SN Computer Science (2021) 2:352
https://doi.org/10.1007/s42979-021-00738-x

SURVEY ARTICLE

A Theoretical and Experimental Comparison of Large-Scale Join Algorithms in Spark

Anh-Cang Phan[1] · Thuong-Cang Phan[2] · Thanh-Ngoan Trieu[2] · Thi-To-Quyen Tran[3]

Received: 11 March 2021 / Accepted: 8 June 2021 / Published online: 23 June 2021

Abstract

Currently, the estimated amount of data created daily have reached the threshold of petabytes or even zettabytes globally. It is no wonder that traditional data processing technologies cannot process and manage extremely large volumes of such data. However, these massive and various data can be used to deal with business problems that we would not have been able to tackle before. To discover their value, it is necessary to effectively perform query operations in a parallel and distributed manner. One of the standard and common query operations is an expensive join operation. This research systematically presents a theoretical and experimental comparison of the prominent join algorithms in the Spark environment. At first, this study shows the details of important strategies of two-way joins and recursive joins. Then, it exposes the advantages and disadvantages of each approach. Especially, the work provides mathematical cost models to make a more convince comparison of the joins before verifying by experiments. The results show that the comparison using the cost models is consistent with that using the experiments. Generally, the two-way and recursive joins using filters are the best choices while performing in the Spark environment.

Keywords Join operation · Big data analytic · Spark · MapReduce · Bloom filter

Introduction

Big data processing is a very important requirement for many applications in a wide range of fields such as commerce, scientific research, health care, security, customer behavior, natural disasters, etc. Processing large datasets requires compatible systems to work in a parallel and distributed manner. The idea is to divide a computation into small tasks that can be executed on a cluster of computers. The main model to handle the problem is MapReduce [12], which works with two main operations: Map and Reduce. The Map functions transform and organize data while the Reduce functions aggregate data. This model proposed by Google in 2004 and has become the current standard and the most popular model for handling large datasets on parallel and distributed systems.

A typical and frequently used operation in data analysis is a join operation. The join operation combines data from two or more different datasets with a key join and results a new dataset. This is a complex and costly operation when implemented in the MapReduce environment. The join operation in MapReduce goes through two phases, map and reduce, with expensive shuffling costs (moving data from mapper to reducer). Therefore, executing join queries on large datasets in MapReduce presents great challenges for researchers. One of the solutions for reducing costs of join computation is removing redundant elements that are not involved in the join operation.

There are several types of join operations such as two-way join, multi-way join [27], chain join [17], and recursive join

This article is part of the topical collection "Future Data and Security Engineering 2020" guest edited by Tran Khanh Dang.

✉ Anh-Cang Phan
cangpa@vlute.edu.vn

Thuong-Cang Phan
ptcang@cit.ctu.edu.vn

Thanh-Ngoan Trieu
ttngoan@cit.ctu.edu.vn

Thi-To-Quyen Tran
thi-to-quyen.tran@irisa.fr

1 Vinh Long University of Technology Education, Vinh Long City, Vietnam

2 Can Tho University, Can Tho City, Vietnam

3 IRISA, University of Rennes 1, Lannion City, France

[24]. Currently, many researches have been done on two-way join operations in MapReduce with Apache Hadoop[1] [8, 15, 23]. The join algorithms investigated mainly include Map-side join [8, 15], Broadcast join [8] that is considered to be a Hash join [11, 21], and Reduce-side join [15, 31] that is similar to a Repartition join [8]. In addition, a number of algorithms is developed to improve the effectiveness of join operations. Most of these algorithms focus on solving the problem of removing redundant data before performing join operations such as Parallelize set-similarity joins [21], Bloom join [10, 16, 18, 20, 23], and Intersection filter-based Bloom join [23, 25]. Recently, with the popularity of Apache Spark[2], there have been a few studies on join activities but mainly done based on the support of database systems such as NoSQL [28], Hive [19], or queries made directly from Spark SQL [5]. The join algorithms in MapReduce has been analyze in several studies [3, 4, 24]. The study [3] has shown a survey on join algorithms in Spark.

A recursive join is also called a fix-point join, which repeats the join operations until a fix-point is reached. It computes the transitive closure of a relation with repeated join operations. There are two types of algorithms dealing with the problem of computing transitive closure: direct calculation algorithms (such as Warshall [30] and Warren [29]) and iterative algorithms (such as Naive [6], Semi-Naive [7], and Smart [1]). However, not all of these these algorithms are appropriate to implement in MapReduce. The results in study [13] show that Semi-Naive can be the best choice to process recursive join in MapReduce. Shaw et al. [26] proposed a solution for recursive join in Hadoop MapReduce environment using Semi-Naive algorithm. The research repeats two type of jobs, which are join job and incremental computing dataset job.

Since Hadoop framework has slow processing speed than Apache Spark [32], our study conducts experiments on Spark for the commonly used join algorithms. As a scientific basis, this study provides an overview of common two-way join algorithms and recursive join with Semi-Naive in MapReduce. We evaluate the algorithms based on general cost models and experiments in Spark. This research extends our previous work [22]. The new contributions include a more complete and systematic presentation on the two-way join algorithms; and a comparative study on complexly recursive join algorithms using theory and empirical models in Spark. We use some abbreviations described in Table 1.

The rest of this paper is organized as follows. Section 2 presents the theoretical background of Apache Spark and filters. Section 3 shows the join algorithms in MapReduce. Section 4 provides a comparison of join algorithms through cost models and experiments. The conclusion of the paper is presented in Sect. 5.

Table 1 List of abbreviation

Abbreviation	Algorithm
MSJ	Map-side join
RSJ	Reduce-side join
BCJ	Broadcast join
BFJ	Bloom join
IBJ	Intersection bloom join
RRS	Recursive join (with reduce-side join)
RIB	Recursive join (with intersection bloom join)

Background

MapReduce

MapReduce [12] is a programming model that processes large amounts data in a parallel and distributed manner. MapReduce was first developed at Google by Jeffrey Dean and Sanjay Ghemawat. Their motivation comes from a multitude of calculations involving a large amount of input data that are performed daily at Google. The input data to the calculations is enormous and requires distributed processing across hundreds or thousands of machines to get results in a reasonable amount of time. When the system is distributed, some problems with distributed data and parallel computing occur that make it difficult for programmers to focus on simple calculations. Therefore, MapReduce is a solution that helps programmers focus on performing computations without regard to the complexities of distributed systems such as fault tolerance, load balancing, and data partitioning.

The MapReduce model consists of two main functions: map and reduce. A map function processes the input data to generate the intermediate data in form of key-value pairs. A reduce function will work on a list of values with the same key to produce the results. An example of using MapReduce model to process the word-count is presented in Fig. 1.

Apache Spark

Apache Spark [32] is an open source computing framework on a cluster, used for fast data analysis that satisfies two criteria, which are fast in execution and fast in read/write data. Spark was first developed in 2009 by AMPLab at the University of California. In 2013, Spark was donated to the Apache Software Foundation. Spark does not have its file system thus it uses other distributed file systems such as Hadoop

[1] https://hadoop.apache.org.

[2] https://spark.apache.org.

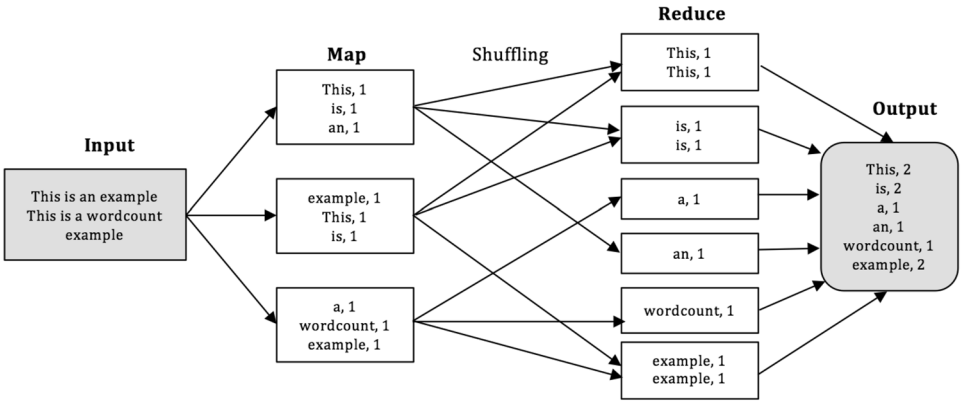


Fig. 1 Example of MapReduce processing model

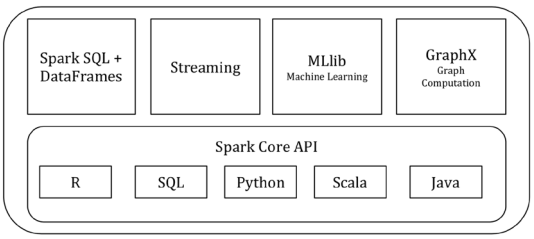


Fig. 2 Spark components

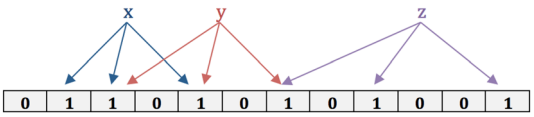


Fig. 3 Bloom filter with $m = 12$ and $k = 3$

HDFS[3], Cassandra[4], Hbase[5], and Amazon S3[6]. Spark allows dividing tasks into small pieces to run in memory of different computing nodes to exploit the fast processing speed.

Spark has several components (Fig. 2), in which Spark Core is the main component to build all other functions on it. Spark Core supports the most basic functions such as scheduling tasks, managing memory, and error recovery. It provides in-memory computing capabilities to provide speed, a general execution model to support a wide range of applications, and provides APIs for popular programming languages such as Java, Scala, and Python. Spark SQL is a Spark module for handling structured data. It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine. Spark Streaming enables powerful interactive and analytical applications on both streaming and batch data. It easily integrates with many popular data sources, including HDFS, Flume[7], Kafka[8], and Twitter[9]. Spark MLlib is a scalable machine learning library that supports to build higher-level algorithms. GraphX is a new component that allows users to build interactions, transform, and interpret graphical data.

Spark solves memory management issues by using Resilient Distributed Datasets (RDDs) with two types of operations (Transformations and Actions). A transformation generates RDDs from existing RDD and an action will return the results to the driver program after performing computations on RDDs. Transformation is a "lazy" operation that does not execute intermediately. A lineage graph is incrementally built when applying transformations. It includes all the parents RDD and the final RDD. Once a related action is performed, the transformations are executed.

An RDD will be re-initializes in each action so that it will take a lot of time if we encounter a case that an RDD is reused many times. Therefore, Spark supports a mechanism called persist or cache. Persisting an RDD, the nodes containing the RDD partitions will store it in memory, and that node will only calculate once. Spark uses a concept called "storage level" to manage the storage level of data. Spark will recalculate the missing parts of RDD if necessary.

Bloom filter and Intersection Bloom Filter

Bloom Filter (BF) [9] was introduced in 1970 by Burton Bloom, is a probability data structure used to check whether an element belongs to a collection or not. A BF structure consists of an array of m bits and k independent hash functions with each function hashes elements to a position in the m bits array. An example of a bloom filter is presented in Fig. 3 with $m = 12$ and $k = 3$. To test element z belongs to the collection S or not, we need to check all k hash positions of z in the m bits array. If all the values at those positions are

[3] https://hadoop.apache.org.
[4] https://cassandra.apache.org.
[5] https://hbase.apache.org.
[6] https://aws.amazon.com/s3/.
[7] https://flume.apache.org.
[8] https://kafka.apache.org.
[9] https://twitter.com.

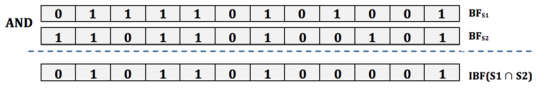


Fig. 4 IBF Using the intersection of two BFs

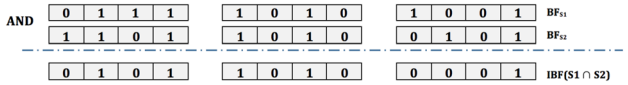


Fig. 5 IBF Using the intersection of two partitioned BFs

1 then $z \in S$. Otherwise, if there exist at least one of these positions with a value of 0 then $z \in S$.

In some cases, a hash function for multiple elements may return the same value thus an element that does not exist in the collection can also have a position with value 1. For this reason, a BF can return false positive elements, but it never returns false negative elements. A false positive element is the one identified by the BF as belonging to S, but actually it is not. A false negative element is the one identified by the BF as not in the S but in fact it does. The false positive probability of a bloom filter is calculated with Eq. 1.

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

Intersection Bloom Filter (IBF) [23, 24] is a probability data structure designed to represent the intersection of datasets. It is used to identify common elements of collections with a false positive probability. A false positive element is the element defined by IBF as belonging to the intersection two datasets but in fact it does not. There are three approaches to build up an IBF for two datasets S_1 and S_2.

- Approach 1: Using two BFs
  Use BF_{S1} for S_2 to select the common elements of S_2; and use BF_{S2} for S_1 to select the common elements of S_1. The combinings of the above results are the common elements of both datasets. The advantage of this approach is that it does not require the two filters to have the same size m and the same k hash functions.
- Approach 2: Using the intersection of two BFs
  This approach requires two BFs to have the same size m bits and k hash functions (Figs. 4 and 5). To build the IBF, we calculate the intersection of the two filters BF_{S1} and BF_{S2} by an AND bit operation ($IBF_{(S1,S2)} = BF_{S1}$ *AND* BF_{S2}).
- Approach 3: Using the intersection of two partitioned BFs
  Partitioned Bloom filter [14] is a variant of the standard BF, defined by an array of m bits divided into k separated sub-arrays with size $mp = m/k$ bits. Similar to approach 2, the $IBF_{S1,S2}$ is generated by an *AND* operation of BF_{S1} and BF_{S2}.

Join Algorithms in MapReduce

Join is an operation that combines tuples from different datasets with several conditions. It is essentially a connection between two or more datasets based on matching columns. Most data queries include the join operations as the basic operations. Thee three main types of join operations are two-way join, multi-way join, and recursive Join.

Two-way join: Given two datasets R and L, a two-way join ($R \bowtie_{k_1=k_2} L$) denotes tuples $r \in R$ and $l \in L$ such that $r.k_1 = l.k_2$ where k_1 and k_2 are the join keys in R and L.

Multi-way join: Given n datasets $R_1, R_2, ..., R_n$, a multi-way join is the two-way joins in pair ($R_1 \bowtie R_2 \bowtie ... \bowtie R_n$).

Recursive join: Given a relation $K(x, y)$ encoding a graph, a recursive join computes the transitive closure of the relation K. It requires an initialization operation and the iterations until now new result found. Recursive join repeats the join operations until the iteration no longer produces new results.

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

Of the three types of join above, two-way join is the most commonly used in data queries. To illustrate the two-way join operation in MapReduce, we consider the join operation of two datasets R and L. The join computation in MapReduce will be performed as depicted in Fig. 6 with the tuples of $R(B, F)$, (D, E), (A, C) and $L(A, D)$, (A, B), (B, C). The map phase is responsible for reading the input data blocks of R and L. Three mappers are created as shown in Fig. 6 to process three data blocks (each block will consist of several tuples). Mappers will convert the tuples into key-value pairs, which are the intermediate data.

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

Intermediate datasets 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 L to produce the results.

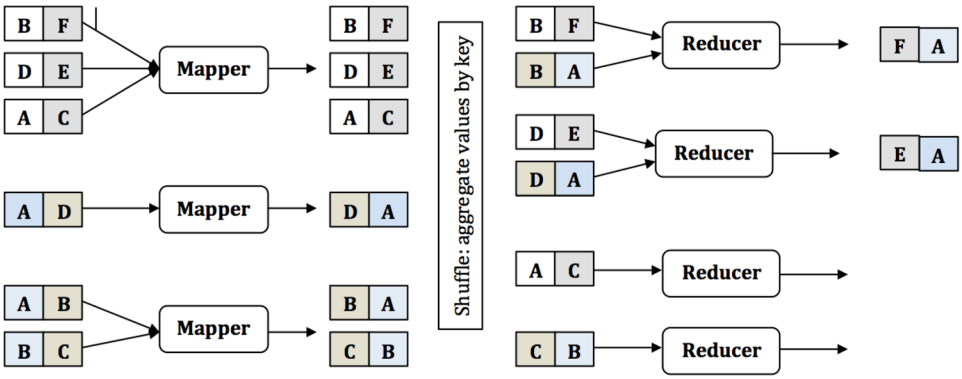


Fig. 6 Join operation in MapReduce

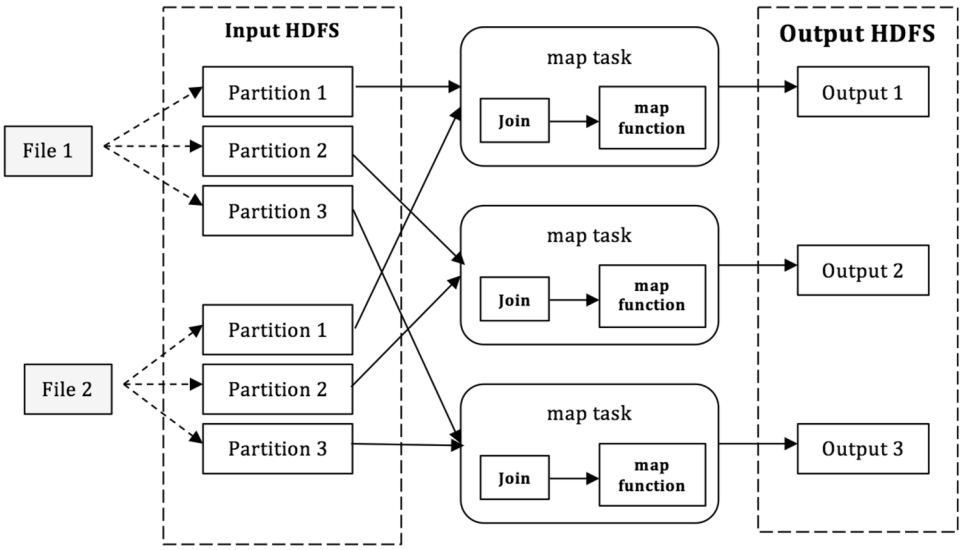


Fig. 7 Map-side join algorithm flowchart

Map-Side Join

Map-side join [15, 31] is similar to sort-merge join in Relational Database Management Systems—RDBMS. The operation is done by joining two datasets at the map stage without shuffle and reduce stages. However, this algorithm requires that the input datasets must be arranged in the order of the join keys and all datasets with the same join key must be brought together in a partition. The flow diagram of the Map-side join algorithm is in Fig. 7.

The MSJ algorithm does not generate intermediate data and does not cost for the shuffle and reduce stages since it only runs through the input data and performs join operation in the map stage. However, this algorithm requires strict input data, which means that the datasets must have the same number of partitions sorted by the join key. In order words, this can be the most efficient algorithm if the input datasets meet all requirements. Otherwise, this algorithm will cost for pre-processing the input data in accordance with the requirements. Furthermore, the algorithm needs to have two buffers containing all the same key sets of the input data, which can lead to memory overflow at computational nodes.

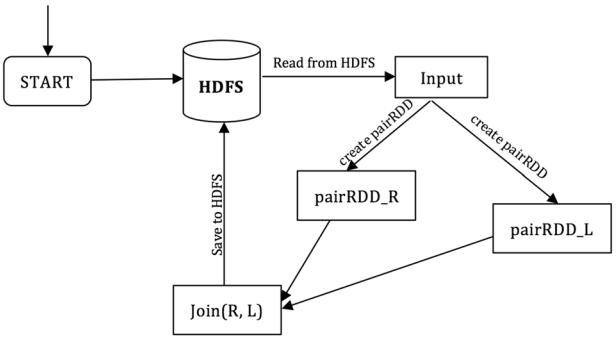


Fig. 8 Reduce-side join algorithm flowchart

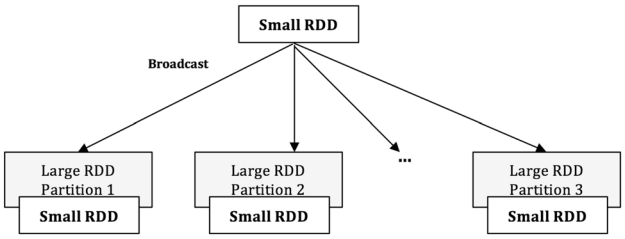


Fig. 9 Broadcast join algorithm flowchart

```
Pseudocode for Map-side join algorithm
    Input: tuples of input datasets R and L with the join columns
    k1 and k2 respectively
    Output: tuples combined by r∈R and l∈L, r.k1 = l.k2
    Init_Map() // init function for map phase
    buff_R  ←  load(partitioni_R); //loading partitioni of R
    buff_L  ←  load(partitioni_L); //loading partitioni of L
    result  ←  empty; //for storing join computation
        if (buff_R !=null & buff_L !=null) then
            for each r in buff_R do
                 for each l in buff_L do
                          result.add(pair( r, l));
        end if
    Map(k: null, v: null)
        for each t in result do
            emit(null, t);
```

Reduce-Side Join

Reduce-side join [8, 15, 31] is known as a "re-partition join", which will be performed at the reduce stage. The algorithm maps the input datasets to have intermediate data in form of key-value pairs and shuffles the data to the reducers for performing join operation. All key-value pairs with the same key join must be sent to the same reducers and sorted by the key join. The reducers then perform a combination of values with the same key. The RSJ algorithm flowchart is presented in Fig. 8.

This algorithm uses the natural way of MapReduce framework to process join operation. It is more general than the MSJ since it has no requirements on input datasets. It limits the case of memory overflow in MSJ algorithm since the buffer only contains tuples that participate in join operation of one dataset. However, this algorithm will cost more on I/O and shuffling data since it creates intermediate data at the map stage and transfers over the network. The RSJ can encounter memory overflow in case of skewed data.

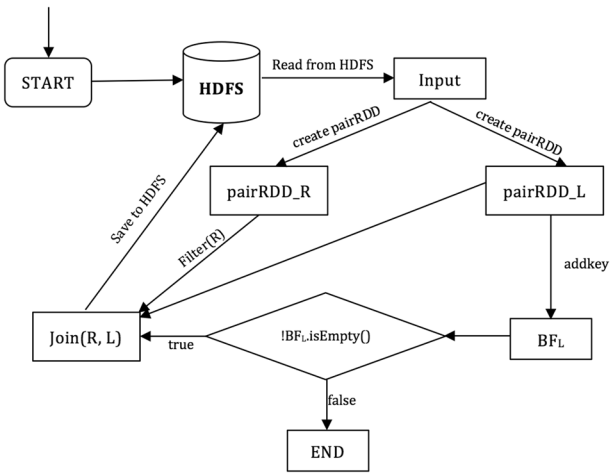


Fig. 10 Bloom join algorithm flowchart

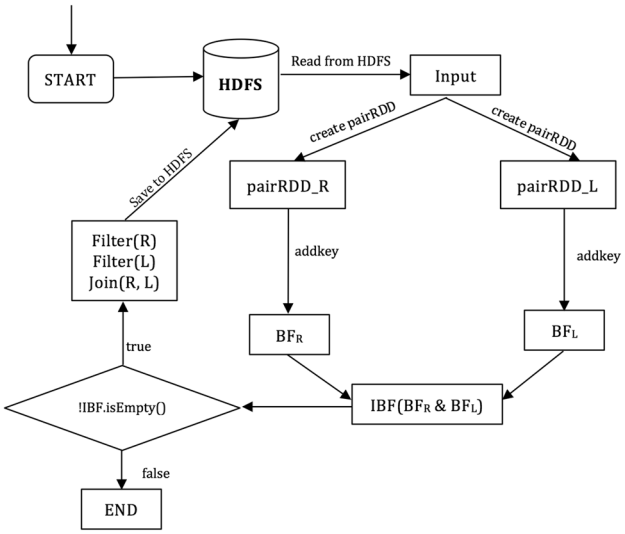


Fig. 11 Intersection bloom join algorithm flowchart

```
Pseudocode for Reduce-side join algorithm
    Input: tuples of input datasets R and L with the join columns
    k1 and k2 respectively
    Output: the tuples combined by r∈R and l∈L, r.k1 = l.k2
    Map(k: null, v: a tuple from an R or L split)
        tag  ←  a bit 0 or 1 corresponding to name of R or L;
        key  ←  extract the join key from v;
        emit(pair(key, tag), v);
    Partitioner(k': taggedkey, v: value, p: the number of reducers)
        return hash_func(k'.key) mod p;
    Init_Reduce() // init function for the reduce phase
        currentKey  ←  '0'; // for storing current key
        buff  ←  empty; // for storing tuples with same key of R
    Reduce(k': taggedkey, v': list of values v with key k')
        if k'.key != currentKey then
            clear(buff);
            currentkey = k'.key;
        endif
        if k'.tag == '0' then
            for each l in v' do
                add tuple l to buff;
        else if k'.tag == '1' then
             for each l in v' do
                 for each r in buff do
                     emit(null, pair(r, l));
         end if
```

Broadcast Join

Broadcast join [8] is similar to the algorithm that uses hash join in RDBMS. It is a special case of the MSJ algorithm but it does not require strictly structured input data. This algorithm is applicable when the size of one dataset is much smaller than the other ($|R| << |L|$). The smaller dataset will be broadcast to all mappers. An in-memory hash table will be used to store the small dataset and using table lookup for matching key join.

In this algorithm, the map stage does not produce intermediate data and does not cost for shuffle and reduce stages. The algorithm is not affected by the problem of skewed data since the smaller dataset is broadcasted to each mapper and each mapper process one partition of the larger dataset that can be fitted in memory. However, in this algorithm, one of the two input datasets is very small to be able to broadcast to all mappers (Fig. 9).

```
Pseudocode for Broadcast join algorithm
    Input: tuples of input datasets R and L with the join columns
    k1 and k2 respectively
    Output: the tuples combined by r∈R and l∈L, r.k1 = l.k2
    Init_Map() // init function for map phase
        buff_R  ←  load(R); //loading all tuples of R
    Map(k: null, v: a tuple from an L split)
        refKey  ←  extract the join column from v
        for each r in buff_R do
            joinKey  ←  extract the join column from r
            if (joinKey == refKey) then
                emit(null, pair(r,v));
            end if join algorithm
```

Bloom Join

Bloom join (BFJ) [16, 20, 23] is a join algorithm improved from RSJ algorithm using Bloom filter, an effective space-efficient data structure. The algorithm is also derived from the Bloom Join approach in the RDBMS, which is an improvement of the semi-join method by using filters. The BFJ algorithm can be implemented by using two MapReduce jobs. One is building BF_L storing all keys of the dataset L. The other is using BF_L to filter non-join tuples in R before processing join operation (Fig. 10).

This algorithm has achieved certain efficiency for filtering unnecessary data from one of the two input datasets that are not involved in the join operation. The size of the filter does not depend on the number of join keys. However, there is a need of pre-processing step to build up the filter by scanning one input dataset, which presents extra costs. In particular, this approach also accepts a false positive probability (Eq. 1) for filtering data in one dataset (Fig. 11).

```
Pseudocode for Bloom join algorithm
    Input: tuples of input datasets R and L with the join columns
    k1 and k2 respectively
    Output: the tuples combined by r∈R and l∈L, r.k1 = l.k2
    Init_Map() // init function for map phase
        globalBF  ←  load(BF_L.uid); //loading filter
    Map(k: null, v: a tuple from an R or L split)
        tag  ←  a bit 0 or 1 corresponding to name of R or L
        key  ←  extract the join key from v
        if (tag == '1' || (tag == '0' & key in globalBF ) ) then
            emit(pair(key, tag), v)
    Partitioner(k': taggedkey, v: value, p: the number of reducers)
        return hash_func(k'.key) mod p
    Init_Reduce() // init function for reduce phase
        currentKey  ←  '0'; // for storing current key
        buff  ←  empty; // for storing tuples with same key of R
    Reduce(k': taggedkey, v': list of values v with key k')
        if k'.key != currentKey then
            clear(buff);
            currentkey = k'.key
        endif
        if k'.tag == '0' then
            for each l in v' do
                add tuple l to buff
        else if k'.tag == '1' then
            for each l in v' do
                for each r in buff do
                    emit(null, pair(r, l))
        end if
```

Table 2 Semi-Naive algorithm

(1)	$F = K, \Delta F = K$;
(2)	**while** $\Delta F \neq 0$ **do**
(3)	$O = \Delta F \bowtie K$;
(4)	$\Delta F = O - F$;
(5)	$F = F \cup \Delta F$;
(6)	**end**

Intersection Bloom Join

Intersection filter-based Bloom join [23] is an improved algorithm from BFJ algorithm with the use of Intersection Bloom Filter (IBF) instead of the standard Bloom filter. The IBJ algorithm can be implemented by using two MapReduce jobs. One is building $IBF_{R,L}$, which is the intersection of two filters BF_L and BF_R. The other is using $IBF_{R,L}$ to filter non-join tuples in the two datasets before processing join operation.

This algorithm uses the IBF to filter data on both input datasets to eliminate most of the tuples that are not participating in join operation. In consequence, it significantly reduces the costs of join operation. It has been shown to be more efficient than other join algorithms [24]. Therefore, the IBJ will be the most optimal algorithm compared to the current join algorithms.

```
Pseudocode for Intersection bloom join algorithm
    Input: tuples of input datasets R and L with the join columns
    k1 and k2 respectively
    Output: the tuples combined by r∈R and l∈L, r.k1 = l.k2
    Init_Map() // init function for map phase
        IBF.uid  ←  load(IBF.uid); // loading intersection filter
    Map(k: null, v: a tuple from an R or L split)
        tag  ←  a bit 0 or 1 corresponding to name of R or L;
        key  ←  extract the join key from v;
        if (key in IBF.uid ) then
            emit(pair(key, tag), v);
        endif
    GroupComparator(taggedKey1: taggedkey, taggedKey2: taggedkey)
        res = compare(taggedKey1.key, taggedKey2.key);
        if (res == 0) then
            res = compare(taggedKey1.tag, taggedKey2.tag);
        endif
        return res;
    Partitioner(k': taggedkey, v: value, p: the number of reducers)
        return hash_func(k'.key) mod p;
    Init_Reduce() // init function for reduce phase
        currentKey  ←  '0'; // for storing current key
        buff  ←  empty; // for storing tuples with same key of R
    Reduce(k': taggedKey, v': list of values v with key k')
        if k'.key != currentKey then
            clear(buff);
            currentkey = k'.key;
            endif
            if k'.tag == '0' then
                for each l in v' do
                    add tuple l to buff;
            else if k'.tag == '1' then
                for each l in v' do
                     for each r in buff do
                         emit(null, pair(r, l));
            end if
```

Table 3 Parameters use in cost model

Parameters	Meaning
$\|R\|$	Size of dataset R
$\|L\|$	Size of dataset L
c_l	Cost for read/write data locally
c_r	Cost for read/write data remotely
c_t	Cost for transferring data from node to node
e_R	Number of executors for dataset R
e_L	Number of executors for dataset L
$e = e_R + e_L$	Total number of executors
$B + 1$	Size of sorting buffer on pages
m	Size of Bloom filter (bits)
f_L	False positive probability of BF L
$f_{(R,L)}$	False positive probability of IBF between R and L
δ_L, δ_R	Join rate of two datasets R, L
$\|D\|$	Size of intermediate dataset for join task
$\|O\|$	Size of result dataset for join task
C_{pre}	Total number cost for pre-processing task
C_{read}	Total cost for read data
C_{tr}	Total cost for transferring data
C_{write}	Total cost for writing data
$\|K\|$	Size of dataset K
$\|F\|$	Size of dataset F
$\|\Delta F_i\|$	Size of the incremental dataset at iteration i
$\|D_i\|$	Size of intermediate dataset of join job J_i
$\|D_i^+\|$	Size of intermediate dataset of incremental computing job I_i
$\|O_i\|$	Size of intermediate result dataset of join job at iteration i
l	The number of iterations
$C_{read}(J_i)$	Total cost of reading incremental dataset ΔF for join job
$C_{tr}(J_i)$	Total cost of transmitting intermediate data between nodes for join job
$C_{cache}(J_i)$	Total cost of reading data partitions of K cached at the reducers
$C_{read}(I_i)$	Total cost of reading result dataset O_i for incremental computing job
$C_{tr}(I_i)$	Total cost of transmitting intermediate results
$C_{cache}(I_i)$	Total cost for local reading data partitions of F_{i-1} at the reducers
$C_{write}(I_i)$	Total cost for writing final result to HDFS

Recursive Join

Semi-Naive is a common and suitable algorithm for performing recursive join in MapReduce. The Semi-Naive algorithm for computing transitive closure is presented in Table 2. K denotes the original graph and F will contains all tuples in the transitive closure of the graph until the end of the algorithm. ΔF contains the new tuples found in the previous iteration. The input data of the algorithm is dataset K with the join columns k_1 and k_2. The output are the tuples combined by $t_1 \in K$ and $t_2 \in K$ such that $t_1.k_1 = t_2.k_2$.

The evaluation of recursive join using Semi-Naive algorithm consists of two repeated jobs, which are join job and incremental computing dataset job. In each iteration, the join job will perform join operation between ΔF and K to generate a result set O ($\Delta F \bowtie K$). The incremental computing dataset job will eliminate the duplicated tuples in O will all the tuples founded ($O - F$). In line (3) of the algorithm, we can see that the dataset K is unchanged over iterations. Thus, this dataset can be cached using Spark caching mechanism to speed up the computation. In addition, there is the appearance of two-way join in the same line. In this work, we use two approaches for processing recursive join with Semi-Naive algorithm: Recursive join with Reduce-side join (RRS) and Recursive join with Intersection Bloom Join (RIB). The intersection bloom filter is used to eliminate redundant data in the join job between ΔF and K.

Evaluation

Cost Model for Two-Way Join

Join computation cost is the total cost of several stages including cost for pre-processing task (C_{pre}), cost for reading data (C_{read}), cost for transferring data (C_{tr}), and cost for writing data (C_{write}). The general cost model for join computation of two datasets can be described as in Eq. 2. The algorithms are implemented in a general model without any restrictions on input datasets. The parameters of the cost model are clarified in Table 3.

- The cost of pre-processing task will be depended on the algorithms used for join operation.
- The cost of reading data includes the cost of remotely reading R and L.
- The cost of writing data is the cost of remotely writing the result O.
- Them cost of transferring data between nodes is the cost of transferring intermediate dataset D. The intermediate dataset will be calculated depending on the type of two-way join algorithms.

$$C = C_{pre} + C_{read} + C_{tr} + C_{write} \tag{2}$$

where:

- $C_{read} = c_r.|R| + c_r.|L|$
- $C_{tr} = c_t.|D|$
- $C_{write} = c_r.|O|$
- $|D|$ *: depending on join algorithms*
- C_{pre}*: depending on join algorithms*

In comparing the effectiveness of the join algorithms, we will consider the cost of pre-processing task C_{pre} and the intermediate dataset D since the other costs stay the same.

Map-Side Join

MSJ splits two datasets into the same partitions and sort in order of the join keys before performing join operation. The pre-processing task needs to transfer tuples of the two datasets to the same partitions.

$$C(MSJ) = C_{pre} + C_{read} + C_{tr} + C_{write} \tag{3}$$

where:

- $C_{pre} = c_l.|D|.2.([log_B|D| - log_B(e)] + log_B(e))$
- $C_{tr} = c_t.|D|$
- $|D| = |R| + |L|$

Reduce-Side Join

RSJ implements join operations by joining data of the tuples with the same keys in reduce stage. There is no need for pre-processing task however the intermediate data (key-value pairs) of both dataset needs to be transferred to the reducers.

$$C(RSJ) = C_{pre} + C_{read} + C_{tr} + C_{write} \tag{4}$$

where:

- $C_{pre} = 0$
- $C_{tr} = c_t.|D|$
- $|D| = |R| + |L|$

Broadcast Join

BCJ join two datasets by broadcasting all the records of the small dataset to all reducers. This is a special case that having a dataset, which is much smaller than the other dataset. We assume that $|R| >> |L|$. The whole dataset $|L|$ needs to be sent to all reducers.

$$C(BCJ) = C_{pre} + C_{read} + C_{tr} + C_{write} \tag{5}$$

where:

- $C_{pre} = 0$
- $C_{tr} = c_t.|L|$

Bloom Join

BFJ implements join operation by filtering redundant data in one of the two datasets using Bloom filter.

$$C(BFJ) = C_{pre} + C_{read} + C_{tr} + C_{write} \tag{6}$$

where:

- $C_{pre} = 2.c_t.m.e_L$
- $C_{tr} = c_t.|D|$
- $|D| = |L| + \delta_L.|R| + f_L.(1 - \delta_L).|R|$

The pre-processing task calculates the BF for a dataset, e.g. L. Each executor for dataset L adds keys to the m bits BF and sends the BF to Spark driver. The driver performs OR operation to combine the BF from e_L executors and then broadcast the final BF to the executors. Since there is a BF of L, we use that BF for filtering dataset R. The intermediate data of BFJ is now the whole dataset L plus the tuples from dataset R that can participate in join operation with a false positive probability of f_L.

Intersection Bloom Join

IBJ joins two datasets by filtering redundant data in both datasets with IBF.

$$C(IBJ) = C_{pre} + C_{read} + C_{tr} + C_{write} \quad (7)$$

where:

- $C_{pre} = 2.c_t.m.e$
- $C_{tr} = c_t.|D|$
- $|D| = \delta_L.|R| + f_{(R,L)}.(1-\delta_L).|R| + \delta_R.|L| + f_{(R,L)}.(1-\delta_R).|L|$

The pre-processing task calculates the IBF for both dataset L and R. The number of executors e adds keys to the m bits BF_L and BF_R. The IBF of the two dataset is a combination of the two BF with AND operation, $IBF = BF_L\ AND\ BF_R$. The IBF will be sent to Spark driver. The driver broadcasts the final IBF to the executors. The intermediate data of BFJ includes the tuples from dataset L plus the tuples from dataset R. Those tuples can be participated in join operation with a false positive probability of $f_{(R,L)}$.

Analyze the Cost Model of Join Algorithms

The MSJ performs pre-processing task that sorts the join keys of all datasets in order and all the same keys must be put on the same partitions. The RSJ does not have pre-processing cost however it costs for the shuffling data between nodes. From formula 3 and formula 4, the total costs of the two algorithms are almost the same except that the MSJ needs to have a cost for sorting data in pre-processing task. Thus, the total cost for computing MSJ is larger than that of RSJ, $C(MSJ) > C(RSJ)$.

It is the same case in comparison MSJ and BCJ. The MSJ has a pre-processing task for sorting join keys while BCJ does not. Thus, the total cost for computing MSJ is larger than BCJ, $C(MSJ) > C(BCJ)$. In comparing between RSJ and BCJ, we can base on the intermediate data transferring over the network. The intermediate data generated by RSJ is $|D| = |R| + |L|$ while that of BCJ is $|D| = |L|$. Therefore, we can conduct that $C(RSJ) > C(BCJ)$. However, it should be noted that the BCJ is only applicable if the dataset $|L|$ is much smaller than the other.

The intermediate data ($|D|$) is an important parameter in comparing the join algorithms since the data transferring through the network will affect the cost of the algorithms. We have:

$$|D|_{RSJ} = |L| + |R|\ (*)$$
$$|D|_{BFJ} = |L| + \delta_L.|R| + f_L.(1-\delta_L).|R|\ (**)$$
$$|D|_{IBJ} = \delta_R.|L| + f_{(R,L)}.(1-\delta_R).|L| + \delta_L.|R| + f_{(R,L)}.(1-\delta_L).|R|\ (***)$$

Table 4 Datasets used in the experiments

Input	Dataset 1 (GB)	Dataset 2 (GB)	Total (GB)
Test 1	5	10	15
Test 2	10	10	20
Test 3	10	20	30
Test 4	20	20	40
Test 5	1	10	11
Test 6	1	20	21

In comparing between RSJ and BFJ, it is clearly seeing that the intermediate data generated by RSJ is greater than that of BFJ. The value of false positive probability of BF (f_L) is much smaller than 1 (0.001 in our experiments) and the join rate between two datasets is usually small so that $\delta_L.|R| + f_L.(1-\delta_L).|R| << |R|$. Thus, from (*) and (**) we can conduct that $C(RSJ) > C(BFJ)$.

Similarly, the intermediate data generated from IBJ is smaller tha BFJ. The value of false positive probability of IBF is much smaller than 1 (0.001 in our experiments). The redundant data is filer out in both datasets L and R instead of one dataset as in BFJ ($\delta_R.|L| + f_{(R,L)}.(1-\delta_R).|L| << |L|$). Thus, from (**) and (***) we can conclude $C(BFJ) > C(IBJ)$.

After analyzing the cost model between algorithms on general datasets, we can conclude that:

$$C(MSJ) > C(RSJ) > C(BFJ) > C(IBJ) \quad (8)$$

In special case, the dataset L is much smaller than the dataset R, we will use the BCJ algorithm and this will cost less than the other algorithms.

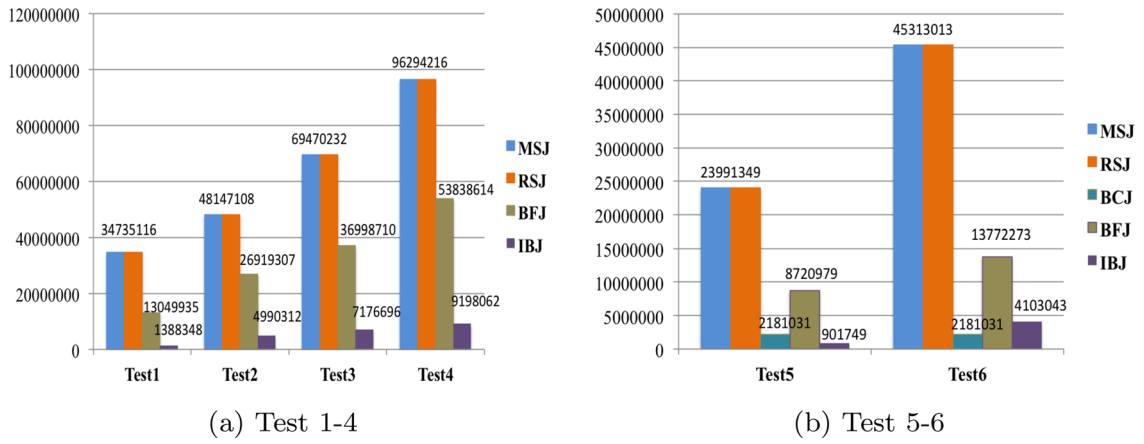
Cost Model for Recursive Join

The general cost model for recursive join in Spark is presented in Eq. 9. The total cost includes costs for preprocessing, join job, incremental computing dataset job, and writing results. The meaning of parameters used in the cost model is also shown in Table 3.

$$\begin{aligned} C =& C_{pre} + \sum_{n=1}^{l}(C_{read}(J_i) + C_{tr}(J_i) + C_{cache}(J_i)) \\ &+ \sum_{n=1}^{l}(C_{read}(I_i) + C_{tr}(I_i) + C_{cache}(I_i)) + C_{write} \end{aligned} \quad (9)$$

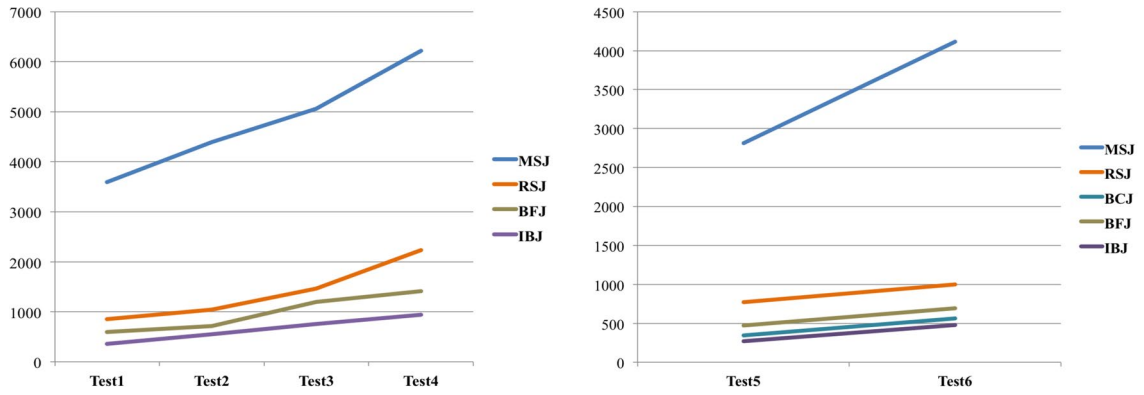
where:

- $C_{pre} = c_r.|K| + c_l.|K| + c_t.|K| + c_l.|K|$
- $C_{read}(J_i) = c_l.|\Delta F_{i-1}|$
- $C_{tr}(J_i) = c_t.|D_i|$



(a) Test 1-4 (b) Test 5-6

Fig. 12 Intermediate results in records



(a) Test 1-4 (b) Test 5-6

Fig. 13 Execution time in seconds

(a) Intermediate results in records (b) Execution time in seconds

Fig. 14 Recursive join algorithms

- $C_{cache}(J_i) = c_l.|O_i|$
- $C_{read}(I_i) = c_r.|O_i|$
- $C_{tr}(I_i) = c_t.|D_i^+|$
- $|D_i^+| = |\Delta F_i| = |O_i|$
- $C_{cache}(I_i) = c_l.|\Delta F_i|$
- $C_{write} = c_r.|F|$

The cost for preprocessing is the total cost to read, map, shuffle, and cache dataset K at the reducers since this dataset is unchanged over the iterations. The $C_x(J_i)$ represents costs for join job and the $C_x(I_i)$ represents costs for incremental computing dataset job. We will consider the cost of recursive join with reduce-side join C(RSS) and the cost of recursive join with intersection bloom join C(RIB). The different between this two approaches is the join job. The intermediate data $|D_i|$ of the two approaches are as follows:

$$|D_i|_{RSS} = |\Delta F_{i-1}|$$
$$|D_i|_{RIB} = \delta_K.|\Delta F_{i-1}| + f_{(\Delta F_{i-1},K)}.(1-\delta_K).|\Delta F_{i-1}|$$

The use of intersection bloom filter will eliminate the redundant data not participating in join operation. Thus, we can conclude that the cost of RIB is smaller than the cost of RSS.

$$C(RSS) > C(RIB) \tag{10}$$

Experiments

Experiments with the join algorithms presented above will be performed on a Spark cluster. The goal of the experiments is to evaluate the efficiency of the join algorithms on large-scale datasets in MapReduce. The efficiency of these algorithms depends on the amount of intermediate data generated during join computation. The amount of intermediate data will determine the communication cost, which is the biggest cost that affects the execution time of the join algorithms. Therefore, we discover the amount of intermediate data generated from the join algorithms in the experiments and decide the suitability of the experimental results with the cost model that we have proposed.

Spark Cluster

The experiments are conducted on a cluster of 14 computing nodes (1 master and 13 slaves). Each computer is configured with 4 CPUs Intel Core i5 3.2 Ghz with 4 GB RAM, and 500 GB HDD. A node is installed Ubuntu 14.04 LTS 64 bits operating system. The installed applications are Java 1.8, Hadoop 2.7.1, and Spark 2.1. This cluster is provided by The Mobile Network and Big Data Laboratory of College of Information and Technology, Can Tho University. The Spark cluster is configured with 13 executors, 3 cores per executor, and 2.5 GB memory per executor.

The data used for the experiments is the standard data generated by PUMA: Purdue MapReduce Benchmarks Suite [2]. The datasets are stored in a text file format with each line has a maximum of 39 fields separated by commas, each field with 19 characters. The join keys used in the experiments are the 5th column (the first dataset) and the 4th column (the set second dataset).

Results

The experiments use five algorithms, which are Map-side join, Broadcast join, Reduce-side join, Bloom join, and Intersection Filter-based Bloom join. We have six test datasets that is described in Table 4. For each algorithm, we conduct experiments twice to have the average execution time and the number of intermediate data.

Figure 12a and b show the amount of intermediate data that needs to be transported across the network of the join algorithms. These results clearly show the effectiveness between join algorithms and conform the proposed cost models. As presented in the two figures, the amount of intermediate data of MSJ is equal to that of RSJ ($|D| = |L| + |R|$) and greater than the amount of intermediate data of BFJ after filter out redundant data in one dataset (R). At the same time, IBJ has the smallest amount of intermediate data since it significantly reduces the amount of redundant data that does not participate in the join operation on both datasets (R, L). BCJ always has the same amount of intermediate data because it broadcasts the smaller dataset (L) to all mappers.

The amount of intermediate data transmitted over the network affect communication cost, which will affect the execution time of the join algorithms. MSJ is the best option in case that we have datasets partitioned and sorted in advance. However, in Fig. 13a and b, MSJ has a longer execution time. It is because the datasets used in the experiments are the standard datasets without sorting and partitioning in advance. These two tasks degrade the overall performance of the algorithm. RSJ has smaller execution time than MSJ and has longer execution time than BFJ. The IBJ is the one with the best performance. It is reasonable since this algorithm uses the intersection bloom filter to significantly reduce the redundant data. Regarding formula 8, the experiment results are appropriate with the cost models presented above. The cost models can be used as a scientific basis for estimation and prediction before implementations.

We also conduct experiments with the recursive join with two approaches, RSS and RIB. The datasets for the experiments have the size of 5 GB, 10 GB, 20 GB, and 30 GB. Figure 14a shows the intermediate data to be transmitted

over the network. It is clear that the reduce of redundant data by intersection bloom filter helps to reduce the intermediate data. Figure 14b shows the different between the two approaches in term of processing time. This result is appropriate with the cost models as can be seen in formula 10. The larger datasets give better performance since it is costly for processing filters in small datasets.

Conclusion

There are many algorithms that are proposed for performing join operations on large-scale datasets. Therefore, it is necessary to provide users an overview and evaluations on these join algorithms. This study fully evaluates common two-way joins and recursive joins in MapReduce with Spark. This work provides: (1) an investigation of common join algorithms on large datasets in MapReduce using Spark, with the advantages and disadvantages pointed out; (2) the cost models for the join algorithms in Spark which is an important theoretical basis for evaluating and comparing the algorithms; (3) the experiments of the two-way joins and recursive joins in Spark.

A join operation on large datasets often is a costly, time-consuming, and resource-intensive operation but commonly used in Spark, so it is worth making this evaluation. Through the cost models and the experiments, we have demonstrated the advantages and disadvantages of the current join algorithms and the common recursive join algorithm. In general, joins based on Intersection Bloom Filters dominate over the other joins because they require no special input data and minimize non-joining data as well as communication costs. It is also worthy to consider the context of skewed data since it is common in data science.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Afrati FN, Ullman JD. Transitive closure and recursive datalog implemented on clusters. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12, pp 132–143. ACM, New York, NY, USA 2012. https://doi.org/10.1145/2247596.2247613.
2. Ahmad F. Puma benchmarks and dataset downloads 2011. URL https://engineering.purdue.edu/~puma/datasets.htm. Last Accessed: 05 Apr 2019.
3. Al-Badarneh A. Join algorithms under apache spark: Revisited. In: Proceedings of the 2019 5th International Conference on Computer and Technology Applications, ICCTA 2019. Association for Computing Machinery, New York, NY, USA 2019, pp 56–62.
4. Al-Badarneh AF, Rababa SA. An analysis of two-way Equi-join algorithms under Mapreduce. J King Saud Univ Comp Inform Sci. 2020. https://doi.org/10.1016/j.jksuci.2020.05.004.
5. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, et al. Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 15. Association for Computing Machinery, New York, NY, USA 2015, pp. 1383–1394.
6. Bancilhon F. Naive evaluation of recursively defined relations. In: On knowledge base management systems. Berlin: Springer; 1986. p. 165–78.
7. Bancilhon F, Ramakrishnan R. An amateur's introduction to recursive query processing strategies. SIGMOD Rec. 1986;15(2):16–52. https://doi.org/10.1145/16856.16859.
8. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y. A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 10. Association for Computing Machinery, New York, NY, USA 2010, pp 975–986.
9. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM. 1970;13(7):422–6.
10. Bratbergsengen K. Hashing methods and relational algebra operations. In: Proceedings of the 10th International Conference on Very Large Data Bases, VLDB 84. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 1984, pp 323–333.
11. Chen S, Ailamaki A, Gibbons PB, Mowry TC. Improving hash join performance through prefetching. ACM Trans Database Syst. 2007;32(3):17.
12. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13.
13. Gribkoff E. Distributed algorithms for the transitive closure 2013.
14. Kirsch A, Mitzenmacher M. Less hashing, same performance: building a better bloom filter. Random Struct Algorithms. 2008;33(2):187–218.
15. Lee KH, Lee YJ, Choi H, Chung YD, Moon B. Parallel data processing with Mapreduce: a survey. SIGMOD Rec. 2012;40(4):11–20.
16. Lee T, Kim K, Kim HJ. Join processing using bloom filter in Mapreduce. In: Proceedings of the 2012 ACM Research in Applied Computation Symposium, RACS 12. Association for Computing Machinery, New York, NY, USA 2012, pp 100–105.
17. Lin X, Orlowska ME. An efficient processing of a chain join with the minimum communication cost in distributed database systems. Distrib Parallel Databases. 1995;3(1):69–83.
18. Mackert LF, Lohman GM. R* optimizer validation and performance evaluation for distributed queries. In: Proceedings of the 12th International Conference on Very Large Data Bases, VLDB 86. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 1986, pp 149–159.
19. Mehta T, Mangla N, Guragon G. A survey paper on big data analytics using map reduce and hive on hadoop framework a survey paper on big data analytics using map reduce and hive on hadoop framework 2016.
20. Michael L, Nejdl W, Papapetrou O, Siberski W. Improving distributed join efficiency with extended bloom filter operations. In: Proceedings of the 21st International Conference on Advanced Networking and Applications, AINA 07. IEEE Computer Society, USA 2007, pp 187–194.
21. Mishra P, Eich MH. Join processing in relational databases. ACM Comput Surv. 1992;24(1):63–113.

22. Phan AC, Phan TC, Trieu TN. A comparative study of join algorithms in spark. In: International Conference on Future Data and Security Engineering. Springer, 2020, pp 185–198.
23. Phan TC, d'Orazio L, Rigaux P. Toward intersection filter-based optimization for joins in Mapreduce. In: Proceedings of the 2nd International Workshop on Cloud Intelligence, Cloud-I 13. Association for Computing Machinery, New York, NY, USA 2013.
24. Phan TC, d'Orazio L, Rigaux P. A theoretical and experimental comparison of filter-based equijoins in mapreduce. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XXV. Springer 2016, pp 33–70.
25. Rababa S, Al-Badarneh A. Optimizations for filter-based join algorithms in Mapreduce. J Intell Fuzzy Syst. 2021;40:1–18 (**Preprint**).
26. Shaw M, Koutris P, Howe B, Suciu D. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In: International Datalog 2.0 Workshop. Springer 2012, pp 165–176.
27. Tan KL, Lu H. A note on the strategy space of multiway join query optimization problem in parallel systems. ACM SIGMOD Rec. 1991;20(4):81–2.
28. Van Hieu D, Smanchat S, Meesad P. Mapreduce join strategies for key-value storage. In: 2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), 2014, pp 164–169.
29. Warren HS Jr. A modification of Warshall's algorithm for the transitive closure of binary relations. Commun ACM. 1975;18(4):218–20. https://doi.org/10.1145/360715.360746.
30. Warshall S. A theorem on Boolean matrices. J ACM. 1962;9(1):11–2. https://doi.org/10.1145/321105.321107.
31. White T. Hadoop: the definitive guide. 4th ed. Newton: O'Reilly Media Inc; 2015.
32. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10. USENIX Association, USA 2010, p 10.