Al-Badarneh et al., 2016

Volume 2 Issue 1, pp. 200-213

Date of Publication: 19th December, 2016

DOI- https://dx.doi.org/10.20319/mijst.2016.s21.200213

This paper can be cited as: Al-Badarneh, A., Al-Rudaini, M., Ali, F., & Najadat, H. (2016). Index-Based Join in Mapreduce using Hadoop Mapfiles. MATTER: International Journal of Science and Technology, 2(1), 200-213.

INDEX-BASED JOIN IN MAPREDUCE USING HADOOP MAPFILES

Amer Al-Badarneh
Jordan University of Science and Technology, Irbid, Jordan
amerb@just.edu.jo

Mohammed Al-Rudaini
Jordan University of Science and Technology, Irbid, Jordan
maalrudaini15@cit.just.edu.jo

Faisal Ali
Jordan University of Science and Technology, Irbid, Jordan
faali14@cit.just.edu.jo

Hassan Najadat
Jordan University of Science and Technology, Irbid, Jordan
najadat@just.edu.jo

Abstract

Map Reduce stays an important method that deals with semi-structured or unstructured big data files, however, querying data mostly needs a Join procedure to accumulate the desired result from multiple huge files. Indexing in other hand, remains the best way to ease the access to a specific record(s) in a timely manner. In this paper the authors are investigating the performance gain by implementing Map File indexing and Join algorithms together.

Keywords

Hadoop, BigData, MapReduce, Join Algorithms, Indexing.

1. Introduction

The increased popularity of using social networks and smart devices led to the age of big data where huge amount of shared multi-type data, stored as un-relational unstructured or semi-structured data files, the term "big data" corresponds to exceptionally large amount of data that cannot be handled by any database suits.

Apache Hadoop (Apache™ Hadoop®, n.d.): an open-source parallel and distributed framework for storing and processing extremely large datasets (big data) within all Hadoop cluster nodes using wide range of programming modules which included in the Hadoop Libraries, the Hadoop framework also detect and fix application runtime errors overall cluster nodes.

Hadoop framework (Apache™ Hadoop®, n.d.) includes the following modules:

- Hadoop Common: common libraries that support the other Hadoop components.
- Hadoop Distributed File System (HDFS): distributed file system offering high-speed access to huge data files.
- Hadoop YARN: framework for scheduling tasks, and managing cluster resources.
- Hadoop MapReduce: YARN-based system for large Datasets parallel processing.

Hadoop MapReduce: a software framework to aid the development of parallel and distributed processing projects working on an extremely big amounts of data on huge clusters in a reliable, fault-tolerant scheme.

HBase (Khetrapal & Ganesh, 2006) (Prasad & Agarwal, 2016): is an Apache open source Hadoop project which works as a parallel and distributed storage to host big data tables where data is logically structured as files, records, and attributes that may have multiple types for the same record key.

The Map File object (Khetrapal & Ganesh, 2006) (Prasad & Agarwal, 2016) (Class MapFile, n.d.) (Hadoop I/O: Sequence, Map, Set, Array, BloomMap Files, 2011) is a HDFS directory encloses two sequence files (Data, Index), created from the original file.

The Data file has all the key and value records, however, all keys stored in a sorted manner, in fact, the append operation check for this condition, in addition, the task scheduler can request the read of data file blocks into memory as necessary.

The Index file has a key and a value which holds the starting byte address of the data file block in the HDFS, in other words, the index file doesn’t enclose all the data file keys, except only a less portion of the keys, similarly, the task scheduler requests the read of the entire index file into the local memory of each map node.

1.1 Join Algorithms

1.1.1 Reduce Side Join (Pigul, 2012)

It is the phase in which data pre-processing done, the supreme general algorithm, with no restriction on data, it contains two phases, the first phase passes the data through network from one phase to another, and passing the information concerning data sources over the network nodes, hence, for these two factors, it is the greatest time consuming algorithm.

In this algorithm, the core improvement is to reduce the data transfer as much as possible, hence, there are three algorithms in this category:

- General Reduce-Side Join.
- Optimized Reduce-Side Join.
- Hybrid Hadoop Join.

1.1.1.1 General Reducer-Side Join (Pigul, 2012)

Also known as GRSJ its pseudo code shown in Figure 1 this algorithm has two phases, Map and Reduce, in Map Phase date read from sources and tagged with the source value.

The Reduce Phase joins the data tagged with the same key, although with different tags, which in fact, stays a problem as the reducer should have enough memory for all the records with the same key.

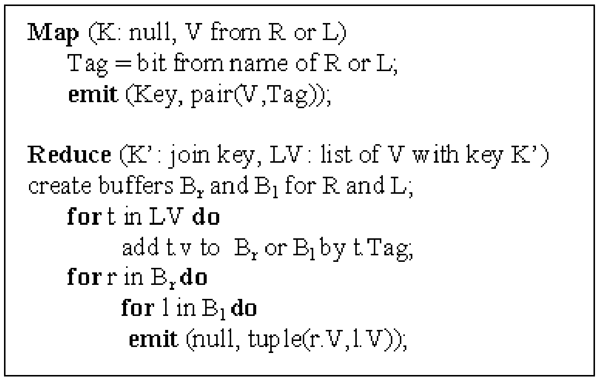
1.1.1.2 Optimized Reducer-Side Join (Pigul, 2012)

An improvement over the past algorithm by overriding sorting and key grouping, also known as ORSJ, its pseudo code shown in Figure 2, in this algorithm the value of first tag followed by the second tag, which only needs buffering for one input set as splitting remains on the key only, in addition to the attached key.

1.1.1.3 The Hybrid Hadoop Join (Pigul, 2012)

This algorithm combines Map-Side and Reduce-Side, its pseudo code demonstrated in Figure 3, this algorithm processes only one of the sets, the second set is pre-partitioned in the Map Phase, next, in the Reduce Phase joins the Datasets which came from the Map Phase.

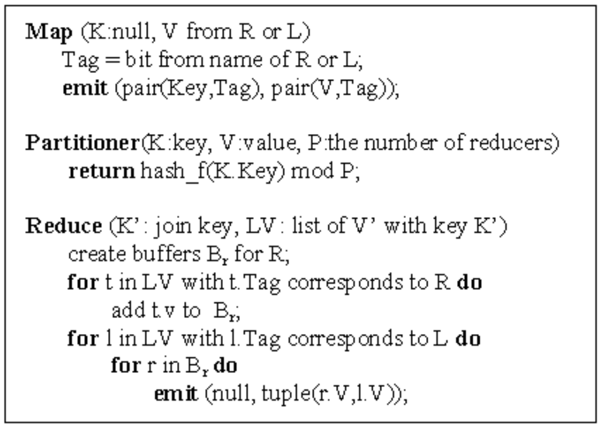
The restriction of this algorithm, is splitting one set in advanced, and similar to Reduce-Side Join it requires two phases for preprocessing, and one phase for the Join, in fact, this algorithm need no information regarding the data source since they came in the Reduce Phase.



```
Map (K: null, V from R or L)
    Tag = bit from name of R or L;
    emit (Key, pair(V,Tag));

Reduce (K': join key, LV : list of V with key K')
create buffers B_r and B_l for R and L;
    for t in LV do
        add t.v to B_r or B_l by t.Tag;
    for r in B_r do
        for l in B_l do
            emit (null, tuple(r.V,l.V));
```

Figure 1: General Reducer-Side Join (Pigul, 2012)

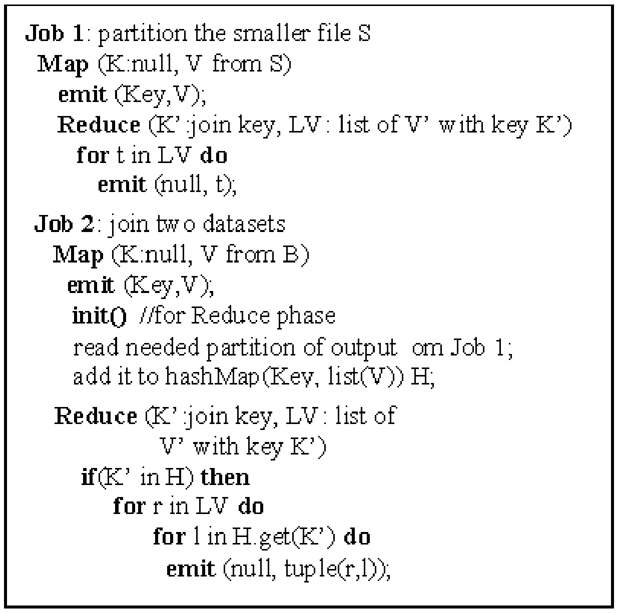


```
Map (K:null, V from R or L)
    Tag = bit from name of R or L;
    emit (pair(Key,Tag), pair(V,Tag));

Partitioner(K:key, V:value, P:the number of reducers)
    return hash_f(K.Key) mod P;

Reduce (K': join key, LV : list of V' with key K')
    create buffers B_r for R;
    for t in LV with t.Tag corresponds to R do
        add t.v to B_r;
    for l in LV with l.Tag corresponds to L do
        for r in B_r do
            emit (null, tuple(r.V,l.V));
```

Figure 2: Optimized Reducer-Side Join (Pigul, 2012)



```
Job 1: partition the smaller file S
 Map (K:null, V from S)
   emit (Key,V);
   Reduce (K' :join key, LV : list of V' with key K')
     for t in LV do
       emit (null, t);
 Job 2: join two datasets
   Map (K:null, V from B)
    emit (Key,V);
    init() //for Reduce phase
    read needed partition of output om Job 1;
    add it to hashMap(Key, list(V)) H;
   Reduce (K' :join key, LV : list of
             V' with key K')
     if(K' in H) then
        for r in LV do
            for l in H.get(K') do
              emit (null, tuple(r,l));
```

Figure 3: Hybrid Hadoop Join (Pigul, 2012)

1.1.2 Map Side Join (Pigul, 2012)

This algorithm doesn't manage the Reduce Phase, in fact, the Join job consists of two jobs, the partition Join, and in-memory Join.

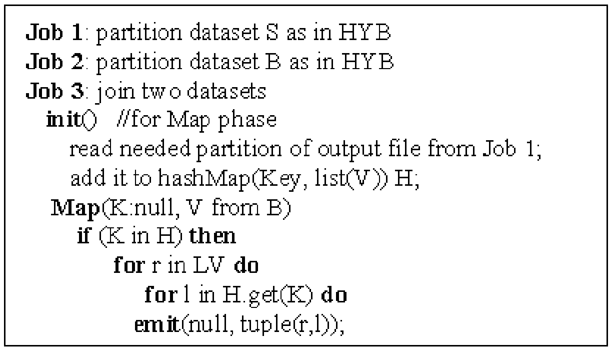
The first job is responsible for partitioning the data into the same number pieces, where the second job is responsible for sending all smaller Datasets to all Mappers, after that, partitioning the bigger Datasets to all Mappers, in fact, this type of Join has a problem when the partitioned sets remain bigger than the memory, and hence, can't fit in.

To solve this, there are three algorithms to prevent this problem:

- JDBM Map Join.
- Multi-Phase Map Join.
- Reversed Map Join.

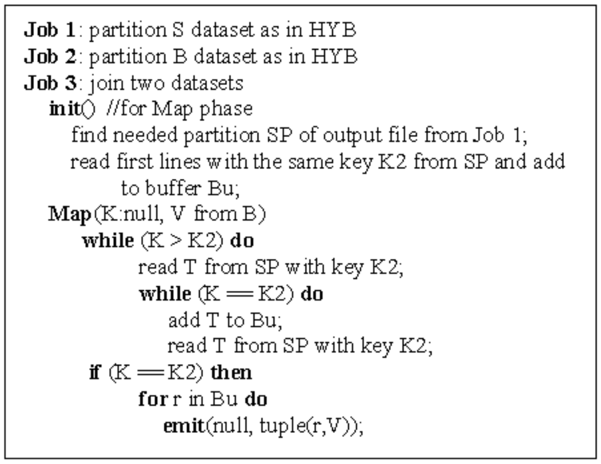
Map-Side Partition Join Algorithm (Pigul, 2012) makes an assumption that the two Datasets are pre-partitioned into the same number, its pseudo code in Figure 4.

The improvement of this algorithm is the Map-Side Partition Merge, its pseudo code in Figure 5, which apply Merge Join to the partitions using the same ordering, in addition, it reads the second set on-demand and it's not read completely, which avoids memory overflow.



```
Job 1: partition dataset S as in HYB
Job 2: partition dataset B as in HYB
Job 3: join two datasets
  init()  //for Map phase
    read needed partition of output file from Job 1;
    add it to hashMap(Key, list(V)) H;
  Map(K:null, V from B)
    if (K in H) then
       for r in LV do
          for l in H.get(K) do
          emit(null, tuple(r,l));
```

Figure 4: Map-Side Partition Join (Pigul, 2012)



```
Job 1: partition S dataset as in HYB
Job 2: partition B dataset as in HYB
Job 3: join two datasets
  init()  //for Map phase
    find needed partition SP of output file from Job 1;
    read first lines with the same key K2 from SP and add
        to buffer Bu;
  Map(K:null, V from B)
     while (K > K2) do
          read T from SP with key K2;
          while (K == K2) do
             add T to Bu;
             read T from SP with key K2;
       if (K == K2) then
          for r in Bu do
             emit(null, tuple(r,V));
```

Figure 5: Map-Side Partition Merge Join (Pigul, 2012)

1.1.3 Semi-Join (Pigul, 2012)

Based on the fact that the Join process is not using all of the Dataset tuples, hence, deleting these tuples reduces the data transfer rate over the network, also reduces the Joined Datasets size, that what semi Join dose by filtering the data.

There are three ways to implement the semi-Join (Pigul, 2012):

- Bloom Filter Semi-Join.
- Selective Semi-Join.
- Adaptive Semi-Join.

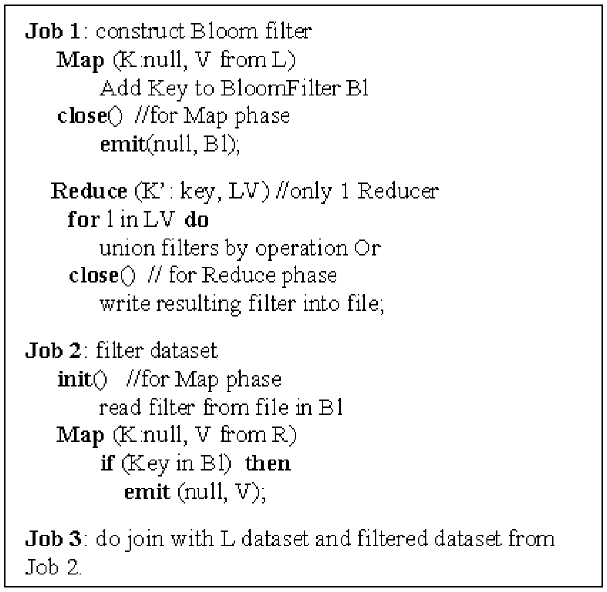
1.1.3.1 Bloom-Filter (Pigul, 2012)

A bitmap of the elements in the set, which also defined by the set, in fact, it may contain false positives but non false negatives, in addition, the bitmap has a fixed size of M.

The semi-Join has two jobs, the first one consists of the Map Phase which handles the designation and addition of keys from one set to the bloom filter, after that, the Reduce Phase combines many bloom filters from the first phase into one bloom filter.

The second job only consists of the Map Phase, in which filters the second data set with the first job Bloom-filter, as the fact that enhancing accuracy by increasing the scope of the bitmap, however, a bigger bitmaps cost additional memory.

The bloom filter pseudo code is displayed in Figure 6.



```
Job 1: construct Bloom filter
    Map (K:null, V from L)
          Add Key to BloomFilter Bl
    close() //for Map phase
          emit(null, Bl);

   Reduce (K' : key, LV) //only 1 Reducer
     for l in LV do
          union filters by operation Or
     close() // for Reduce phase
          write resulting filter into file;

Job 2: filter dataset
    init()  //for Map phase
          read filter from file in Bl
    Map (K:null, V from R)
          if (Key in Bl) then
             emit (null, V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

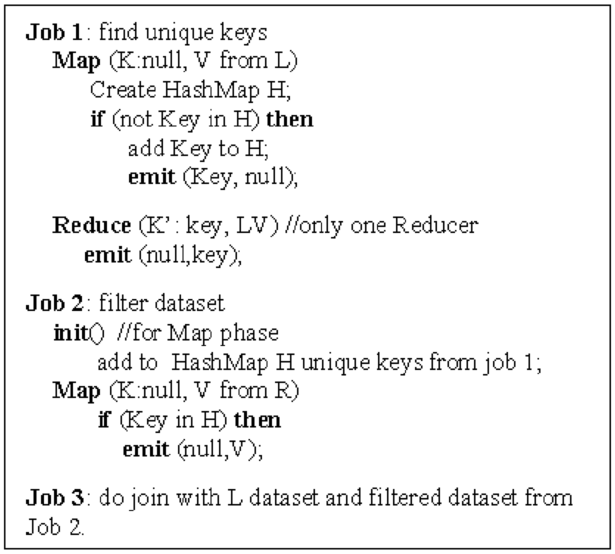
Figure 6: Bloom-Filter (Pigul, 2012)

1.1.3.2 Selective Semi-Join (Pigul, 2012)

Select unique key and build a hash table, the semi-Join performs 2 jobs the first phase in which a unique key will be selected in the Map Phase, the second job contains only Map Phase, which filters the second set, the hash table may be extremely large depending on the key size, its pseudo code shown in Figure 7.

1.1.3.3 The Adaptive Semi Join (Pigul, 2012)

Performs one job and invoking filtering in the Join similar to Reduce-Side Join Map Phase the key from two Datasets and in the Reduce Phase a key with different tags selected, the disadvantage of this way is the large Dataset information transmission across the network, it's pseudo code explained in Figure 8.



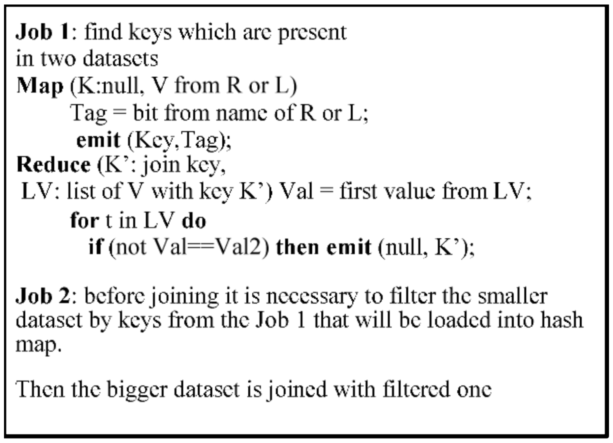
```
Job 1: find unique keys
   Map (K:null, V from L)
      Create HashMap H;
      if (not Key in H) then
         add Key to H;
         emit (Key, null);

   Reduce (K' : key, LV) //only one Reducer
      emit (null,key);

Job 2: filter dataset
   init() //for Map phase
      add to HashMap H unique keys from job 1;
   Map (K:null, V from R)
      if (Key in H) then
         emit (null,V);

Job 3: do join with L dataset and filtered dataset from
Job 2.
```

Figure 7: Simi-Join with Selection (Pigul, 2012)



```
Job 1: find keys which are present
in two datasets
Map (K:null, V from R or L)
      Tag = bit from name of R or L;
      emit (Key,Tag);
Reduce (K': join key,
 LV: list of V with key K') Val = first value from LV;
      for t in LV do
         if (not Val==Val2) then emit (null, K');

Job 2: before joining it is necessary to filter the smaller
dataset by keys from the Job 1 that will be loaded into hash
map.

Then the bigger dataset is joined with filtered one
```

Figure 8: The Adaptive Semi Join (Pigul, 2012)

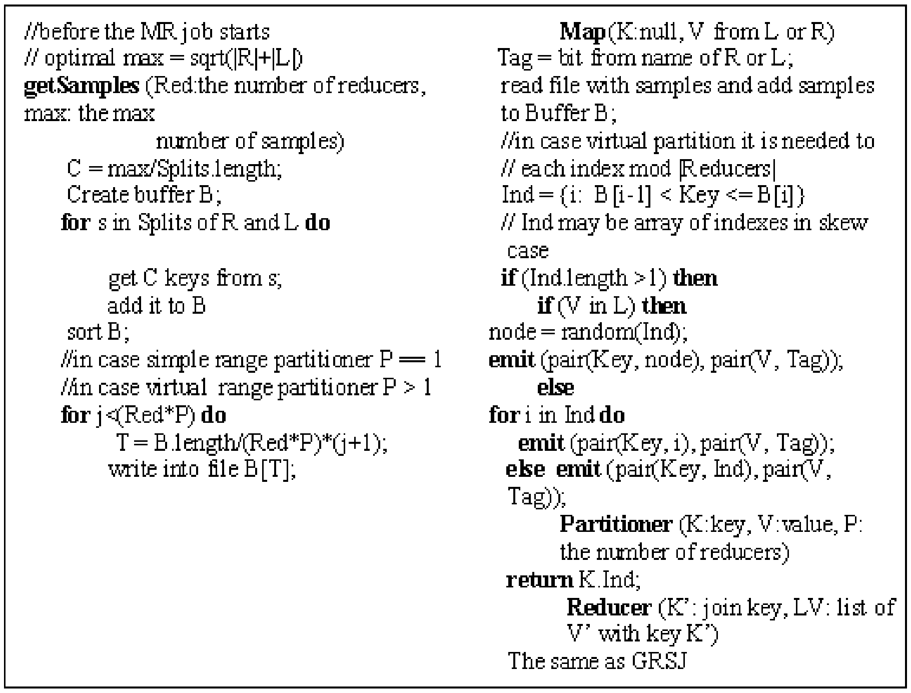
1.1.4 Range Practitioners (Pigul, 2012)

All the algorithms except in-memory Join have issues with the data skew, the methods of the default hash Partitioner replacement are:

- Simple Range-Based Partitioner.
- Virtual Processor Partitioner.

Simple Range-Based Partitioner: builds the vector from the original set of keys, and set of numbers randomly chosen, later, a node is selected arbitrarily in the situation of data on which to construct a hash table, else, the key will send to all nodes.

Virtual Processor Partitioner (Pigul, 2012): based on increasing the number of partitions. The number of divisions specifies multiple task numbers, after that, the same keys are scattered through extra nodes than in the previous algorithm, Figure 9 shows the pseudo code for it.



```
//before the MR job starts
// optimal max = sqrt(|R|+|L|)
getSamples (Red:the number of reducers, max: the max
                number of samples)
    C = max/Splits.length;
    Create buffer B;
    for s in Splits of R and L do

        get C keys from s;
        add it to B
    sort B;
    //in case simple range partitioner P == 1
    //in case virtual range partitioner P > 1
    for j<(Red*P) do
        T = B.length/(Red*P)*(j+1);
        write into file B[T];

Map(K:null, V from L or R)
Tag = bit from name of R or L;
read file with samples and add samples
to Buffer B;
//in case virtual partition it is needed to
// each index mod |Reducers|
Ind = {i: B[i-1] < Key <= B[i]}
// Ind may be array of indexes in skew
 case
if (Ind.length >1) then
    if (V in L) then
node = random(Ind);
emit (pair(Key, node), pair(V, Tag));
    else
for i in Ind do
    emit (pair(Key, i), pair(V, Tag));
  else emit (pair(Key, Ind), pair(V,
  Tag));
Partitioner (K:key, V:value, P:
the number of reducers)
  return K.Ind;
Reducer (K': join key, LV: list of
V' with key K')
  The same as GRSJ
```

Figure 9: The Range Partitioners. (Pigul, 2012)

2. RELATED WORK

A. Pigul (Pigul, 2012) did a great comparative study on parallel processing Join algorithms and their enhancements, the comparison is shown in Figure 10 and Figure 11 shows the execution time comparison between the experimented algorithms in the study.

The Authors of this article choose the work of A. Pigul (Pigul, 2012)as basis for selecting and using the algorithms.

M. H. Khafagy (Khafagy, 2015) applied an index Join algorithm with a various size large Datasets (up to 500 million records) that shows a better performance regardless of the size of the Datasets, without mentioning the technique operated in creating the index and applies only experimental results to prove the performance of the proposed algorithm.

Wen L. et.al. (Liu, Shen, & Wang, 2016) introduced a similarity Join in MapReduce with the help of a proposed similarity Join tree (SJT) as an index, SJT partitions the data file on the underlying data distribution, and put similar records to the same partitions.

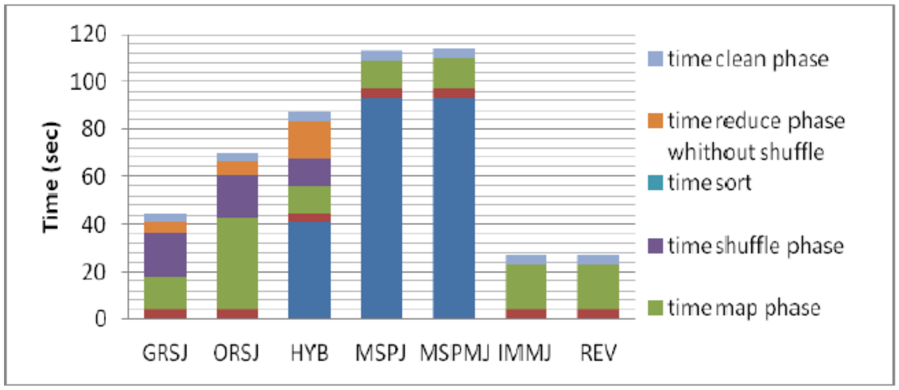


Figure 10: Phases executions time of various algorithms. Size 10^4*10^5 (Pigul, 2012)

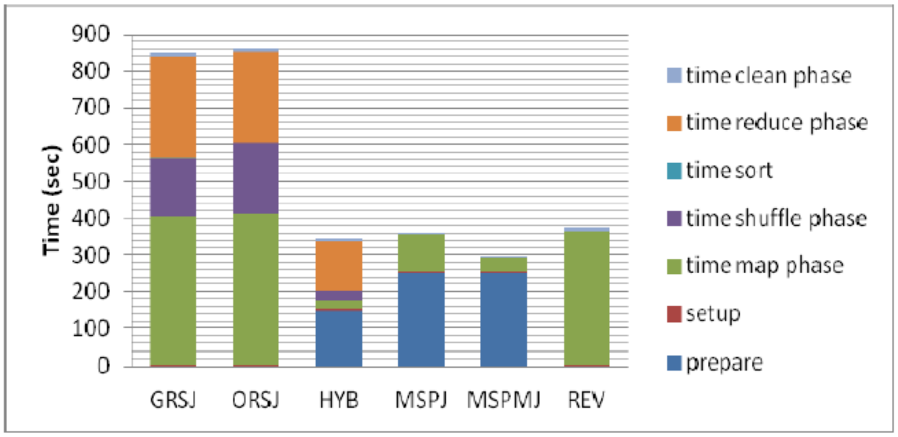


Figure 11: Phases executions time of various algorithms. Size 10^6*10^6 (Pigul, 2012)

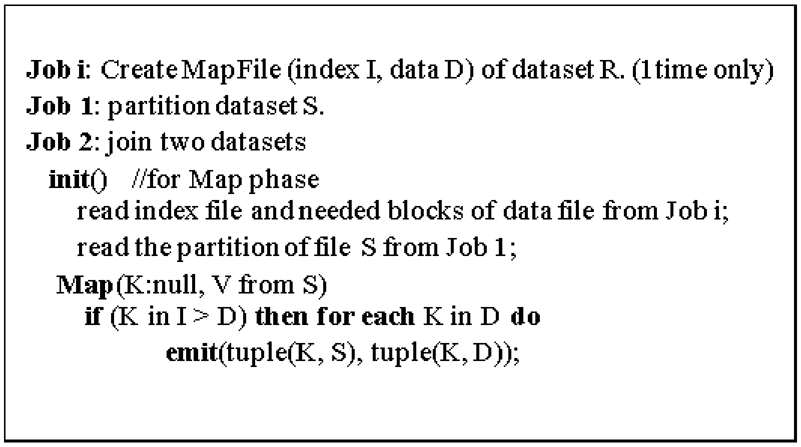
3. Proposed Method

The proposed idea in this paper is to integrate data file indexing using Hadoop HBase MapFiles with map-side join algorithm, providing both mathematical and experimental results.

MapFile Index-Based Join Algorithm (MFIJ).

The main idea of using Hadoop MapFile is not only to reduce the HDFS access operations, as the both index and data file blocks is transferred into memory, depending on requested data for each map node, leading to a faster memory side processing.

The transferred blocks of the data file are CPU-Memory accessed to invoke key search, records join, and then write them into output file in HDFS storage, as described in the MFIJ pseudo code in Figure 12, in addition, the creation of MapFile could be for one, multiple, or all datasets used in the join operation depending on the needed query.



```
Job i: CreateMapFile (index I, data D) of dataset R. (1time only)
Job 1: partition dataset S.
Job 2: join two datasets
  init()   //for Map phase
    read index file and needed blocks of data file from Job i;
    read the partition of file S from Job 1;
   Map(K:null, V from S)
     if (K in I > D) then for each K in D do
          emit(tuple(K, S), tuple(K, D));
```

Figure 12: MapFile Index-Based Join.

3.1 Mathematical Comparison

Hadoop MapReduce splits the work load on multiple nodes (N) in which the overall complexity of accessing a big data file will be improved, assuming that using two files (FS, FR) in the join operation, each with (S, R) records respectively, and (S<R). The Reduce-Side Join (RSJ) cost will be:

$$RSJ_{cost} = \left((N+1) \times \frac{S}{N} \times \frac{R}{2} \times \left(\frac{log_2 R}{2} \right) \right)$$

Where, (S/N) is the amount of records from file (S) in each map task, (R/2) is the average number of joined record instances from file (R) for each record in file (S), (1) is a representation of the parallel records search and read operations in the map phase while, (N) is the number of join-write operations in the reduce phase, finally, the ((log2 R)/2) is average binary search cost for a record in file (R). The Map-Side Join (MSJ) cost is as follows:

$$MSJ_{cost} = \left(\frac{S}{N} \times \frac{R}{2} \times \left(\frac{log_2 R}{2} \right) \right) + \left(\frac{S}{2} \right)$$

Where, (S/2) is the average write operations in the reduce phase, the deferent issue here is that (N) join operations in the reduce phase are replaced by (S/2) normal, average write operations, as mapping takes the responsibility of doing all the read and join operations.

In the proposed MapFile Index-Based Join (MFIJ) the (R) file conversion to a MapFile with a data file of (D=R) records, and an index file of (I) records, each addressing a block of the data file containing (D/I) records. The MapFile Index-Based Join cost will be:

$$MFIJ_{cost} = \left(\frac{S}{N} \times \frac{D}{2} \times \left(\frac{log_2 I}{2} + \frac{log_2 {}^{D}/_{I}}{2}\right)\right) + \left(\frac{S}{2}\right) + \left(\frac{ND}{2}\right) + NI$$

Where, (ND/2, NI) are the average read cost of both data and index files respectively into the memory.

For the first look into MFIJ cost, it appears that it could be more than both RSJ and MSJ costs, in fact, it is less if you know that both index file and data file blocks are cached into memory, reducing a large amount of HDFS data storage access operations, and hence, faster join operations, with more memory processing operations.

3.2 Experiment and Results

The environment used for the experiment is Apache™ Hadoop® Version 2.5.2 (Apache™ Hadoop®, n.d.), on Cloudera© QuickStart® CDH Version 5.5 (Cloudera.com, 2016), a single node with 1 CPU, 100% execution capacity, and 4GB RAM, over Oracle© VM VirtualBox® Version 5.0.16 platform (Oracle© VM VirtualBox® V 5.0.16," , 2016), installed over a Microsoft© Windows® 10 Pro 64-bit (10.0, Build 10586) (Microsoft Windows 10 Pro, 2015) with Intel® Core™ 2 Duo CPU P8400 2.26 GHz, and 8GB RAM memory.

The code used is inspired by work of S. Asad Articles in the CodeProject© Website (Asad, 2015) (Asad S. , 2015), which address the same proposed methodology, using Adventure Works 2012 OLTP Script Dataset from Microsoft© SQL Server® Database Product Samples (Adventure Works for SQL Server 2012, 2012).

The experiment is based on comparing the performance of three join programs based on RSJ, MSJ, and MFIJ join algorithms, over the Hadoop MapReduce environment. The creation of the MapFile to be used in MFIJ program is done for only one time by a forth separate program which creates the MapFile of the second dataset file and store it in the HDFS storage, however, the cost of the MapFile Creation is also calculated in the results.

Each join program is executed 30 times to ensure the average value of performance which leads to best comparison, the metrics collected are the CPU execution time, and the run time for each 30 run instances, in each program.

The CPU execution time result that MFIJ algorithm has the most CPU usage which is approximately 18%, and 19% more than RSJ, and MSJ respectively, as a result of the higher number of CPU-to-memory operations, while, RSJ, and MSJ has less CPU time usage, as illustrated in Figure 13. The result of average run time shown in Figure 14, prove that MFIJ has better performance than RSJ, and MSJ, by approximately 38%, and 34% less average run time than RSJ, and MSJ algorithms respectively

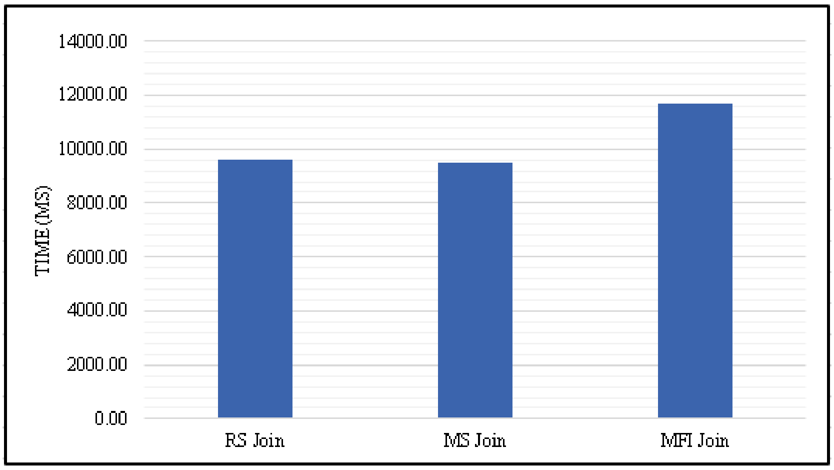


Figure 13: Average CPU Time (ms).

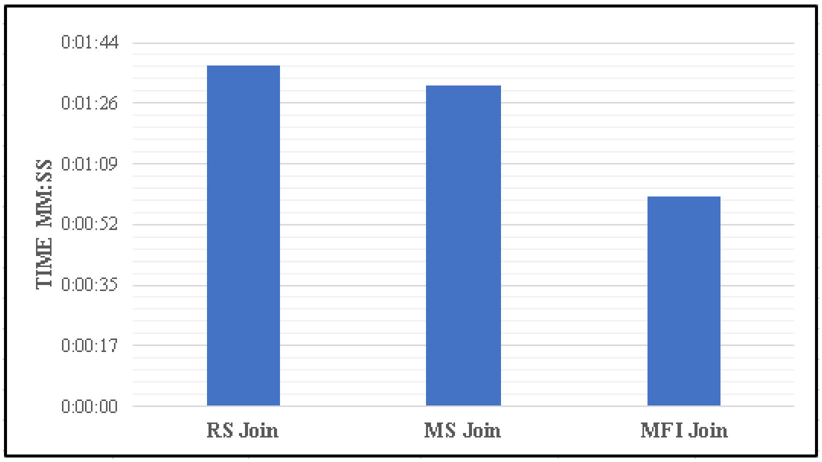


Figure 14: Average Run Time (Min:Sec).

4. Conclusion and Future Work

The real performance improvement in this algorithm is using more CPU-Memory operations to accelerate the overall join operations. As proved by the experiment results displayed in this paper, the using of Hadoop MapFiles improved the join in Hadoop MapReduce environment, the authors of this paper encourage the usage of MapFiles as actual datasets rather than a transformation of dataset, for both static and live-stream big data datasets. For live-stream datasets, an automated MapFile storage tool could be useful to collect live-stream data and directly store it as MapFiles inside the Hadoop HDFS storage system.

The authors of this paper are planning to continue the investigation of the effect on performance by implementing MapFiles in one or more different MapReduce based join algorithms like Self-Join and Multi-Join Algorithms.

REFERENCES

Adventure Works for SQL Server 2012. (2012, 3 12). Retrieved from http://msftdbprodsamples.codeplex.com/releases/view/55330

Apache™ Hadoop®. (n.d.). Retrieved from http://hadoop.apache.org/

Asad, S. (2015, Jan 29). Implementing Joins in Hadoop Map-Reduce. Retrieved from http://www.codeproject.com/Articles/869383/Implementing-Join-in-Hadoop-Map-Reduce

Asad, S. (2015, 3 16). Implementing Joins in Hadoop Map-Reduce using MapFiles. Retrieved from http://www.codeproject.com/Articles/887028/Implementing-Joins-in-Hadoop-Map-Reduce-using-MapF

Class MapFile. (n.d.). Retrieved from https://hadoop.apache.org/docs/r2.6.2/api/org/apache/hadoop/io/MapFile.html

Cloudera.com. (2016). Retrieved from QuickStart Downloads for CDH 5.5: http://www.cloudera.com/downloads/quickstart_vms/5-5.html

Hadoop I/O: Sequence, Map, Set, Array, BloomMap Files. (2011). Retrieved from http://blog.cloudera.com/blog/2011/01/hadoop-io-sequence-map-set-array-bloommap-files/

Khafagy, M. H. (2015). Indexed Map-Reduce Join Algorithm. International Journal of Scientific & Engineering Research, 6(5), 705-711.

Khetrapal, A., & Ganesh, V. (2006). HBase and Hypertable for large scale distributed storage systems. Dept. of Computer Science, Purdue University, 22-28.

Liu, W., Shen, Y., & Wang, P. (2016). An efficient MapReduce algorithm for similarity join in metric spaces. The Journal of Supercomputing, 72(3), 1179-1200.

Microsoft Windows 10 Pro. (2015). Retrieved from https://www.microsoft.com/en-us/windows/features.

Oracle© VM VirtualBox® V 5.0.16,”. (2016, 3 4). Retrieved from Oracle: https://www.virtualbox.org

Pigul, A. (2012). Comparative Study Parallel Join Algorithms for MapReduce environment. Труды Института системного программирования РАН, 23.

Prasad, B. R., & Agarwal, S. (2016). Comparative Study of Big Data Computing and Storage Tools.

Zhang, C., Li, J., & Wu, L. (2013). Optimizing Theta-Joins in a MapReduce Environment. International Journal of Database Theory and Application, 6(4), 91-107.