Journal of Computer Science and Cybernetics, V.37, N.2 (2021), 107–122
DOI 10.15625/1813-9663/37/2/15889

RECURSIVE JOIN PROCESSING IN BIG DATA ENVIRONMENT

ANH-CANG PHAN[1,*], THANH-NGOAN TRIEU[2], THUONG-CANG PHAN[2]

[1] *Vinh Long University of Technology and Education, Viet Nam*
[2] *Can Tho University, Viet Nam*

Abstract. In the era of information explosion, Big data is receiving increased attention as having important implications for growth, profitability, and survival of modern organizations. However, it also offers many challenges in the way data is processed and queried over time. A join operation is one of the most common operations appearing in many data queries. Specially, a recursive join is a join type used to query hierarchical data but it is more extremely complex and costly. The evaluation of the recursive join in MapReduce includes some iterations of two tasks of a join task and an incremental computation task. Those tasks are significantly expensive and reduce the performance of queries in large datasets because they generate plenty of intermediate data transmitting over the network. In this study, we thus propose a simple but efficient approach for Big recursive joins based on reducing by half the number of the required iterations in the Spark environment. This improvement leads to significantly reducing the number of the required tasks as well as the amount of the intermediate data generated and transferred over the network. Our experimental results show that an improved recursive join is more efficient and faster than a traditional one on large-scale datasets.

Keywords. Apache spark; Big data; Recursive join; Optimize three-way join.

1. INTRODUCTION

Every person using the Internet contributes to create a large amount of data every day. Managing, storing, querying, and processing Big Data has been and still is an issue of concern in order to take advantage of the value of the existing data sources. Google has designed a programming model for distributed and parallel data processing to solve problems related to big data. The MapReduce model has become one of the most popular big data processing models today [12].

One of the most common operations involved in data analysis and data retrieval is a join query. Join is a basic operation in data processing, which combines multiple relationships or datasets that have some common properties into a new relationship. A recursive join is a complex operation with a huge cost and it is not directly supported by the MapReduce model

*Corresponding author.
E-mail addresses: cangpa@vlute.edu.vn (A.C.Phan); {ttngoan, ptcang}@cit.ctu.edu.vn ({T.N. Trieu, T. C. Phan}).

This paper is an extended and edited version of the report selected from ones presented at the 13th National Conference on Fundamental and Applied Information Technology Research (FAIR'13), Nha Trang University, Khanh Hoa, 08–09/10/2020.

© 2021 Vietnam Academy of Science & Technology

[18, 21, 23]. Since this operation is often used in queries, researches are needed in this area to improve the performance of queries, especially in the context of big data environment.

There are many well-known algorithms for calculating the transitive closure of a relation in traditional databases such as Naive [9], Semi-Naive [9], Magic-set [11], or Smart [16, 18]. Afrati et al. [3] outline how to compute a non-linear transitive closure on a computer cluster for a recursive query. They use two groups of tasks, which are join tasks and Dup-elim tasks (eliminating duplication). The join operation will join the tuples, and the de-duplication operation will remove the tuples that match the one found before passing it to the next join operation. The study discusses a number of alternatives to assist in recovering the system from errors without having to restart the entire job.

Shaw et al. [28] proposed an optimization of the Semi-Naive algorithm for recursive queries in Hadoop MapReduce environment. In Hadoop, the implementation of the Semi-Naive algorithm requires three MapReduce group tasks: one for joining tuples (Join), one for calculating and deleting duplicate tuples (Projection), and one for combining the result with the previous results (Difference). The authors point out that it is not necessary to use a separate MapReduce job for Projection but instead can be incorporated into the Join and Difference jobs. In summary, the solutions perform two repeated MapReduce group tasks, which are join tasks and incremental dataset computation tasks. They also use cache mechanism to minimize the costs involved. However, the cost of reading and writing cache are significant since the whole cache has to be rewritten every time new tuples in the incremental dataset are found.

A recent study [24] has proposed a solution to optimize recursive join in Spark environment. The research team improves recursive join with Semi-Naive algorithm using Intersection Bloom Filter [21, 22, 23] to remove redundant data that does not participate in the join operation to significantly reduce the costs involved. Spark provides an on-memory iterative procession mechanism with Resilient Distributed Datasets. In addition, the study also uses cache mechanism to reuse the datasets that do not change to reduce I/O costs.

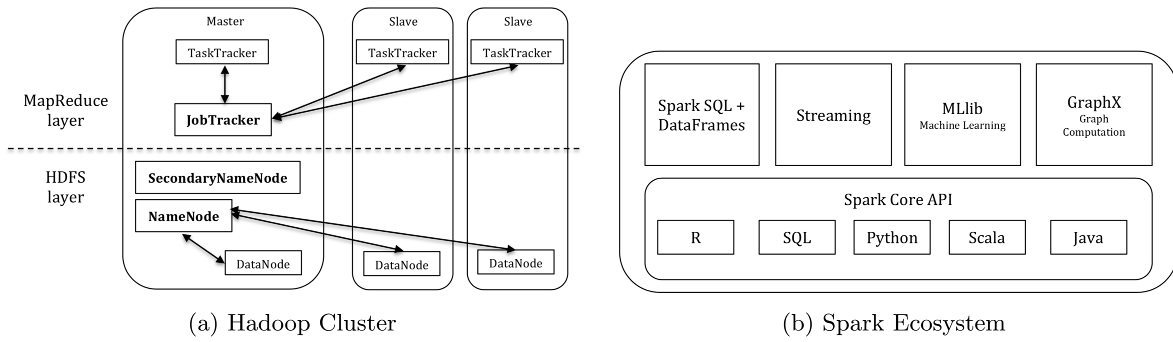
In the studies mentioned above, the authors take advantage of iterative processing, cache, and filter mechanisms to be able to optimize recursive join. These studies focus on redundant data filtering or applying Spark's utilities to support recursive join without directly improving the Semi-Naive algorithm. In the Semi-Naive algorithm for recursive join, at each iteration, we perform joining on dataset K and this dataset is unchanged. Thus, if we perform one more operation of joining dataset K on the same iteration, the number of iterations of the algorithm will be decreased by two. This proposal will be evaluated through the experiments to clarify in which cases and conditions the proposed improvement will be effective.

The structure of this paper is organized as follows. Section 2 presents the background related join operations in Spark environment. Section 3 provides our proposal to effectively process recursive join in Big Data environment. Section 4 gives the evaluation with the experiments. The conclusion of the paper is presented in Section 5.

2. BACKGROUND

2.1. Apache Hadoop and Apache Spark

Apache Hadoop [7] is an open source software utilities collection that supports the use of clusters of multiple computers to solve problems involving huge amounts of data and com-



(a) Hadoop Cluster (b) Spark Ecosystem

Figure 1. Apache Hadoop and Apache Spark

putation. It provides a software framework for distributed storage and large data processing using the MapReduce programming model. The core of Apache Hadoop consists of the storage unit, called the Hadoop Distributed Set System (HDFS), and the processing unit, the MapReduce processing model (Fig. 1a). Hadoop splits input datasets into independent chunks and distributes them across computing nodes in a cluster to parallel process the data.

HDFS is a distributed, scalable, and portable data system written in Java for the Hadoop framework. It provides shell commands and Java APIs as in other data systems. In a large computing cluster, the Datanodes are managed through a dedicated NameNode server to host the data system index and a Secondary NameNode can generate a snapshot of NameNode's memory structure, thus preventing corruption and data loss.

Apache Spark [14, 29] is a unified analytics tool for big data processing, originally developed at University of California Berkeley in 2009. It offers a lot of built-in modules for fast handling of large dataset. Spark Core is the main component of Apache Spark, which is the basic general execution engine for Spark platform on which all other functions are built on top (Fig. 1b). It provides in-memory computing capabilities to delivers speed and provides APIs for popular programming languages such as Java, Scala, and Python.

Spark allows splitting tasks into small parts that can be performed in parallel on computing clusters. Spark takes advantage of in-memory processing power thus it can speed up the processing faster than Apache Hadoop. Spark does not provide its own distributed storage system. Therefore, when using Spark, it is common to install additional software to support distributed data storage. That is the reason users often install Spark on top of the Hadoop platform to use Hadoop's HDFS. Spark allows processing and reusing data stored in memory instead of having to continuously read and write into HDFS as Hadoop.

Spark is a powerful tool for iterative processing on large dataset through Resilient Distributed DataSet (RDD) operations. RDD is a core concept in Spark that is a distributed recoverable collection of elements. It is a highly fault-tolerant parallel data structure, allowing the user to retain intermediate results in memory, controlling storage partitions to optimize data storage location, and processing based on a set of operations. In Spark, all work is performed by creating new RDDs, transforming existing RDDs, or performing calculations on RDDs to get the results.

An RDD provides two types of operations: transformations and actions. Transformations create a new RDD from an existing RDD and actions compute the results based on an RDD and return the results to the program or save them to external memory (e.g., HDFS). Although we can define a new RDD at any time, Spark does the task in a "lazy" way. This

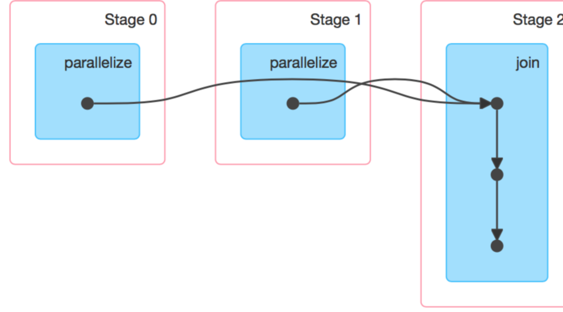


Figure 2. Example of Spark lineage graph in Web UI

means that the transformations will not be performed immediately, but only remember the steps. They are only performed when there is an action related to them is operated. Spark saves the steps as a set of dependencies between different RDDs, which are called the linage graph (Fig. 2). Spark uses this information to compute each requested RDD and to recover data in case a partition of an RDD is lost.

2.2. Bloom filter

Bloom filter [10], proposed by Burton Howard Bloom in 1970, is an space-efficient probability data structure used to test whether an element is a member of a collection. An empty Bloom filter is an array of m bits, all set to zero. There should be k different hash functions defined, each of which maps the element to a position in the m bits array, generating a uniform random distribution. Usually, k is a constant, which is much smaller than m. A choice of k is determined by the intended false positive rate f of the filter (Eq. 1)

$$f = (1-p)^k = (1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-\frac{kn}{m}})^k. \quad (1)$$

The activity of a BF can be described as follows:

- Initially, all bits of BF with 0.
- Add an element x of set S to the BF: The positions of x in the BF are specified by k hash functions and all bits at these positions are set to 1.
- To query an element z, we need to test all k positions of z corresponding with k hash functions.

  - If all bits all k positions are 1 then $z \in S$.
  - In contrast, if any of the bits is 0 then definitely $z \notin S$.

In some cases, a hash function for multiple elements can return the same value so that one element that does not exist in the collection possibly has bits at k positions set to 1. For this reason, a false positive is probable but a false negative is not. False positive elements are elements confirmed by BF as belonging to the collection but they do not. False negative elements are elements confirmed by BF as not belonging to the collection but they do. Bloom filter has many advantages.

- Space efficiency: The size of BF is fixed and independent with the number of elements in the collection but there is a relation between m bits and false positive elements with the probability as equation 1 [13].
- Fast construction: Execute a BF is quite fast since it requires only a single scanning data.
- Element querying efficiency: Checking an element z in collection S requires k hash functions and accessing to k bits.

2.3. Join activity in MapReduce

MapReduce was first introduced by Google in 2004 and is used in many computational operations on large datasets [12]. MapReduce has two main actions, which are Map and Reduce. The Map function is applied on input datasets and produces intermediate results as a list of key-value pairs. The key-value pairs of the same key will be returned to the same Reducer to produce the results for that key with a Reduce function. The MapReduce processing model is presented in Figure 3. Input data in form of (k_1, v_1) is passed to the map function and produces intermediate data, which is a list of key-value pairs $list(k_2, v_2)$. Values with the same key k_2 are grouped together and the reduce function on key k_2 yields a list of values (v_3).

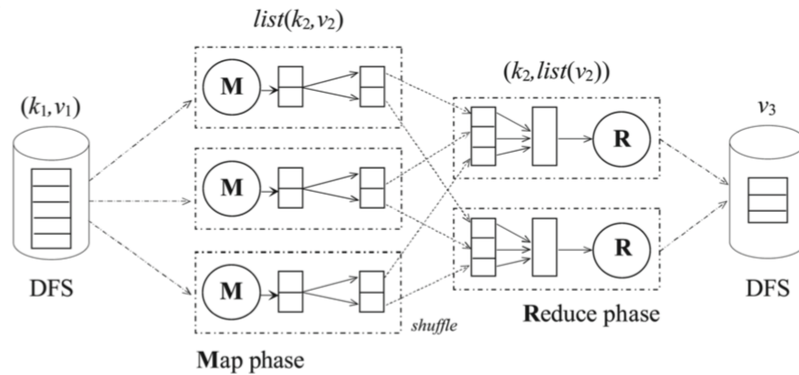


Figure 3. MapReduce processing model [23]

A join operation [21] is used to combines tuples from different datasets based on some conditions. To illustrate the join operation in the MapReduce environment, we consider a join processing for two datasets $R(k_1, v)$ and $S(w, k_2)$.

The join operation in MapReduce will be performed with the tuples of $R\{(A, B), (B, C), (A, D)\}$ and the tuples of $S\{(A, C), (B, F), (D, E)\}$. The map stage is responsible for reading the input blocks of R and S (each block will consist of several tuples). Three Mappers are created as shown in Figure 4 to process three data blocks. The Mappers will convert the tuples into key-value pairs such that k_1 and k_2 are the join keys. The key-value pairs are called intermediate data. Intermediate data with the same key are sent to the same Reducer. A simple reduce function will take each intermediate tuple of R combined with an intermediate tuple of S to produce the result.

- 1: {(B, F), (D, E), (A, C)} → {(B, F), (D, E), (A, C)}

- 2: {(A, D)} → {(D, A)}
- 3: {(A, B), (B, C)} → {(B, A), (C, B)}

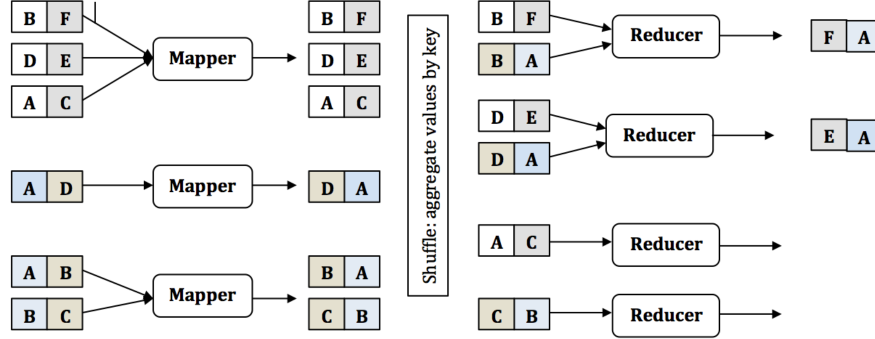


Figure 4. Join operation in MapReduce environment

There are some types of join queries such as two-way join, multi-way join, or recursive join. Of the three types of join above, recursive join is a fairly complex query with a large cost and is applied in many fields that require repeated calculations such as PageRank [17], Graph mining [27], and Friend graph [20, 26] in social networks. Recursive join is also known as "fix-point" join, with repeated operations until the iteration no longer produces new results

$$\begin{cases} \text{(Initialization) } A(x,y) = K(x,y), \\ \text{(Iteration) } A(x,y) = A(x,z) \bowtie K(z,y). \end{cases}$$

A good example of a recursive join is in a friend-of-friend query for a person in social networks. Person X is a friend of person Y when person X knows person Y. Person X is also a friend of person Z if there is a person Y that X is a friend of Y and Y knows person Z.

$$\begin{cases} Friend(X,Y) \leftarrow Know(X,Y), \\ Friend(X,Z) \leftarrow Friend(X,Y) \bowtie Know(Y,Z). \end{cases}$$

2.4. Transitive closure algorithms

The transitive closure of a graph is the reachability matrix to go from vertex u to vertex v of a graph. For a given graph, calculating transitive closure is to find a vertex v accessible from another vertex u, for all pairs of vertices (u, v) [15, 16]. There have been many investigative studies on transitive closure algorithms on large relational databases. The algorithms to calculate transitive closure can be divided into two categories: direct computational algorithms and iterative algorithms. Direct computational algorithms do not consider the problem in terms of recursion and have a starting point as a logical matrix [5].

Commonly found direct computational algorithms are Warshall [31], Warren [30], and Schmitz [25]. Given an initial logical matrix $v * v$ of the a_{ij} elements on a v-node graph, a_{ij} is 1 if there is an arc from node i to node j, otherwise a_{ij} is 0. The transitive closure can be calculated by Warshall's algorithm as described in Algorithm 1. Warren improved Warshall's algorithm by swapping i and k loops. However, this swap can miss some arcs on the graph thus the iteration is done twice (Algorithm 2).

Algorithm 1. Warshall algorithm

```
for k ← 1 to v do
    for i ← 1 to v do
        for j ← 1 to v do
            a_ij = a_ij ∪ (a_ik ∩ a_kj);
        end
    end
end
```

Algorithm 2. Warren algorithm

```
for i ← 1 to v do
    for k ← 1 to i − 1 do
        for j ← 1 to v do
            a_ij = a_ij ∪ (a_ik ∩ a_kj);
        end
    end
end
for i ← 1 to v do
    for k ← i + 1 to v do
        for j ← 1 to v do
            a_ij = a_ij ∪ (a_ik ∩ a_kj);
        end
    end
end
```

The second type is the iterative algorithm developed in the context of recursive queries. These algorithms do not use the special structure of a transitive closure problem. In the iterative algorithms, they can be divided into two sub-categories: linear (Naive [8], Semi-Naive [8]) and non-linear (Smart [2, 16]). Algorithm 3 is Semi-Naive algorithm to calculate bridging closure for graph K. Initialize F with K, F will contain all tuples in the transitive closure of the graph until the end of the algorithm. ΔF contains new tuples created from the previous iteration. On the iteration n, ΔF contains all pairs of nodes such that the shortest path between them has the length of $n+1$. These pairs cannot be detected in the previous loop (thanks to the shortest path between them os of length $n+1$). If any pair created by the join has been found, then there exists a shorter path between the two nodes found on the previous iteration. The F subtraction is set to discard any such tuples in that situation. At the end of iteration n, each tuple in ΔF represents a new tuple discovered in the transitive closure. Smart shares a property of linear algorithms that it is desirable to discover each shortest path only once [2]. On each iteration, it combines paths of exponential length of 2 with paths of smaller length. In this way, it is possible to compute transitive closure in a logarithm number of iterations while acquiring fewer duplicate results (Algorithm 4).

Algorithm 3. Semi-Naive algorithm

```
F = K, ΔF = K;
while ΔF ≠ ∅ do
    ΔF = ΔF ⋈ K − F ;
    F = F ∪ ΔF;
end
```

Algorithm 4. Smart algorithm

```
Q = Edges;
P = ∅;
while Q ≠ ∅ do
    ΔP = Q ⋈ P;
    P = Q ∪ P ∪ ΔP;
    Q = Q ⋈ Q − P;
end
```

3. RECURSIVE JOIN ALGORITHM ON LARGE DATASETS

3.1. Recursive join in MapReduce

Semi-Naive [7, 9, 16, 18] is a popular linear transitive closure algorithm and suitable for deployment in the MapReduce environment. Semi-Naive algorithm for recursive join (briefly called RJ) in MapReduce environment can be represented as follows.

Table 1. Recursive join with Semi-Naive - RJ

$F_0 = K,\ \Delta F_0 = K,\ i = 1;$	(1)
do	(2)
$\quad O_i = \Delta F_{i-1}(c_1, c_2) \bowtie_{c_2 = c_{1'}} K(c_{1'}, c2')$;	(3)
$\quad \Delta F_i = O_i - F_{i-1};$	(4)
$\quad F_i = F_{i-1} \cup \Delta F_i;$	(5)
$\quad$ i++;	(6)
while $\Delta F_i \neq \emptyset$	(7)

Semi-Naive algorithm for recursive join in MapReduce will be performed by two task groups. The join tasks perform the join operation of the tuples, and the tasks for incremental computing dataset are to remove the duplicated tuples found in the previous iteration. At the i_{th} iteration, ΔF_{i-1} joins with K to create the intermediate dataset O_i. In this intermediate dataset O_i, there will be tuples that were found in previous iteration. The incremental computing dataset task will remove these duplicated tuples ($O_i - F_{i-1}$). At the end of each loop, ΔF will contain the new tuples found for the next iteration. Therefore, Semi-Naive is better than Naive since it avoids repeating joining the whole dataset F and K at every iteration. It only allows new tuples to participate in a join operation. The join results can be overlapped to the previous results but those will be removed by the incremental computing dataset task.

In line (3) of the algorithm presented above, we can see that in each iteration, dataset K is constant. It is unchanged over iterations. The most important job in the Semi-Naive algorithm is to join dataset K and dataset ΔF. Thus, if there are n iterations, the dataset K will participate in join operation n times. Consequently, we change the two-way join operation by a three-way join operation to reduce the number of iteration. In other words, we perform the three-way join of dataset K in the same iteration thus the required iterations will be reduced by two.

3.2. Proposal approach for recursive join

The proposed approach using Semi-Naive algorithm for recursive join (briefly called PRJ) in MapReduce environment is represented as follows.

Given dataset $K = [\{A, B\}, \{B, C\}, \{C, D\}, \{D, E\}, \{E, F\}, \{F, G\}]$, we will illustrate our proposal for recursive join.

RJ approach: The first iteration will be a self cartesian product of K. The results of this iteration will be used to compute a cartesian product with K in the next iteration. The

Table 2. Proposal Recursive Join with Semi-Naive - PRJ

$F_0 = K,\ \Delta F_0 = K,\ i = 1;$	(1)
do	(2)
$O_i = \Delta F_{i-1}(c_1, c_2) \bowtie_{c_2 = c_{1'}} K(c_{1'}, c2') \bowtie_{c_{2'} = c_{1'}} K(c_{1'}, c2')$;	(3)
$\Delta F_i = O_i - F_{i-1};$	(4)
$F_i = F_{i-1} \cup \Delta F_i;$	(5)
i++;	(6)
while $\Delta F_i \neq \emptyset$	(7)

iterations will take place until the cartesian product is empty. The results of recursive join with RJ approach in each iteration is provided in Table 3.

Table 3. Illustrations of RJ through iterations

Iteration 1	{A,C}	{B,D}	{C,E}	{D, F}	{E,G}
Iteration 2	{A,D}	{B,E}	{C,F}	{D, G}	
Iteration 3	{A,E}	{B,F}	{C,G}		
Iteration 4	{A,F}	{B,G}			
Iteration 5	{A,G}				

PRJ approach: The first iteration will be join operation of $K \bowtie K \bowtie K$. At each iteration, the join results will be used to join with two original input datasets K in the next iteration. Execution will be continued until the join results become empty. Each iteration in this approach corresponds to two iterations in RJ approach since the join results of one iteration in this approach equal with the join results found in two iterations in RJ. The results of recursive join with PRJ approach in each iteration is provided in Table 4.

Table 4. Illustrations of PRJ through iterations

Iteration 1	{A,C}	{B,D}	{C,E}	{D, F}	{E,G}
	{A,D}	{B,E}	{C,F}	{D, G}	
Iteration 2	{A,E}	{B,F}	{C,G}		
	{A,F}	{B,G}			
Iteration 3	{A,G}				

It is clear that the number of iterations of the proposed algorithm will be reduced by two in comparison with the original one. In other words, one iteration of this algorithm will be equivalent to two iterations of the original Semi-Naive algorithm for recursive join. Provided the number of RJ iterations, we have analyzed the number of iterations of PRJ.

$N_{RJ} = 2 \Rightarrow N_{PRJ} = 1$

$N_{RJ} = 3 \Rightarrow N_{PRJ} = 2$

$N_{RJ} = 4 \Rightarrow N_{PRJ} = 2$
...
$N_{RJ} = 10 \Rightarrow N_{PRJ} = 5$
$N_{RJ} = 11 \Rightarrow N_{PRJ} = 6.$

Given the number of RJ iterations is n, the number of iterations of PRJ is the ceiling of $n/2$. This will be valid for any size of dataset. In fact, recursive join of dataset K is to calculate the transitive closure for graph K. With PRJ approach, new tuples found in the transitive closure in one iteration is equal to that of two iterations in RJ approach. Thus, by around half of n, all tuples in the transitive closure of K are found in PRJ approach. The equation 2 shows the relation between the iterations of the two approaches

$$N_{PRJ} = \lceil N_{RJ}/2 \rceil, \tag{2}$$

where, N_{PRJ} is the number of iterations of PRJ, N_{RJ} is the number of iterations of RJ.

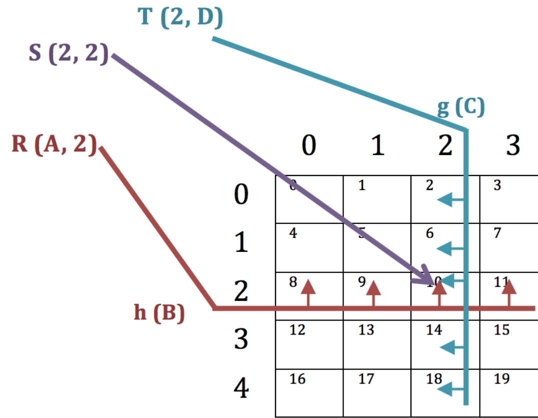


Figure 5. Sending tuples to reducers in one round three-way join

There is the appearance of a three-way join operation in our proposal. Computing a three-way join on three datasets $R(A, B, V) \bowtie S(B, C, W) \bowtie T(C, D, X)$ can be done in two ways. The first way will compute a join of R and S then joins the results with T or vice versa. This method does two rounds of MapReduce jobs. Afrati and Ullman [1, 4] have proposed to do one round MapReduce job for a three-way join operation [19]. In this method, the number of reducers to use is an explicit parameter $k = k_1 * k_2$. There are two hash functions h and g corresponding to the number of k_1 and k_2 slots. Mappers will generate a key-value pair for each tuple in dataset S, generate k_1 and k_2 key-value pairs for each tuple in the datasets T and R. Figure 5 describes the one round three-way join method that passes the tuples to the reducers. The key-value pairs of R and T are sent to the rows and columns in the Reducers matrix, and a key-value pair of S is sent to a reducer to perform join processing.

In the study [23] by T.C. Phan and his colleagues showed that a one round three-way join of three datasets $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ will be better than two round three-way join with the condition $r < (|R|.\alpha)^2$, where r is the number of Reducers, $|R| = |S| = |T|$, and α is the probability of two tuples from different datasets having the same key join. This conclusion is made when considering the size of the intermediate dataset generated from the above two solutions as follows

$$|D| = \begin{cases} 2.|R|.\sqrt{r} & \text{(One round three-way join)}\ , \\ R^2.\delta & \text{(Two round three-way join)}. \end{cases}$$

The larger amount of intermediate data generated, the higher the communication cost, and the higher the join processing cost. The study [23] has shown that the amount of intermediate data generated by one round three-way join is less than the other thus we will use this method for optimizing recursive join operation.

In Afrati and Ullman proposal, key-value pairs of R and T will be sent to rows and columns the reducers matrix. This will greatly increase the number of intermediate tuples. Thus, our study proposes an improvement to the solution of Afrati and Ullman. To avoid the limitation of sending a large number of key-value pairs to rows and columns of the reducers matrix, we will calculate the positions required to send data and save in partitionTable. When generating the key-value pairs for R and T datasets to the rows or columns of the reducers matrix, we must first check which positions of the matrix to transmit data based on the partitionTable (illustrate in Fig. 6). Reducers in unnecessary positions will be ignored. This helps to significantly limit the number of key-value pairs emitting. The partitionTable is calculated by the formula 3

$$pos = H(k_1) * r + H(k_2), \tag{3}$$

where,

- pos: is the reducer position to send data;
- $H(k_1)$: hash position by key k_1;
- $H(k_2)$: hash position by key k_2;
- r: number columns in reducers matrix.

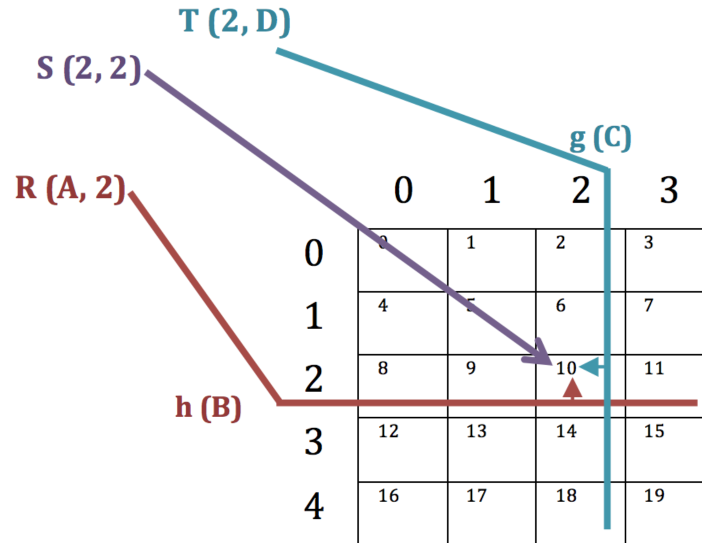


Figure 6. Using partitionTable to send tuples to reducers in one round three-way join

The pseudocode of our proposal for recursive join is shown bellow.

```
Pseudocode
    //Read input data with Spark context
    input = sc.textFile(inPath);
    partitionTable = createPartitionTable(input, keyCol1, keyCol2);
    createBF(input, keyCol1, keyCol2); //create filters
    //Create the key-value pairs
    F = createPairRDD(input, keyCol1, keyCol2);
    deltaF = F;
    K1pairs = createPairRDD(input, partitionTable, BFK, keyCol1); //cache
    K2pairs = createPairRDD(input, keyCol1, keyCol2); // cache
    do{
        deltaFpairs = createPairRDD(deltaF, partitionTable, BFdeltaF, keyCol2);
        //Join deltaF and K1, K2
        result = join(deltaFpairs, K1pairs, K2pairs);
        //Remove redundant data
        deltaF = result.subtract(F).distinct();
        //Save to HDFS
        deltaF.saveAsTextFile();
        //Update F
        F = F.union(deltaF);
        iteration++;
    }while(!deltaF.isEmpty() && iteration < MaxIteration)
```

In addition, to reduce the amount of unnecessary data involved in the join operation, we built two bloom filters for filtering redundant data. It is necessary to create an array of m bits and k hash functions ($m = 120,000$ and $k = 8$). In the two bloom filters $BF_{\Delta F}$ and BF_K, one is used for dataset K and one is used for dataset ΔF respectively.

4. EXPERIMENTS

4.1. Computer cluster

We install a Spark cluster on a computer system including 1 master and 10 slaves provided by the Mobile Network and Big Data Laboratory of College of Information and Communication Technology, Can Tho University. The computer configurations are 5 CPUs, 8GB RAM, and 100GB hard disks. The operating system used is Ubuntu 18.04 LTS and the applications are Hadoop 3.0.3, Spark 2.4.3, andJava 1.8.

The experiments were conducted with datasets from PUMA Benchmarks [6] with the capacities of 1GB, 5GB, 10GB, 15GB, 20GB, and 25GB corresponding to 2.6, 13.5, 26.8, 40.2, 53.6, and 67.1 million data records respectively. The datasets are stored in plain text file format with 39 fields per line separated by commas and 19 data characters per column.

Experiments will conduct to evaluate the two approaches RJ and PRJ for recursive join. On each experimental dataset, we will record the amount of intermediate data transmitted over the network, the execution time, and the number of iterations for analysis and comparison.

Table 5. Number of iterations

Approaches \ Size	**1GB**	**5GB**	**10GB**	**15GB**	**20GB**	**25GB**
RJ	4	4	6	9	11	11
PRJ	2	2	3	5	6	6

4.2. Results

The number of iterations corresponding to the two approaches was recorded. It can be seen that the number of iterations of PRJ is around two times lower than that of RJ (Table 5).

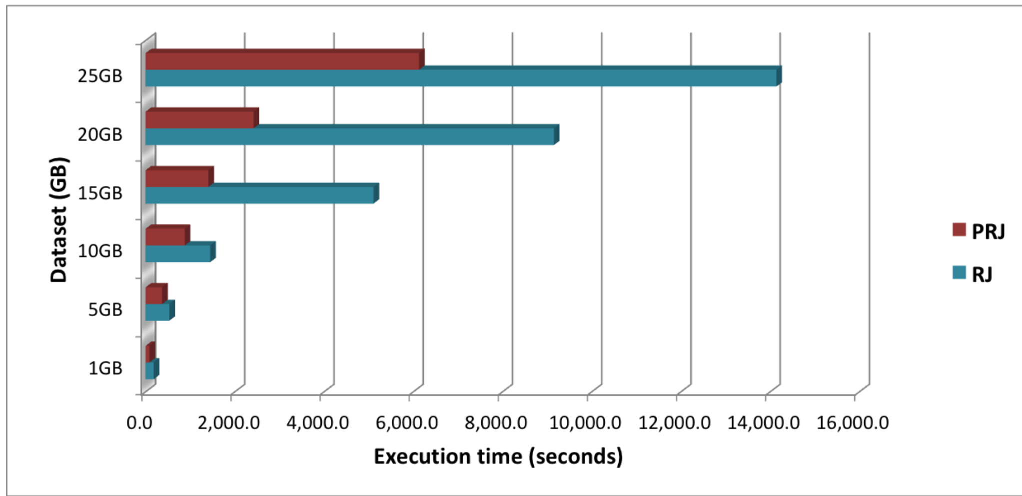


Figure 7. Execution time of RJ and PRJ (seconds)

The execution time specified by each approach is recorded in seconds. There is a big difference in processing time between RJ and PRJ. The results in Figure 7 show clearly the improvement in processing speed with PRJ approach compared to RJ. The performance of one round three-way join in Semi-Naive algorithm for recursive join has greatly reduced the execution time. However, with a small dataset, the processing speed is quite the same of the two approaches. This can be understandable since PRJ has the preprocessing to build a partitionTable to transmit data efficiently and to build filters to remove redundant data that does not participate in join operation. When the amount of input data is small, it will be time-consuming for preprocessing, thus it is not effective.

Figure 8 shows the amount of intermediate data to be transmitted over the network for a recursive join of the two approaches. The decrease in the number of iterations reduces the number of MapReduce jobs, which in turn reduces the amount of intermediate data. Besides, filters help a lot for reducing redundant data that do not participating in join operation to optimize the recursive join.

In addition, we also test the proposal approach on a 10GB dataset with different number of working nodes. The execution time is presented in Figure 9.

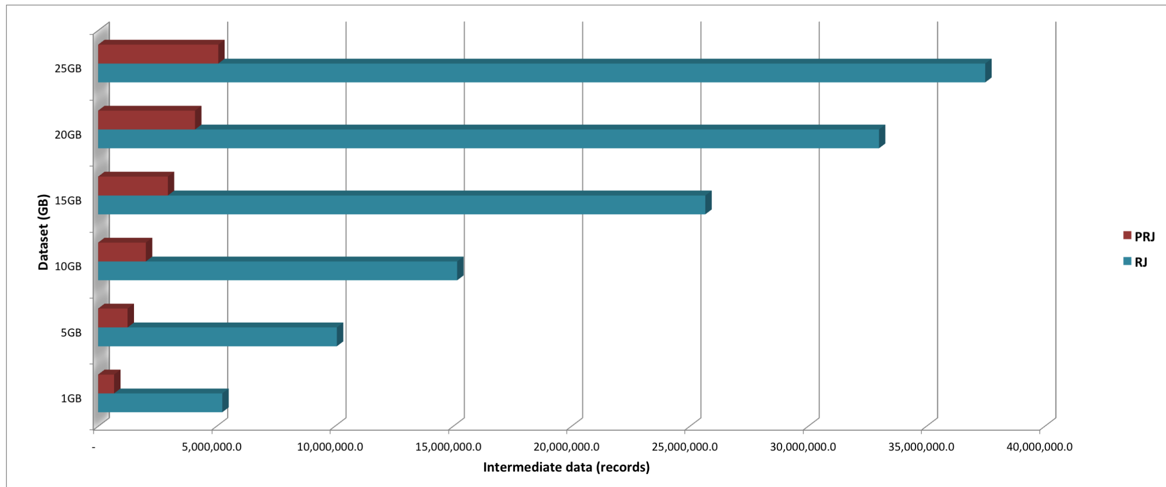


Figure 8. Intermediate data of RJ and PRJ (records)

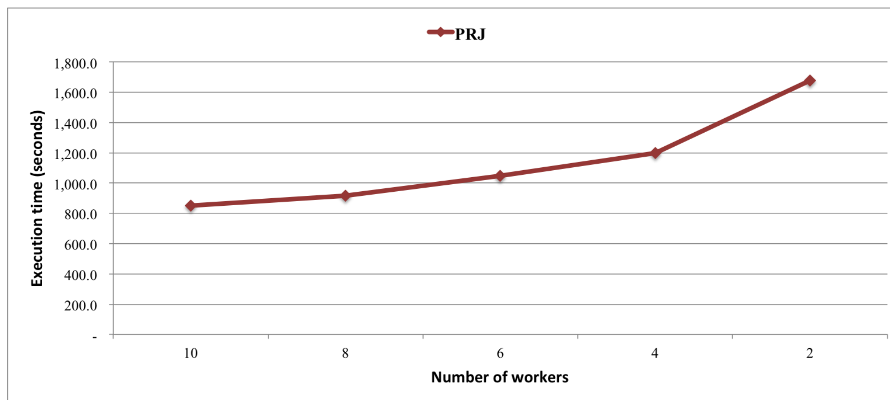


Figure 9. PRJ execution time for 10GB data

5. CONCLUSION

The study has fully analyzed the recursive join in the big data processing environment with MapReduce and proposed important improvements to significantly reduce the costs involved. In our proposal, we utilize dataset K that is constant through iterations to propose the use of three-way join for reducing the number of iterations and the number of MapReduce jobs. We set up the one round three-way join using the idea from the study of Afrati and Ullman. To avoid extreme generating key-value pairs to send to the whole rows and columns of the reducers matrix, we construct a partitionTable that can partially reduce the number of unnecessary data. Besides, the use of filters is also to remove redundant data that does not participate in the join operation.

In brief, this study has come up with a new approach to effectively optimize recursive join in MapReduce environment. The experiments show the effectiveness of improvement for Semi-Naive in recursive join in MapReduce. This is a highly practical contribution since the Semi-Naive algorithm is a very common algorithm used in recursive joins in big data environments.

REFERENCES

[1] F. N. Afrati and J. D. Ullman, "Optimizing multiway joins in a map-reduce environment," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1282–1298, Sep. 2011.

[2] F. N. Afrati and J. D. Ullman, "Transitive closure and recursive datalog implemented on clusters," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT 12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 132–143.

[3] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in *Proceedings of the 14th International Conference on Extending Database Technology*, 2011, pp. 1–8.

[4] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*, ser. EDBT 10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 99–110.

[5] R. Agrawal and H. V. Jagadish, "Direct algorithms for computing the transitive closure of database relations," in *Proceedings of the 13th International Conference on Very Large Data Bases*, ser. VLDB 87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, pp. 255–266.

[6] F. Ahmad. (Oct.) Puma benchmarks and dataset downloads. [Online]. Available: https://engineering.purdue.edu/~puma/datasets.htm

[7] Apache, "Apache hadoop," 2002, last Accessed: July 20, 2019. [Online]. Available: https://hadoop.apache.org

[8] F. Bancilhon, *Naive Evaluation of Recursively Defined Relations*. Berlin, Heidelberg: Springer-Verlag, 1986, pp. 165–178.

[9] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *SIGMOD Rec.*, vol. 15, no. 2, pp. 16–52, Jun. 1986.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] Y. Chen, "On the bottom - up evaluation of recursive queries," *International Journal of Intelligent Systems*, vol. 11, pp. 807–832, 1996.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *In Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006, pp. 1–12.

[14] D. Inc, "Apache spark is a lightning-fast," 2009, last Accessed: July 20, 2019. [Online]. Available: https://databricks.com/spark/about

[15] Y. Ioannidis and R. Ramakrishnan, "Efficient transitive closure algorithms." 01 1988, pp. 382–394.

[16] Y. E. Ioannidis, "On the computation of the transitive closure of relational operators," in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB 86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 403–411.

[17] T. A. Jilani, U. Fatima, M. M. Baig, and S. Mahmood, "A survey and comparative study of different pagerank algorithms," *International Journal of Computer Applications*, vol. 120, no. 24, pp. 24–30, 2015.

[18] R. Kabler, Y. E. Ioannidis, and M. J. Carey, "Performance evaluation of algorithms for transitive closure," *Information Systems*, vol. 17, no. 5, pp. 415–441, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030643799290035L

[19] B. Kimmett, A. Thomo, and S. Venkatesh, "Three-way joins on mapreduce: An experimental study," in *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*, July 2014, pp. 227–232.

[20] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Social-Network Graphs*, 2nd ed. Cambridge University Press, 2014, pp. 325–383.

[21] T.-C. Phan, "Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce," Theses, Université Blaise Pascal - Clermont-Ferrand II, Jul. 2014. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01066612

[22] T.-C. Phan, L. d'Orazio, and P. Rigaux, "Toward intersection filter-based optimization for joins in mapreduce," in *Proceedings of the 2nd International Workshop on Cloud Intelligence*, ser. Cloud-I 13. New York, NY, USA: Association for Computing Machinery, 2013.

[23] T.-C. Phan, L. D'Orazio, and P. Rigaux, "A theoretical and experimental comparison of filter-based equijoins in mapreduce," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXV - Volume 9620.* Berlin, Heidelberg: Springer-Verlag, 2015, pp. 33–70.

[24] T.-C. Phan, A.-C. Phan, T.-T.-Q. Tran, and N.-T. Trieu, "Efficient processing of recursive joins on large-scale datasets in spark," in *Advanced Computational Methods for Knowledge Engineering*, H. A. Le Thi, H. M. Le, T. Pham Dinh, and N. T. Nguyen, Eds. Cham: Springer International Publishing, 2020, pp. 391–402.

[25] L. Schmitz, "An improved transitive closure algorithm," *Computing*, vol. 30, no. 4, pp. 359–371, Dec 1983.

[26] J. Seo, S. Guo, and M. S. Lam, "Socialite: Datalog extensions for efficient social network analysis," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 278–289.

[27] S. Seufert, A. Anand, S. Bedathur, and G. Weikum, "High-performance reachability query processing under index size restrictions," 2012.

[28] M. Shaw, P. Koutris, B. Howe, and D. Suciu, "Optimizing large-scale semi-naïve datalog evaluation in hadoop," in *Datalog in Academia and Industry*, P. Barceló and R. Pichler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 165–176.

[29] A. Spark, "Lightning-fast unified analytics engine," 2009, last Accessed: July 20, 2019. [Online]. Available: https://spark.apache.org

[30] H. S. Warren, "A modification of warshall's algorithm for the transitive closure of binary relations," *Communications of the ACM*, vol. 18, no. 4, pp. 218–220, Apr. 1975.

[31] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

Received on February 23, 2021
Accepted on May 12, 2021