broadcast join extended the capacity of the input size, the join performance degrades as the input size increases and more complexity are added.

All map-side join algorithms eliminate the network transmission overhead of the intermediate results, unlike reduce-side joins. Therefore, map-side joins optimize the join performance. However, there are some restrictions with regarding the characteristics of the input datasets, and most of the times, additional MapReduce jobs are required to meet these restrictions.

3.1.2. Reduce-Side joins

Reduce-side joins are the most common join algorithms in MapReduce. Standard Repartition Join (Blanas et al., 2010) is one of the commonly used reduce-side join algorithm. Hadoop provides a default implementation for the standard repartition join (White, 2015), which is run in one MapReduce job. Consider the example shown in Fig. 4, suppose that R and S are the input datasets and the number of reduce tasks is equal to two. In the map phase, each mapper iterates through each record of the input split and outputs the join attribute as the key and a tag along with the other attributes as the value, the purpose of the record's tag is to identify its origin. In other words, each mapper outputs a < *key*, *tag value* > pair for each input record. Then, the MapReduce framework partitions, sorts and merges the output of the map phase, such that all records with the same join key are grouped together and eventually fed to a reducer. Next, for each join key, the *reduce* function separates the records based on their tag into two sets and buffers the records of each set. Finally, it performs a cross-product between the buffered records and outputs the join result to HDFS. The standard repartition join is a general algorithm, it has no restrictions with regarding the input datasets, unlike the map-side join algorithms. However, it has some drawbacks as well. The main drawback of the standard repartition join is that all of the input records must be sent to reducers, including irrelevant records to the join operation. These records are transmitted to reducers via the network connection and require extra processing time in the reduce phase. Thus, a performance degradation and a network bottleneck could occur.

Blanas et al. (2010) introduced the Improved Repartition Join as an optimization for the standard repartition join to resolve the buffering problem. One of the drawbacks of the standard repartition join is that all the records for a given key from both datasets must be buffered in the reducer's memory. In case the input datasets are highly skewed or the number of keys in both datasets is relatively small, then it is likely that all the records for a given key may not fit in the reducer's memory. The improved repartition join fixes the buffering problem of the standard repartition join by introducing three critical changes. Firstly, the output of each mapper is changed to a tagged key and a tagged value, i.e., it outputs a < *key tag*, *tag value* > pair for each input record. The tag of each table is generated in a way that the records of the smaller table will be sorted ahead of those records from the larger table. Secondly, to guarantee that the same join key from both tables will be transferred to the same reducer, the improved repartition join overrides the default implementation of the partitioning function, such that the hashcode is computed from just the join key of a record instead of the composite key. The grouping function in the reducer is overridden as well, to ensure that all records are grouped based on the join key. Finally, since the records from the smaller table are guaranteed to be ahead of those records from the larger table, each reducer will only buffer the records of the smaller table and streams the records of the larger table to perform the join for each join key. Fig. 5 shows an example of joining R and S tables using improved repartition join. Although the improved repartition join resolves the buffering problem, it increases the overhead of network transmission compared to the standard version, since each record requires to be tagged twice. In addition, a large amount of redundant records is transmitted through the network, as in the standard version.

Atta (2010) introduced a hybrid approach that is a combination of the map-side and reduce-side joins, which is called the Hybrid Hadoop Join. Like the map-merge join, the input datasets should be partitioned into an equal number of partitions based on the join key. Like the standard repartition join, the join operation is carried out in the reduce phase. However, unlike map-merge join, it requires only one of the input datasets to be partitioned in advance. And unlike standard repartition join, tagging the intermediate records is not required. The hybrid Hadoop join is implemented in two MapReduce jobs. The first job is run as a full MapReduce job, which partitions and sorts the smaller dataset R based on the join key into i partitions. In the second job. Each mapper iterates through each record in an input split of S and passes the key/value pair to the partitioning function, which in turn partitions and sorts S the same way that R was partitioned. Then,

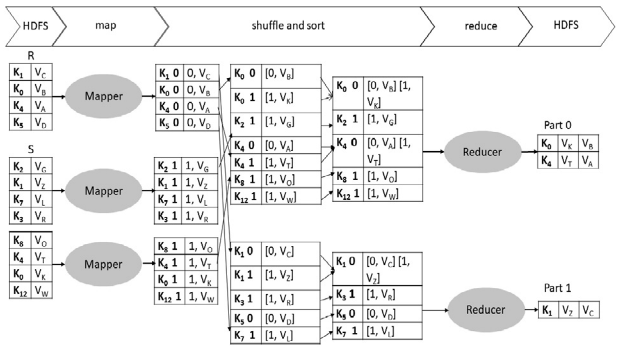


Fig. 5. Improved repartition join implementation.

the output of the partitioning function is fed to a number of reducers that is equal to the number of partitions of R, i.e., i reducers. Next, in each reduce task, the *setup* function loads the counterpart partition R_i into an in-memory hash table, according to the reducer's number. Finally, the *reduce* function probes the hash table for each input key of S_i and performs the join. Fig. 6 shows an example of join processing using hybrid Hadoop join. The hybrid Hadoop join eliminates the overhead of the tagging in the standard repartition join. Moreover, it limits the prerequisites for the map-merge join, since only one of the input datasets is required to be partitioned in advance. However, it inherits the drawbacks of map-side and reduce-side joins as well. In addition, the number of partitions should be determined appropriately.

3.2. Filter-Based joins

Often, in equi-join processing, many records in both datasets do not contribute to the join operation. For instance, consider Facebook's users table R and log table L. R contains over two billion users, and L contains the log data of the active users in a given period of time. In this case, joining R and L can be improved, since the number of active users is likely far less than the total number of users in the reference table R in a given period of time. Therefore, filtering the redundant records significantly reduces the amount of data transferred over the network connection, as well as, the processing time of the join operation. In reduce-side joins, filtering redundant records saves the cost of transferring large portion of R from mappers to reducers and improves the processing time of reducers. In map-side joins, filtering redundant records reduces the amount of data transferred to mappers and minimizes the total amount of memory used by each mapper. However, filtering redundant records requires the use of auxiliary data structures, such as Bloom filter, and an additional MapReduce job is required to build them. Join algorithms that use auxiliary data structures can be divided into three categories: Semi-Joins, Bloom Joins and Adaptive Filter-Based Joins.

3.2.1. Semi joins

Semi-Join (Bernstein and Chiu, 1981) is a classic join algorithm that has been widely used in distributed database processing. It was adapted to MapReduce by Blanas et al. (2010). Semi-join using MapReduce consists of three MapReduce jobs. The main idea of semi-join is to filter out redundant records in one table by using the unique keys list of the other table. Consider an equi-join between two tables, R and L, we need to filter out redundant records in the reference table R. The first MapReduce job runs a map task for each input split of L and one reduce task to aggregate

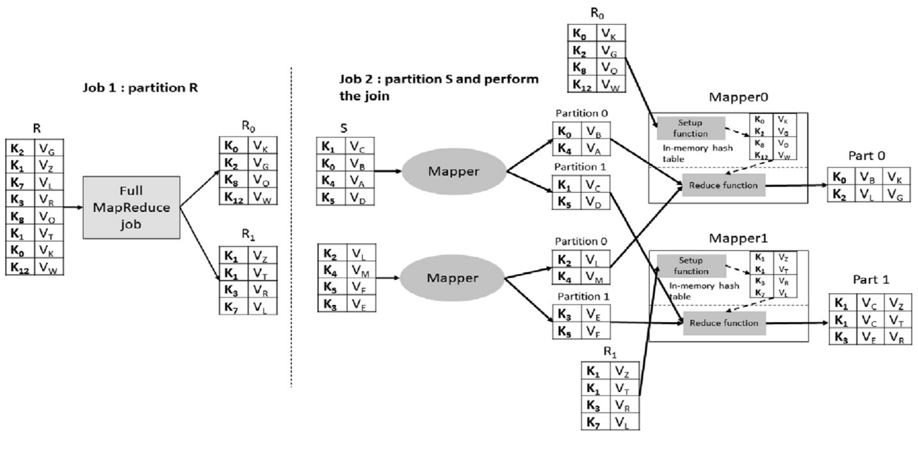


Fig. 6. The hybrid Hadoop join implementation.

the mappers output. In the *map* function, each mapper reads a split of L and builds an in-memory hash table using the join key. Before sending each join key to the reducer, each mapper verifies if the key already exists in the hash table, if not, then it adds the join key to the hash table and sends it over to the reducer. Otherwise, the key is already sent and the mapper proceeds to reading the next record. The main benefit of the hash table is to reduce the total amount of data being sent from mappers to the reducer through sending only the unique keys of the map input. Next, in the *reduce* function, the reducer outputs each unique join key and adds it to the output file, named *L.uk*. The reason that the number of reducers should be one is to consolidate all the unique keys into one file, which is assumed to fit in a node's memory. The second job of the semi-join is run as a map-only job, similar to the broadcast join. The *setup* function of each mapper loads the *L.uk* file into an in-memory hash table. Then, the *map* function iterates through each record of R and extracts the record's key. Finally, it probes the hash table for the record's key and outputs the record in case of a matching. The output of the second job is a list of all records in R that are relevant to the join operation, i.e., each mapper outputs R_i, where R_i is a list of the joinable records in the i^{th} split of R. The third and the final job of the semi-join performs the join operation between the filtered version(s) of R and L using broadcast join. When the size of R is bigger than the size of a split of L, then the standard repartition join is the best choice. Fig. 7 shows an example of the semi-join using MapReduce.

The semi-join filters out redundant records of R that do not join with the records of L and reduces the total amount of transferred data in the join job. However, there are still some redundant records in the filtered version(s) of R that do not join with a specific split L_i of L. For example, in Fig. 7, the number of joined records in R_1, for $R_1 \bowtie L_1$, is equal to one and the rest of records in R_1 are redundant. The Per-Split Semi-Join, a solution to this problem, have been introduced in Blanas et al. (2010). It is implemented in three phases, and each corresponds to a separate MapReduce job. The first and the second jobs are more involved compared to the semi-join, while the third job makes the join processing even cheaper to implement. The idea of the per-split semi-join is to filter out records of the reference table R according to each split L_i of L. The first job is run as a map-only job, each mapper iterates through each record of the input split L_i and outputs a list of unique keys L_i.*uk*, then, it stores the L_i.*uk* file to HDFS. The output of the first job is a collection of files L_i.*uk*'s, each containing a list of the unique keys in the i^{th} split of L. The second job is run as a full MapReduce job, it filters out redundant records in R. In the *map* function, each mapper reads all records of the input split R_i and loads them into an in-memory hash table. Then, the *cleanup* function of the map task loads each L_i.*uk* file from HDFS into an in-memory hash table, and for each L_i.*uk* file, the *cleanup* function probes its hash table for records' keys of R, and outputs the matched records along with a tag, R_{Li}, which will be used by the reducer to collect all the matched records of R_{Li} with a particular split L_i of L. Next, the output of the mappers is eventually fed to one reducer, which in turn outputs different filtered versions of R, i.e., each version is filtered according to a particular split L_i of L. Finally, the third job runs a map-only job to perform map-merge join between R_{Li} and L_i pairs. Fig. 8 shows an example of the per-split semi-join.

The semi-join and the per-split semi-join require three MapReduce jobs to implement a join operation between two tables, which results in increasing the I/O cost and the amount of network transfer. Additionally, the overhead of submitting a MapReduce job to the framework multiple times degrades the overall performance.

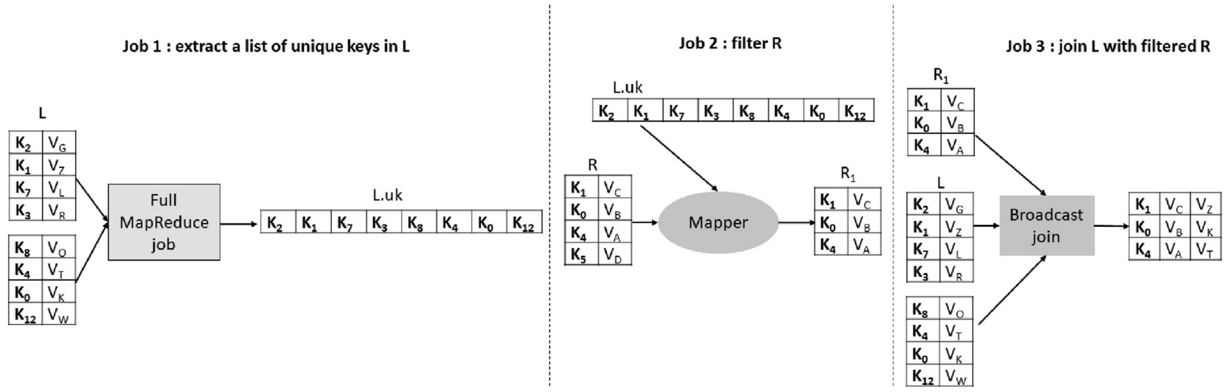

Job 1 : extract a list of unique keys in L
L
Full MapReduce job
L.uk
Job 2 : filter R
L.uk
R
Mapper
R1
Job 3 : join L with filtered R
R1
L
Broadcast join

Fig. 7. Semi-join implementation.

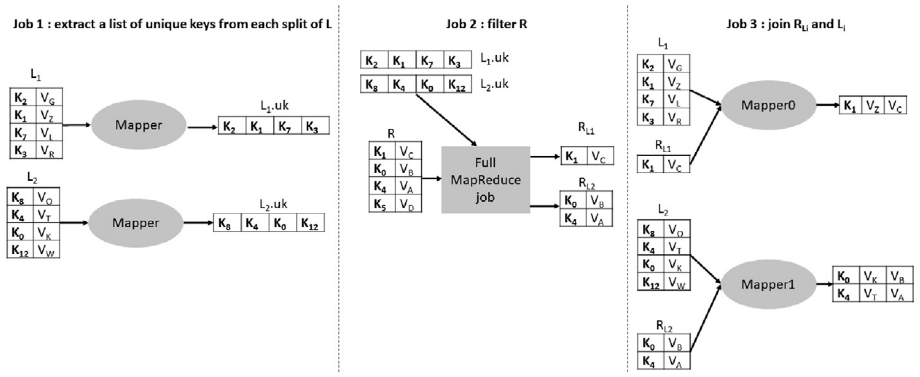


Fig. 8. Per-split semi-join implementation.

Consequently, Matono et al. (2015) introduced the hash-and position-based per-split semi-join as an improvement of the per-split semi-join. It extends the per-split semi-join by introducing three reductions: (1) a reduction in the network traffic through transferring a set of hash values instead of a set of unique keys, (2) a reduction in the total number of MapReduce jobs through implementing the join operation in the second job, (3) a reduction in the amount of disk I/O through using a position-based approach. The hash-and position-based per-split semi-join implements the join in two MapReduce jobs, the first one is run as a map-only job, while the second one is run as a full MapReduce job. Fig. 9 depicts the implementation of the algorithm through an example. In the map task of the first job, the *run* function is overridden in order to obtain position information *p* for each input key/value pair in L_i of *L*. Then, the *map* function adds the hashed value of each key and a set of the key's positions *P* into a Hash table H_L. Finally, the *cleanup* function stores the hash table H_L of each split as a file, named L_i*.hp*, in the HDFS. In the second job, the *map* function builds a hash table H_{Ri} to store the hash values of the records' key/-value pairs of the input spilt R_i of *R*. Then, the *cleanup* function iterates through each file of L_i*.hp*, that is stored in HDFS, and filters redundant records of R_i. Finally, it outputs R_{Li} as the key and *r* + *P* as the value, where R_{Li} is the filtered version of R_i, and *r* + *P* is the concatenation of a joinable record of R_i and a set of records' positions within a split L_i of *L* that will join with *r*. Next, the *reduce* function changes the data structure of the input values from the list of *V*'s (*r* + *P*) to a HashMap *M*, where *P* is hashed as the keys and *r* is hashed as the value. The purpose of this step is to improve the performance of reading *L* through sorting the HashMap *M* by position order. Then, the *reduce* function seeks for the position *p* in order to read the record *l* of L_i, and, finally, it performs the join between *r* and *l*.

3.2.2. Bloom joins

The semi-join algorithms improve the performance of the join operation by virtue of a data structure that contains a list of unique keys of one input dataset or their hashed values. However, the larger the size of the input datasets, the larger the size of the constructed data structure. That means a potential problem will arise in transferring the constructed data structure through the network connection. To cope with this problem, a data structure that is space-efficient regardless of the size of the input datasets and does not yield false negatives is required. One of which is the Bloom filter.

A Bloom filter (Bloom, 1970) is a space-efficient data structure used for testing the membership of an element in a set. False positives are possible in the Bloom filter, but it never produces a false negative. Bloom Join (Mackert and Lohman, 1986) is a distributed join algorithm that uses a Bloom filter, with an accepted false positive probability, to filter out redundant records in the input datasets. Bloom joins using MapReduce have been introduced in (Palla, 2009: Lam, 2010), the approaches are implemented in two independent phases and each corresponds to a separate MapReduce job. Consider an equi-join operation between two tables *R* and *S*. The first job constructs a Bloom filter for the smaller table *R*. In the *map* function, a local Bloom filter BF_i is constructed for each split R_i of *R*. Then, all of the local Bloom filters are eventually fed to one reducer, which in turn merges the them using bitwise OR operation and writes the global filter BF_R to HDFS. Next, the second job filters out redundant records in each input split S_i of *S* using the global Bloom filter BF_R and performs the join in the reduce phase. Fig. 10 shows an example of Bloom join. In order to enhance the filtering process, Palla (2009) and Atta (2010) implemented reduce-join using Bloom filter with different filter sizes and picked the size that resulted in an optimal performance, i.e., the smallest size that

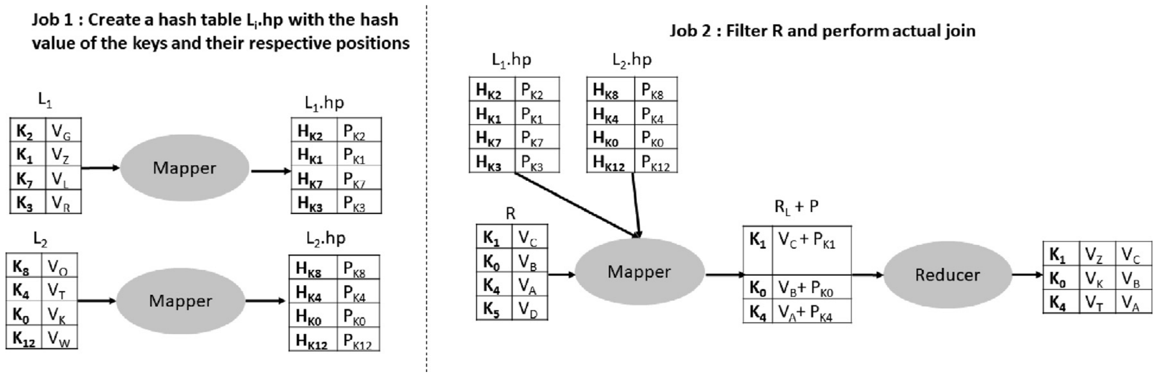
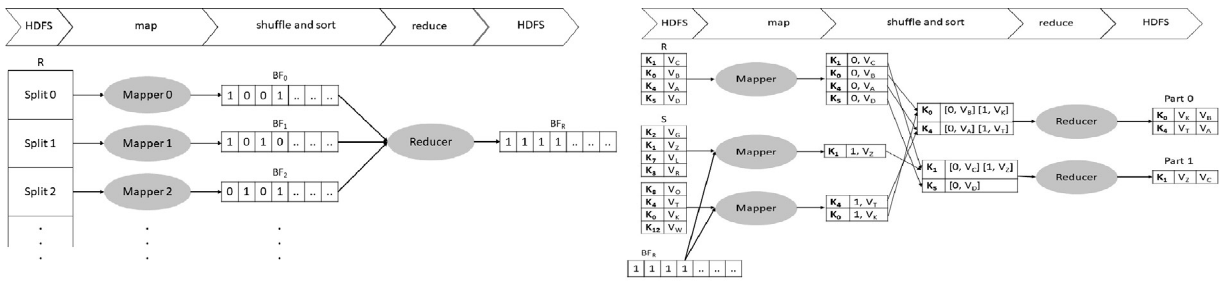

Job 1 : Create a hash table L_i.hp with the hash value of the keys and their respective positions
Job 2 : Filter R and perform actual join
L1
L2
L1.hp
L2.hp
R
R_L + P
Mapper
Reducer

Fig. 9. Hash-and position-based filtering of a reduce-side join.



(a) Construction of a global Bloom (b) Reduce-join with a Bloom filter

Fig. 10. Bloom join implementation.

results in minimal false positives rate. However, these approaches do not discuss how to efficiently build and use the Bloom filter in MapReduce environment.

Zhang et al. (2013) investigated the potential of the Bloom filter in join processing using MapReduce. Three strategies were introduced to build the Bloom filter for large-scale datasets. *Strategy*1 is the same as the one depicted in Fig. 10(a). That is, each mapper creates a local Bloom filter of a fixed size and passes the result to one reducer, which in turn merges all the local filters and outputs one global filter. The size of the constructed filter is determined based on the size of the input dataset R. *Strategy*2 is run as a map-only job, which creates a local Bloom filter for each input split R_i of R and writes it to HDFS. The size of each local filter is determined based on the size of each input split in HDFS, rather than the size of the whole dataset $|R|$. *Strategy*3 is a combination of *strategy*1 and *strategy*2 and runs as a full MapReduce job. Each mapper in *strategy*3 creates a local Bloom filter and passes the intermediate filters to k reducers. Then, each reducer receives M/k filters, where M is the number of mappers, and merges them using bitwise OR operation and writes the merged filter BF_i to HDFS. Fig. 11 shows an example of *strategy*2 and *strategy*3. All three strategies are designed to guarantee that the output filter(s) have the same false positives probability p. The experimental results of (Zhang et al., 2013) showed that *strategy*2 had the best performance in the build phase compared to *strategy*2 and *strategy*3, because it reduces the processing time of the filters and eliminates the cost of transferring the intermediate results to the reduce phase. However, *strategy*2 had the worst performance in the join phase compared to the other strategies, since the look-up cost is relatively high. *Strategy*1 had the best performance in the join phase, but it had the worst performance in the build phase. In contrast to the latter strategies, *strategy*3 exhibited the best performance in both phases if k is chosen properly.

The mentioned Bloom join algorithms, so far, efficiently reduce the total amount of intermediate data by filtering out redundant records in one table. However, there still are many intermediate records, generated from the mappers of the other table, that do not actually participate to the join operation. Consequently, eliminating all redundant intermediate records will significantly boost the join performance. Phan et al. (2013) introduced a method to address this problem. An intersection Bloom filter were designed to remove most of the disjoint records in both tables. Moreover, three approaches were introduced to build the intersection Bloom filter and their feasibilities were discussed in terms of two-way and multi-way joins. The three approaches are quite similar to the strategies suggested in (Zhang et al., 2013) but with an extension to accommodate multiple inputs. *Approach*1 builds a pair of

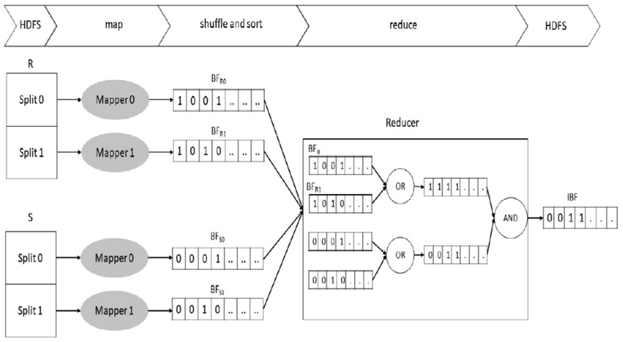


Fig. 12. Example of constructing the intersection Bloom filter using *approach*2.

global Bloom filters, BF_R and BF_S. *Approach*2 builds one global Bloom filter *IBF*; an intersection of BF_R and BF_S using bitwise AND operation. *Approach*3 builds k Bloom filters; each one is an intersection of M/k Bloom filters of R and M/k Bloom filters of S using bitwise AND operation. All approaches guarantee that the created filter(s) can eliminate redundant records in both tables. However, *approach*2 is the most efficient one according to their cost comparison. Fig. 12 shows an example of *approach*2. Join processing using the intersection filter is more efficient than join processing using one filter of either R or S, since it filters out most of the redundant records in both tables and improves the join performance as well. However, extra I/O and processing operations are required to build the intersection filter.

Phan et al. extended their work in (Phan et al., 2016). The impact of early using of the intersection filter for two-way joins, multi-way joins and recursive joins, were examined both analytically using a cost model and practically through experiments. For two-way join algorithms, the intersection Bloom join was implemented and compared to the standard repartition join and Bloom join, and the intersection filter was built using *approach2*. The experimental results showed that the intersection Bloom join outperformed the standard repartition join and Bloom join. Although the intersection filter might increase the false positive probability and requires more preprocessing, its filtering capability and space efficiency outweighs its drawbacks.

3.2.3. Adaptive filter-based joins

The Bloom join algorithms efficiently improve the join performance by filtering out redundant records before transferring them through the network connection. However, an independent

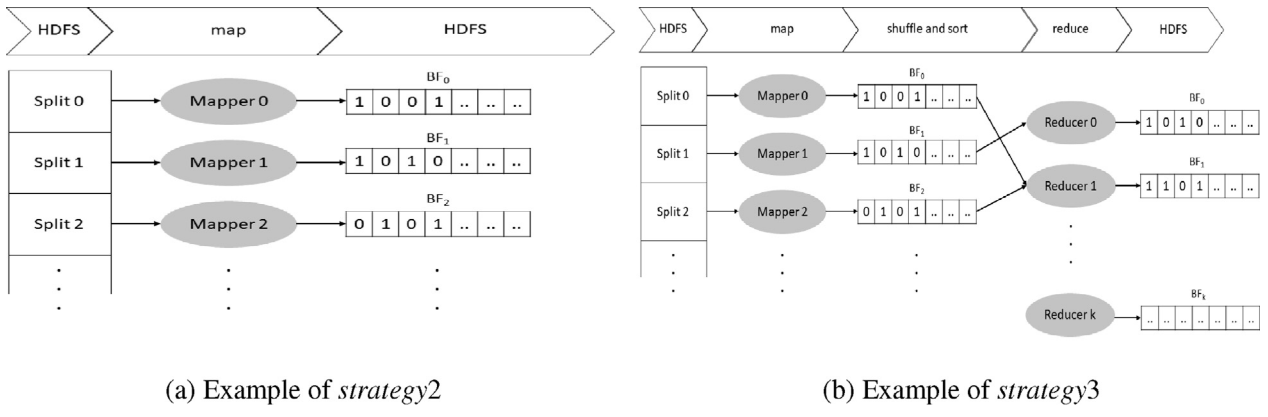


Fig. 11. Strategies for constructing Bloom filters.

MapReduce job is required to create the filter. Therefore, the datasets must be processed multiple times. For instance, the intersection Bloom join processes the input datasets two times, one to build the intersection filter and the other to perform the actual join.

Koutris (2011) theoretically investigated how to apply the Bloom join within one MapReduce job. Two strategies were introduced on how to construct and broadcast the Bloom filter. Strategy *A* computes the Bloom filter at one node and broadcasts it to every participating node in the cluster. Strategy *B* overcomes the bottleneck of central processing in strategy *A* by computing the local Bloom filters in parallel for all nodes and sending them to a target node in order to merge the local filters. However, strategy *B* increases the communication cost by a factor of n (the number of nodes) compared to strategy *A*. Although Koutris (2011) investigated several techniques for join processing using Bloom filter within a single MapReduce job, not enough technical details were provided. Lee et al. (2012a,b,2013) addressed the implementation issue and provided an architecture for join processing using Bloom filter within a single MapReduce job. Two internal modifications to Hadoop framework were made in the proposed architecture. Namely, the order of map tasks is assigned according to the order of the datasets and the construction of Bloom filters is performed in a distributed fashion. Lee et al. (2012) extended their work in Lee (2014); Lee et al. (2014) to introduce Threshold-based Map-Filter-Reduce Join (TMFR-Join), which measures the efficiency of the constructed Bloom filter. That is, if the false positive rate of the global filter exceeds a determined threshold τ, it will be disabled, and reduce-side join is implemented instead. The experimental results showed that the introduced technique had a stable performance close to that of the better of reduce-side join and Bloom join. However, the introduced architecture requires two internal modification of the MapReduce framework.

3.3. Skew-insensitive joins

Basically, the implementation of the join operation in MapReduce passes through four essential phases: mapping, partitioning, shuffling and sorting, and reduce. In the partitioning phase, the MapReduce framework, by default, invokes the *partitioning* function to partition the output of the mappers according to the hash value of the join key. Then in the shuffling and sorting phase, the intermediate results are sent to the reducers, which in turn aggregate the join keys and perform the predefined *reduce* function. As a result, if the input datasets are sufficiently skewed, a potential obstacle can degrade the join performance, since the longest running task dictates the join processing time. Many attempts have been conducted to address this problem in MapReduce context (Atta et al., 2011: Hassan and Bamha, 2015: Myung et al., 2016: Afrati et al., 2018, Gavagsaz et al., 2019). Atta et al. (2011) adapted the range-based partitioning method (DeWitt, 1992) for MapReduce, which primarily divides the join attribute into subranges that approximately have the same number of records in order to balance the workload in subsequent processing. The introduced skew handling algorithm (SAND Join) utilizes two partitioning methods: the simple range partitioner and the virtual processor range partitioner (DeWitt and Ghandeharizadeh, 1990). The simple range partitioner collects samples from each input split and merges them into table T. Then, it retrieves p samples from T to construct a partitioning vector that determines the boundaries for splitting the join keys into p partitions, where p is the number of reducers. The virtual processor range paritioner, on the other hand, creates n^*p partitions, where n is an integer number, and assigns them to p reducers in a round robin fashion. In this way, the processing becomes more evenly distributed among the reducers. However, a serious load imbalance could occur since the size of the join results is not taken into consideration.

The randomized partitioning (Okcan and Riedewald, 2011) is an alternative method for handling data skew in MapReduce. It has been utilized to implement different types of join conditions (Elseidy et al., 2014: Vitorovic et al., 2016). The algorithm exploits the cross-product between two randomly partitioned datasets S and T, where S is divided into m bocks and T is divided into n blocks. The rows in the cross-product space correspond to the m blocks and the columns correspond to the n blocks. Then, the space is divided between k reducers, each of which covers a rectangle in the space. To efficiently balance the load, k is chosen in a way such that the rectangles are of approximately equal sizes. Since S and T are randomly partitioned, each block of m must be joined with each block of n. In other words, the randomized partitioning achieves load balancing with the cost of duplicating the input records. Therefore, it is not suited for larger relations.

Vitorovic et al. (2016) attempted to avoid high input replication by introducing a multistage load balancing algorithm, called EWH. It takes into consideration the properties of both input and output through sampling of the join matrix, which contains important information about the input and output distribution. A category of equi-weight histograms is proposed to divide the join matrix into regions of approximately equal sizes. Gavagsaz et al. (2019) introduced a novel multistage algorithm to avoid input duplication, called fine-grained partitioning for skew data (FGSD). It considers the properties of both input and output through a proposed stream sampling method. Then, FGSD balances the workload among the reducers based on the estimated frequencies of input and output tuples. More recent works in this field can be found in (Potluri et al., 2020: Ibrahim and Bassiouni, 2020).

3.4. MapReduce variants

There have been several attempts to modify the MapReduce framework to cope with the join problem. Map-Reduce-Merge (Yang et al., 2007) is the first one. It introduces a third phase to MapReduce called *merge*, which is invoked after the reduce phase and receives as input the reduced outputs that come from two distinguishable sources. That is, the *merge* phase processes two pairs of < *key*, *value*>, each is coming from one input dataset, and outputs a new merged < *key*, *value* > pair. Consequently, the Map-Reduce-Merge is capable of processing heterogeneous sources in an efficient manner. The Map-Reduce-Merge retains many great features of MapReduce while adding an extra advantage of processing heterogeneous inputs. The authors evaluated the new system in implementing relational database operators and especially the join operator.

Map-Join-Reduce (Jiang et al., 2011) is another variant of MapReduce that adds a *join* phase between map and reduce phases. It introduces a filtering-join-aggregation programming model that uses the *join* phase to implement a join operation in one MapReduce job. The join task is executed within the reduce task. It takes as an input the output of the map tasks and joins qualified records. Then, the output of the *join* phase is pipelined to the reduce phase without the need of checkpointing and shuffling, which in turn improves the join performance. An alternative to a single join job using Map-Join-Reduce is presented with two successive MapReduce jobs. The first job filters out redundant records, joins the qualified records and pushes the intermediate results to reducers for partial aggregation. The second job assembles the partial aggregates to produce the final result.

Although the MapReduce variants frameworks improve the performance of heterogeneous data processing, they cannot reduce the size of intermediate results. In addition, they require modifications of the legacy of MapReduce framework.

4. Analysis of equi-Join algorithms in MapReduce

In this section, we analyze the characteristics of equi-joins in MapReduce based on seven essential metrics that affect the overall performance. Then, we conclude with a summary table of the advantages and disadvantages of each approach to assist the readers in selecting the most appropriate join technique for their application.

The first metric is the *number of jobs*. In general, the lesser the number the lesser the I/O cost. However, in some cases, performing additional jobs can have a positive impact on the overall performance. Second, to further scrutinize the join algorithms, we mention the type of *preprocessing*, which includes: filtering, sorting and sampling. Third, *partitioning* is a crucial factor that determines the work distribution among the worker nodes. This attribute identifies the partitioning method used before performing the join operation, whether in the map phase or reduce phase. The fourth metric, *sensitivity to data skew*, specifies whether the algorithm is sensitive to data imbalance or not. This is a result of the partitioning method used in the join algorithm. Assuming that relation S has m splits and relation R has n splits, the fifth metric, *degree of parallelism*, indicates how many map tasks are being simultaneously executed. This attribute is useful to describe the performance of parallel processing and to measure the utilization of resource. Next, the *tagging* attribute denotes if there is any addition to the intermediate data. Finally, *modifications to Hadoop* specifies whether the algorithm runs on top of the framework or requires some prior modifications.

Table 1 provides a summary of the analysis. In the following we highlight the features of the introduced equi-join algorithms in terms of the mentioned metrics.

Number of jobs and *preprocessing*. To improve the join performance, some join algorithms include a preprocessing step. Map-merge join, map-side partition merge join and hybrid Hadoop join require partitioning and sorting the input datasets beforehand. The aim of this preprocessing step is to facilitate the implementation of the join job. Filter-based joins, on the other hand, construct auxiliary data structures in the preprocessing job(s) to eliminate redundant records in the join job. The type of the constructed data structure differs from one approach to another, semi-join and per-split semi-join use a list of unique keys, hash-and position-based per-split semi-join uses the hash value of the keys, bloom join, intersection bloom join and TMFR-Join use a Bloom filter. Finally, skew-insensitive joins, such as SAND join, require sampling the input relations in the preprocessing job to extract useful partitioning information.

Partitioning and *Sensitivity to Data Skew*. Hash partitioning is the default partitioning technique in MapReduce framework, which leads to serious load imbalance in case of a skewed data. This is the reason why most join algorithms are sensitive to data skew. On the other hand, join algorithms, such as fragment replicate join, that broadcast the smaller relation to all mappers of the larger relation are usually more tolerant in handling data skew, since there is no partitioning involved and the join is performed in memory. Finally, SAND join uses range-based partitioner to balance the load among the reducers.

Degree of Parallelism and *Tagging*. Generally, whether the join is performed in the map phase or reduce phase determines the number of simultaneously running tasks and the need to tag the intermediate data. Joins carried out in the map phase run m map tasks in the join job, and therefore, tagging is not required. On the other hand, joins carried out in the reduce phase run $m + n$ map tasks in parallel and require tagging, except hybrid Hadoop join. TMFR-Join is a special case, which runs max(m,n) map tasks, since it alters Hadoop framework to assign map tasks in a sequential order.

Modifications to Hadoop. Most join algorithms run on top of Hadoop framework. However, there are some MapReduce optimizations for join processing. TMFR-Join modifies Hadoop in two ways: map tasks are assigned in a sequential order and the *jobtracker* and *tasktrackers* functionalities are expanded to be able to

Table 1
Analysis of equi-join processing in MapReduce.

Approach		Number of Jobs	Preprocessing	Partitioning	Sensitive to Data Skew	Degree of Parallelism	Tagging	Modifications to Hadoop
Standard Equi-Joins	Map-Merge Join White (2015)	3	Sorting	Hash-based	Yes	m	No	No
	Map-Side Partition Merge Join Pigul (2012)	3	Sorting	Hash-based	Yes	m	No	No
	Fragment Replicate Join Andreas (2010)	1	No	Broadcast	No	m	No	No
	Reversed Map Join Luo and Dong (2010)	1	No	Broadcast	No	m	No	No
	Broadcast Join Blanas et al. (2010)	1	No	Broadcast	No	m	No	No
	Standard Repartition Join Blanas et al. (2010)	1	No	Hash-based	Yes	$m + n$	Yes	No
	Improved Repartition Join Blanas et al. (2010)	1	No	Hash-based	Yes	$m + n$	Yes	No
	Hybrid Hadoop Join Atta (2010)	2	Sorting	Hash-based	Yes	m	No	No
Filter-Based Joins	Semi-Join Blanas et al. (2010)	3	Filtering	Broadcast	No	m	No	No
	Per-Split Semi-Join Blanas et al. (2010)	3	Filtering	Broadcast	No	m	No	No
	Hash-and Position-Based Per-Split Semi-Join Matono et al. (2015)	2	Filtering	Hash-based	Yes	m	Yes	No
	Bloom Join Zhang et al. (2013)	2	Filtering	Hash-based	Yes	$m + n$	Yes	No
	Intersection Bloom Join Phan et al. (2016)	2	Filtering	Hash-based	Yes	$m + n$	Yes	No
	TMFR-Join Lee et al. (2014)	1	N/A	Hash-based	Yes	max(m, n)	Yes	Task secluding and node functionality
Skew-Insensitive	SAND Join Atta et al. (2011)	2	Sampling	Range-based	No	m	No	No
MapReduce Variants	Map-Reduce-Merge Yang et al. (2007)	1	N/A	N/A	N/A	N/A	N/A	Merge phase
	Map-Join-Reduce Jiang et al. (2011)	2	N/A	N/A	N/A	N/A	N/A	Join phase

Table 2
Equi-join algorithms advantages and disadvantages.

Approach	Advantages	Disadvantages
Map-Merge Join White (2015)	• No intermediate records	• The input datasets must be partitioned and sorted in advance • Sensitive to data skew
Map-Side Partition Merge Join Pigul (2012)	• No intermediate records • Avoid memory overflow	• The input datasets must be partitioned and sorted in advance • Overhead of discrete reading • Sensitive to data skew
Fragment Replicate Join Andreas (2010)	• No intermediate records • Easily implemented • Tolerate data skew	• Only applicable if one of the input datasets is small enough to fit in a node's memory
Reversed Map Join Luo and Dong (2010)	• No intermediate records • Tolerate data skew	• The join performance degrades as the input size increases • Complexity
Broadcast Join Blanas et al. (2010)	• No intermediate records • Tolerate data skew	• The join performance degrades as the input size increases • Complexity
Standard Repartition Join Blanas et al. (2010)	• No size restrictions	• Redundant records • Memory overflow • Sensitive to data skew
Improved Repartition Join Blanas et al. (2010)	• No size restrictions • Fix the buffering problem in Standard Repartition Join	• Redundant records • Network overhead • Sensitive to data skew
Hybrid Hadoop Join Atta (2010)	• Eliminate the overhead of the tagging in Standard Repartition Join • Limit the prerequisites of the Map-Merge Join	• Redundant records • One of the input datasets must be partitioned and sorted in advance • The number of partitions should be determined appropriately • Sensitive to data skew
Semi-Join Blanas et al. (2010)	• Eliminate redundant records in one table • Tolerate data skew	• Inefficient if the number of keys is large • The join performance degrades as the input size increases • Inefficient if the join ratio is high • Not all redundant records are filtered
Per-split semi-join Blanas et al. (2010)	• Eliminate all redundant records in one table • Tolerate data skew	• Inefficient if the number of keys is large • The join performance degrades as the input size increases • Inefficient if the join ratio is high • Complexity
Hash-And Position-BasedPer-Split Semi-Join Matono et al. (2015)	• Eliminate redundant records in one table • Use hash values of the keys instead of a list	• Inefficient if the number of keys is large • Complexity • Inefficient if the join ratio is high • Sensitive to data skew
Bloom Join Zhang et al. (2013)	• Eliminate redundant records in one table • Utilize a space-efficient filter	• Optimizing filter's parameters • Redundant records in the other table • Inefficient if the join ratio is high • Sensitive to data skew
Intersection Bloom Join Phan et al. (2016)	• Eliminate redundant records in both tables • Utilize a space efficient filter	• Optimizing filter's parameters • Extra preprocessing • Inefficient if the join ratio is high • Sensitive to data skew
TMFR-Join Lee et al. (2014)	• Implement the filtering and the join within one MR job • Filter evaluation	• Modifications to Hadoop framework • Sensitive to data skew
SAND Join Atta et al. (2011)	• Simplicity • Insensitive to data skew	• Output distribution is not taken into consideration • Less skewed relation samples are disregarded
The Randomized Partitioning Okcan and Riedewald (2011)	• Work well for any join condition • Address the problems of SAND Join	• High input duplication
FGSD Gavagsaz et al. (2019)	• Avoid input duplication	• Complexity
Map-Reduce-Merge Yang et al. (2007)	• Efficient for heterogenous data processing	• Modifications to Hadoop framework • Redundant Records
Map-Join-Reduce Jiang et al. (2011)	• Efficient for heterogenous data processing	• Modifications to Hadoop framework • Redundant Records

send the local filters and receive the global filter. Map-Reduce-Merge introduces a new phase, called *merge*, which is invoked after the reduce phase is completed. Finally, Map-Join-Reduce alters Hadoop framework by adding a *join* phase between map and reduce phases.

Many join optimizations have been studied in the past 13 years. Determining the most efficient join technique is quite dependent on the application and available resources. For instance, if the join ratio between the input relations is relatively small, then filter-based joins are the most efficient techniques. Table 2 provides a summary of the advantages and disadvantages of each approach.

5. Conclusion

The join operation is one of the most essential, costly, and frequently used operations for data analysis. Join processing using MapReduce is expensive and not easy to implement. In this study, we reviewed the state-of-the-art two-way equi-join algorithms in MapReduce and provided a detail implementation for each approach. Moreover, we introduced an in-depth analysis of the join techniques and summarized their advantages and disadvantages to guide the readers for the most suitable technique for their application. More than a decade of research in join processing using MapReduce can be classified into four categories: adapting existing join approaches in parallel query processing systems to MapReduce, optimizing MapReduce to handle heterogenous data processing, improving low-selectivity joins, and most recently, handling data skew problems. We believe that the next generation of join algorithms in MapReduce should automatically handle data skew and load balancing problems. Furthermore, we expect to see future join optimizer systems that are capable of selecting the most efficient join strategy based on useful information extracted from a reduced cost preprocessing.

In future work, we will extend our study to include multi-way joins and other types of join conditions. We also plan to conduct experimental studies and provide a cost-based comparison to theoretically evaluate the join algorithms. Evaluating join processing in Spark environment is another interesting extension to this study.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Afrati, F.N., Ullman, J.D., 2010. Optimizing Joins in a Map-Reduce Environment. In: In *Proceedings of the 13th International Conference on Extending Database Technology*, pp. 99–110.

Afrati, F.N., Stasinopoulos, N., Ullman, J.D., Vassilakopoulos, A., 2018. SharesSkew: An algorithm to handle skew for joins in MapReduce. Inf. Systems 77, 129–150.

Andreas, C., 2010. *Designing a Parallel Query Engine over Map/Reduce*. Master of Science thesis. School of Informatics. University of Edinburgh, UK.

Atta, F., 2010. Implementation and Analysis of Join Algorithms to Handle Skew for the Hadoop Map/Reduce Framework. Master of Science thesis. School of Informatics. University of Edinburgh, UK.

Atta, F., Viglas, S.D., Niazi, S., 2011. SAND Join—A Skew Handling Join Algorithm for Google's MapReduce Framework. In: In *Proceedings of the 14th International Multitopic Conference (INMIC)*, pp. 170–175.

Bernstein, P.A., Chiu, D.M.W., 1981. Using semi-joins to solve relational queries. J. ACM 28 (1), 25–40.

Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y., 2010. A Comparison of Join Algorithms for Log Processing in MapReduce. Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 975–986.

Bloom, B.H., 1970. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13 (7), 422–426.

Bruno, N., Kwon, Y., Wu, M.C., 2014. Advanced join strategies for large-scale distributed computation. Proc. VLDB Endowment 7 (13), 1484–1495.

Chaiken, R., Jenkins, B., Larson, P.Å., Ramsey, B., Shakib, D., Weaver, S., Zhou, J., 2008. SCOPE: Easy and efficient parallel processing of massive data sets. Proc. VLDB Endowment 1 (2), 1265–1276.

Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R., 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. Proc. VLDB Endowment 1 (2), 1277–1288.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

DeWitt, D.J., 1992. Practical Skew Handling in Parallel Joins. In: In *Proceedings of the 18th International Conference on Very Large Data Bases*, pp. 27–40.

DeWitt, D.J. and Ghandeharizadeh, S., 1990. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machine. In Proceedings of the 16th International Conference on Very Large Data Bases, (pp. 481-492).

Doulkeridis, C., NØrvåg, K., 2014. A survey of large-scale analytical query processing in MapReduce. VLDB J. 23 (3), 355–380.

Elseidy, M., Elguindy, A., Vitorovic, A., Koch, C., 2014. Scalable and adaptive online joins. Proc. VLDB Endowment 7 (6), 441–452.

Gates, A., 2011. Programming Pig. O'Reilly Media Inc., USA.

Gavagsaz, E., Rezaee, A., Javadi, H.H.S., 2019. Load balancing in join algorithms for skewed data in MapReduce systems. J. Supercomput. 75 (1), 228–254.

Graefe, G., 1993. Query evaluation techniques for large databases. ACM Comput. Surv. 25 (2), 73–170.

Hassan, M.A.H., Bamha, M., 2015. Towards scalability and data skew handling in groupby-joins using mapreduce model. Procedia Comput. Sci. 51, 70–79.

Ibrahim, I.A., Bassiouni, M., 2020. Improvement of job completion time in data-intensive cloud, computing applications. J. Cloud Comput. 9, 8.

Jiang, D., Tung, A.K., Chen, G., 2011. MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters. IEEE Trans. Knowl. Data Eng. 23 (9), 1299–1311.

Koutris, P., 2011. Bloom Filters in Distributed Query Execution. University of Washington, USA. [Online] Available at: https://courses.cs.washington.edu/courses/cse544/11wi/projects/koutris.pdf. [Accessed on 7 January 2019].

Lam, C., 2010. Hadoop in Action. Manning Publications Co., USA.

Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B., 2012a. Parallel data processing with MapReduce: A survey. ACM SIGMOD Record 40 (4), 11–20.

Lee, T., 2014. Join Processing with Filtering Techniques on MapReduce Cluster. Doctoral dissertation. Department of Electrical Engineering and Computer Science, College of Engineering. Seoul National University, South Korea.

Lee, T., Bae, H.C., Kim, H.J., 2014. Join processing with threshold-based filtering in MapReduce. J. Supercomput. 69 (2), 793–813.

Lee, T., Kim, K., Kim, H.J., 2012b. Join Processing using Bloom Filter in MapReduce. In: In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pp. 100–105.

Lee, T., Kim, K., Kim, H.J., 2013. Exploiting bloom filters for efficient joins in MapReduce. Inf. Int. Interdisciplinary J. 16 (8), 5869–5885.

Luo, G. and Dong, L., 2010. Adaptive Join Plan Generation in Hadoop. Duke University, Durham NC, USA. [Online] Available at: https://www.semanticscholar.org/paper/Adaptive-Join-Plan-Generation-in-Hadoop-For-Course-Luo-Dong/ca5ec09a367f7c0a10924d88c79d5a7e2e1e8cac. [Accessed on 7 January 2019].

Mackert, L.F., Lohman, G.M., 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In: In *Proceedings of the 12th International Conference on Very Large Data Bases*, pp. 219–229.

Marr, B., 2015. *Big Data: Using SMART Big Data, Analytics and Metrics to Make Better Decisions and Improve Performance*. John Wiley & Sons, UK.

Matono, A., Ogawa, H., Kojima, I., 2015. Improvement of Join Algorithms for Low-Selectivity Joins on MapReduce. In: In *Proceedings of the 26th Australasian Database Conference*, pp. 117–128.

Mishra, P., Eich, M.H., 1992. Join processing in relational databases. ACM Comput. Surv. 24 (1), 63–113.

Myung, J., Shim, J., Yeon, J., Lee, S.G., 2016. Handling data skew in join algorithms using MapReduce. Expert Syst. Appl. 51, 286–299.

Okcan, A., Riedewald, M., 2011. In: Processing Theta-Joins Using MapReduce. Data, pp. 949–960.

Palla, K., 2009. *A Comparative Analysis of Join Algorithms using the Hadoop Map/Reduce Framework*. Master of Science thesis. School. of Informatics, University of Edinburgh.

Phan, T.C., d'Orazio, L., Rigaux, P., 2016. A theoretical and experimental comparison of filter-based equijoins in MapReduce. Transac. Large-Scale Data-and Knowledge-Centered Systems XXV 9620, 33–70.

Phan, T.C., d'Orazio, L., Rigaux, P., 2013. Toward Intersection Filter-Based Optimization for Joins in MapReduce. In: In *Proceedings of the 2nd International Workshop on Cloud Intelligence*, pp. 1–2.

Pigul, A., 2012. Comparative Study Parallel Join Algorithms for MapReduce Environment. Proceedings of the Institute for System Programming of the Russian Academy of Sciences, pp. 285–306.

Potluri, A., Bhattu, S.N., Kumar, N.N., Subramanyam, R.B.V., 2020. Design Strategies for Handling Data Skew in MapReduce Framework. In: In *Proceedings of International Conference on Inventive Computation Technologies*, pp. 240–247.

Reinsel, D., Gantz, J. and Rydning, J., 2018. Data Age 2025: The Digitization of the World From Edge to Core. International Data Corporation (IDC) White Paper, [Online] Available at: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. [Accessed on 7 December 2019].

Vitorovic, A., Elseidy, M., Koch, C., 2016. Load Balancing and Skew Resilience for Parallel Joins. In: In *Proceedings of the 32nd International Conference on Data Engineering*, pp. 313–324.

White, T., 2015. Hadoop: The Definitive Guide. O'Reilly Media Inc., USA.

Yang, H.C., Dasdan, A., Hsiao, R.L., Parker, D.S., 2007. In: MAP-REDUCE-MERGE: Simplified Relational Data Processing on Large Clusters. Data, pp. 1029–1040.

Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., 2016. Apache spark: A unified engine for big data processing. Commun. ACM 59 (11), 56–65.

Zhang, C., Wu, L., Li, J., 2013. Efficient processing distributed joins with bloom filter using MapReduce. Int. J. Grid Distrib. Comput. 6 (3), 43–58.

Zhang, X., Chen, L., Wang, M., 2012. Efficient multi-way theta-join processing using Mapreduce. Proc. VLDB Endowment 5 (11), 1184–1195.