Received 28 June 2018
Revised 2 October 2018
Accepted 14 November 2018

Handling data-skewness in character based string similarity join using Hadoop

Kanak Meena and Devendra K. Tayal
Department of CSE, Indira Gandhi Delhi Technical University for Women, Delhi, India
Oscar Castillo
Division of Graduate Studies and Research, Tijuana Institute of Technology, Tijuana, Mexico, and
Amita Jain
Department of CSE, Ambedkar Institute of Advanced Communication Technologies and Research, Delhi, India

Abstract

The scalability of similarity joins is threatened by the unexpected data characteristic of data skewness. This is a pervasive problem in scientific data. Due to skewness, the uneven distribution of attributes occurs, and it can cause a severe load imbalance problem. When database join operations are applied to these datasets, skewness occurs exponentially. All the algorithms developed to date for the implementation of database joins are highly skew sensitive. This paper presents a new approach for handling data-skewness in a character- based string similarity join using the MapReduce framework. In the literature, no such work exists to handle data skewness in character-based string similarity join, although work for set based string similarity joins exists. Proposed work has been divided into three stages, and every stage is further divided into mapper and reducer phases, which are dedicated to a specific task. The first stage is dedicated to finding the length of strings from a dataset. For valid candidate pair generation, MR-Pass Join framework has been suggested in the second stage. MRFA concepts are incorporated for string similarity join, which is named as "MRFA-SSJ" (MapReduce Frequency Adaptive – String Similarity Join) in the third stage which is further divided into four MapReduce phases. Hence, MRFA-SSJ has been proposed to handle skewness in the string similarity join. The experiments have been implemented on three different datasets namely: DBLP, Query log and a real dataset of IP addresses & Cookies by deploying Hadoop framework. The proposed algorithm has been compared with three known algorithms and it has been noticed that all these algorithms fail when data is highly skewed, whereas our proposed method handles highly skewed data without any problem. A set-up of the 15-node cluster has been used in this experiment, and we are following the Zipf distribution law for the analysis of skewness factor. Also,

Applied Computing and Informatics
Vol. 18 No. 1/2, 2022
pp. 22-44
Emerald Publishing Limited
e-ISSN: 2210-8327
p-ISSN: 2634-1964
DOI 10.1016/j.aci.2018.11.001

a comparison among existing and proposed techniques has been shown. Existing techniques survived till Zipf factor 0.5 whereas the proposed algorithm survives up to Zipf factor 1. Hence the proposed algorithm is skew insensitive and ensures scalability with a reasonable query processing time for string similarity database join. It also ensures the even distribution of attributes.

Keywords Similarity join, Big data, Hadoop, MapReduce, Data skewness

Paper type Original Article

1. Introduction

The term "Big Data" [1–10] has turned into a buzzword and is broadly used in both research and industrial world. Big Data refers to a term which is a blend of the 4V's, namely Volume, Variety, Veracity, and Velocity. But now it is evolving with time. It is closely being related to data integration, which aims at combining the various forms of data from different sources and provides a consolidated view. Data integration [3,11] can be achieved by using the String similarity join [2,3], which provides a similar pair of strings from the two-given collection of strings. The similarity of the two strings can be calculated by using their similarity functions. There are two types of similarity functions which are used to calculate the similarity viz.: character-based similarity functions [12–23] and set-based similarity functions [14–18].

1. Character-Based Similarity Functions: These are the functions which are based on the number of character operations performed, to transform one string into another string. These are calculated by the Edit Distance (ED) [12–16], which can be referred to as the minimum number of operations that are needed to transform one string into another. Operations such as insertion and deletion are performed under the edit distance. For example: Given r="Sanskrit" and s="Sanskirt". We have ED(r,s)=2.
2. Set-Based Similarity Functions: There exist well-known methods of the set-based similarity function, namely: Dice Similarity [18,19], Jaccard Similarity [18–23] and Cosine Similarity [15–18]. These similarity measures are given by the following equations:

- $\mathrm{Dice}(r,s) = (2|r \cap s|/|r| + |s|)$

where Dice(r,s) refers to the Dice similarity that exists between string r and s.

- $\mathrm{Jac}(r,s) = (|r \cap s|/|r \cup s|)$

where Jac(r,s) refers to the Jaccard similarity that exists between string r and s.

- $\mathrm{Cos}(r,s) = (|r \cap s|/\sqrt{|r|.|s|})$

where Cos(r,s) refers to the Cosine similarity that exists between string r and s.

For example, given r="I am cool person" and s="I am cool". We have $|r| = 4$ and $|s| = 3$, further $|r \cup s| = 4$ and $|r \cap s| = 3$. Thus, $\mathrm{DICE}(r,s) = (6/4)$, $\mathrm{JAC}(r,s) = (3/4)$ and $\mathrm{COS}(r,s) = (3/\sqrt{12})$.

String similarity joins have many real-world applications, e.g., entity resolution, copy detection, document clustering [19], plagiarism detection and data integration [23]. Traditional algorithms of string similarity joins have memory constraints and hence they have limited applications when they deal with a large amount of data.

Similarity join is the critical issue of discovering all sets of records from a given set that have similarity scores more noteworthy than a predefined Similarity limit under a given similitude work. It can be applied in various applications where it needs to deal with the progressively immense amount of information; the issue of scaling up the similarity joins is always getting more imperative. Performing Similarity joins on the enormous amount of data presents two principal difficulties. First, the information can never again fit into the memory of one machine, which calls for workload partitioning. Given, the pairwise-comparison nature

of the issue, dividing data to guarantee load balancing, while at the same time limiting correspondence cost and redundancy is troublesome. The trouble of load adjusting is additionally aggravated by the need to deal with various informational indexes with skewed distributions and high dimensionality. Second, since the number of examinations required develops quadratically as data increases, methods that require looking at all sets of records don't scale well. Along these lines, another test is to filter the candidate pair without calculating the actual similarity. Designing filters to help a vast class of valuable similarity function is of incredible and practical significance [23].

Data Skewness [23,24–31] occurs when the computational load is unbalanced among map tasks or among reducing tasks. Skewness can lead to significantly longer job execution time, lower cluster throughput, high computation time and high computation cost. Performance of existing algorithms degrades or not up to the mark due to the presence of data skewness. It is essential to define the method and overcome this problem for better results. Existing data techniques causes data skewness during the partitioning which creates the imbalance of data and data duplication. As well they require a high cost of computation for handling the vast amount of data. In this paper, character-based similarity functions are being used to make a scalable approach to process big data.

The MapReduce [5] framework proposed by Google is utilized to perform substantial scale information in a distributed manner. Therefore, there has been valuable research on analytical join query handling in MapReduce for extensive datasets. In any case, few applications need to deal with an immense amount of data, and there are three fundamental difficulties to be intended which occurs usually.

First, because the size of the data collection is enormous, the data must be partitioned and prepared appropriately. Thus, workload-aware information dividing procedures are required. These guarantees not only balance the data as well as an output of each machine. Second, a Complex filtering strategy is needed since the quantity of correlations develops significantly as the dataset size increase. Third, a principle issue occurs when handling a join query on MapReduce over the multiple datasets.

This paper presents a hybrid approach by using Pass Join (pl. refers Section 2 of this paper), the map-reduce framework with the concept of the Map-Reduce Frequency Adaptive (MRFA) (pl. refers Section 2), to handle the basic record join. This approach has been designed to Handle Data-Skewness in the Character Based String Similarity Join.

As explained, pass join itself is a robust algorithm which states "proposed a high-quality partition-based signature scheme for the edit distance function" [14–19], it has been proven to be an effective and efficient method when it comes to generating the various candidate pairs in the concerned dataset. This paper presents a MapReduce framework of pass join using MRFA [24] concepts. For the string, the similarity joins named as MRFA-SSJ. Hence, the proposed work has the combined advantages of pass join [14] and MRFA [24].

The proposed technique can be used in various felid to have better result estimation and calculation. The proposed technique helps in better decision making as it evenly distributes the data. An area in which it can be applied viz. optimization algorithms, IOT of an educational environment, quadratic assignment problem, decision-based methods, neutrosophic problems, etc. [32–40]

The rest of the paper is organized as follows: Section 2 describes the preliminaries which are related to the proposed work. Section 3 is devoted to the literature review of the existing techniques. Formulation of the proposed work has been discussed in Section 4. Experimental results are described in Section 5 and the conclusion has been discussed in Section 6.

2. Material and methods

This section briefly describes the important concepts which are repeatedly used in this paper. The proposed work strictly focuses on the data skewness effects in join operations [19].

For implementing the same, one must be familiar with the concepts of Hadoop and MapReduce. All these concepts are briefly described as follows:

a) Pass Join [14]: Pass join is the method which efficiently and adaptively deals in short and long strings simultaneously by employing its partition-based methods. This approach is used to generate the candidate pairs by using inverted indices and is a novel pruning technique to verify the candidate pairs.

b) Partition Scheme [14,18–19]: Partition scheme is used for joining key of records R belonging to relation S, followed by the even partitioning scheme and it is partitioned into T+1 segments. Let s_len be the length of the join key. Let segment be ask= s_len−⌊s_len/(T + 1)⌋ * |−(T + 1))⌊−|. Thus the length of the first T + 1−k segments will be ⌊(s_len/(T + 1)|−)⌋, −|while the length of the last k segments will be⌈s_len/|−(T + 1)⌉ −|. Also, Lil (inverted index) is used as a set of ith segments of length l.

c) Substring Selection and Candidate Pair Generation [14,18–19]: Substring selection and candidate pair generation are important steps in pass join. For each segment belonging to Lil, choose substrings of the join key of the record of 'R' according to the multi-match aware method as described in "Pass Join". The minimal start position for the substring is max (1, pi−(i−1)) while the maximal start position is min (s_len − li + 1, pi + (i − 1)), where pi is the start position of the segment, i stands for the ith segment, s_len is the length of the original join key which has been partitioned into segments and li is the length of the segment. Further, performing the enhancement of the time complexity which generates even fewer substrings, changing the minimal and maximal start positions.

$$\text{Minimal start position} = \max\ (\text{pi} - -(\text{i} - 1),\ \text{pi} - -\text{Ur} - \text{s}) \tag{1}$$

$$\text{Maximal start position} = \min\ (\text{pi} + \text{s}_{\text{len}} - -\text{li} + (\text{U} + 1 - \text{i}),\ \text{pi} + \text{Us} - \text{r}) \tag{2}$$

where, Ur − s = li − s _ len + (s _ len − r _ len - i), Us − r = i − l and U = Ur − s + Us − r

d) Going this way, the numbers of substrings generated are reduced from O (T3) to O (T2) [19], where l is the length of the entire string. After generating the substrings, there is a need to check that the segment matches any one of the substrings or not. If it does, then it implies that the pair of records from relation R and S are candidate pairs. Using candidate pairs, there is a possibility that their edit distance satisfies the threshold value for edit distance. This method helps to avoid calculating the edit distance of all pairs from relations R and S, as computing edit distance is a time-consuming process.

e) Verification of the Candidate Pairs [14,19]: After generating the candidate pairs there needs to verify whether these pairs satisfy the edit distance threshold or not. For this, the edit distance threshold of the verification algorithm is used as described in Pass Join. Let the join keys of the candidate pair be r and s & Let r_len and s_len be the lengths of the join keys from relations R and S respectively [14]. Let length difference be D = s _ len − r _ len.

f) Let us use r_len × s_len matrix M. Using dynamic programming to compute the edit distance, the entries of M can be defined recursively [14] as:

$$\text{M}[0, \text{j}] = \text{j}, 0 < \ = \text{j} < \ = \text{s}_{\text{len}}$$

$$\text{M}[\text{i}, \text{j}] = \min(\text{M}[\text{i} - 1, \text{j}] + 1,\ \text{M}[\text{i}, \text{j} - 1] + 1,\ \text{M}[\text{i} - 1, \text{j} - 1] + \text{d} \tag{3}$$

d = 0, if the ith character of r=jth character of s = 0, else 1.

g) The length-aware verification only compute values of M[i,j] such that $i - \lfloor |T - D| - /2 \rfloor - | <= j <= i + \lfloor |T - D| - /2 \rfloor - |$. To terminate the process early, the expected edit distance is computed for every row, as $E(i,j) = M[i,j] + (s\_len - j) - (r\_len - i)$. This gives a lower bound on the value of M[i,j]. Thus if for some i and all j, E[i,j] is greater than T, then further terminate the verification process and reject the candidate pair [14].

h) To further improve the process, extension-based verification [14] and sharing-computation [14] algorithms are being used. The extension-based verification provides tighter edit distance thresholds. Partition the s and r into three parts: sl, sm, sr and rl, rm, rr respectively. sm and rm correspond to the portions of s and r that match, sl and rl are the portions to the left of sm and rm respectively. Similarly, sr and rr are portions to the right of sm and rm respectively. Let EDl be the edit distance of sl and rl and EDr be the edit distance between sr and rr. The ith segment of s matched with a substring of r. Thus, for EDl the threshold is chosen as $Tl = i - 1$, whereas the threshold for EDr is $(Tr = T + 1 - i)$. Firstly, compute the EDl, if $EDI > Tl$ then, simply reject the candidate pair, else compute the EDr and check whether $EDr <= Tr$.

i) Repeating computations can be avoided by sharing-computation [14]. Assume that, s got compared to ri. Further, it compares, s to $ri + 1$. The longest common prefix between $ri + 1$ and ri is found and assume its length be c. Values of M[i,j] for $1 <= i <= c$ and $1 <= j <= s\_len$ are copied from the previous matrix to the new matrix. Thus, individually avoid computing these values again for the next verification.

j) MRFA [24]: This is an approach which is used to handle the data skewness in join operations. It is a skew insensitive join algorithm, based on a distributed histogram. It depends on randomized key redistribution and deals in hash-based joins.

k) Hadoop [1,25–28]: It is an open source tool to handle big data. It provides effective results for the large dataset and handles the real data which is motivated by Google's MapReduce and Google File System [16]. Hadoop Distributed File System (HDFS) has been used to create the multiple replicas of data blocks.

l) Similarity Join [1–3,12–24,41–47]: It evaluates whether two strings are similar or not. If the similarity function exceeds a given threshold, the two strings are said to be similar. Basically, the similarity function is broadly divided into three categories: token-based similarity, character-based similarity and hybrid similarity and this paper are using the character-based similarity function.

m) String Similarity [12–41]: Two strings are said to be similar if the edit distance between them satisfies the threshold i.e. $ED(r, s) <= T$, where r and s are strings from the relations R and S respectively. For example, r=bscde and s=abscde and T=2. ED ("bscde", "abscde")=1 since the r can be transferred to s by inserting a character "a". ED must be smaller than the given threshold for similar strings.

3. Literature review

This section presents the literature review of existing techniques, which are related to the proposed work. Two principal methods for addressing the problem of edit distance similarity join are the TRIE-Join [12] and Ed-join [13]. Ed-join employs a filter and refines framework. In filtering, the signature for each of the strings is generated. These signatures are used to create candidate pairs. Refining involves verifying the candidate pairs. TRIE-Join [12] uses a TRIE-based framework on the other hand. A TRIE structure is used to share prefixes.

Prefix pruning is also used to improve the performance. The drawback of Ed-join is that its performance is not good for short strings. The signatures generated are not good enough and thus many candidate pairs are generated, elongating the time required for the verification or refining step. The drawback of the TRIE-Join is that it is not efficient for long strings, as long strings do not have many shared prefixes. Pass Join [14] overcomes the problem we have to process both long and short strings by employing its partition-based method. In this paper, Pass Join has been adapted for the MapReduce framework. This method will support not only long but also short strings, in big data. Q-Chunk [15] is an asymmetric signature scheme. This method is based on the q-grams and q-chunks methods. The proposed two algorithms are IndexChunk and IndexGram. Both algorithms achieve the minimum number of signature as +1. Adapt Join [16] is based on the prefix filtering based framework. It selects an object whose prefixes have no overlap. This method supports sim search queries. V-Chunk [17] is a novel technique which deals in non-overlapping substrings. This approach is based on the concept of a chunk boundary dictionary. It has been designed by integrating existing filtering techniques and also a new greedy algorithm which automatically selects the good chunks from the given dataset. Mass join [19] supports both characters and set based similarity function. Limitation of both Pass join, and Mass join is that they have been applied only in self-join operations [14,18,19,43]. Pass join is self-sufficient to handle the edit distance [18,19].

Skewreduce [28,29] handles the skewness in the MapReduce framework in a passive mode and puts an extra burden on the user who must provide cost functions. It cannot handle any unexpected conditions at runtime. To overcome the drawbacks of this algorithm SkewTune has been proposed. SkewTune [28] is designed for MapReduce-type systems, where each operator reads data from disk and writes data to disk and is thus decoupled from other operators in the data flow. Share skew [30] has been proposed to handle the skewness specifically for the multi-way join. It finds the join attribute values that frequently appear (heavy hitters). For random data, high communication cost is there. MG-Join [18] is multiple global ordering methods based on set-based similarity functions and applied explicitly to multiple joins.

MRSim Join [43] has been proposed to handle the similarity join problems. It can be used with any dataset which lies in metrics space. It has excellent execution and a scalability property which uses MapReduce programming model and based on iteratively partition scheme but inefficient to handle the data skewness. V-SMART Join [2] is a MapReduce framework to handle all pair of entities, and it is a two-stage algorithm to analyze the exact similarity for all candidate pairs. Cluster Join [23] method achieves the high probability of handling the load balancing (data skewness) by using sampling. They have introduced the novel filtering method based on bisector specific to the distance functions including Euclidean and Hamming distance; hence it cannot handle the edit distance. Map-Reduce Distance based Self Join (MR-DSJ) [47] grid partitioning based self-join algorithm gives results without duplication. Due to data skewness, the runtime is high and enables handling multiple iterations. Algorithm for load balancing [5] addresses the two factors: an amount of message transmission and data skewness by extending the estimation-based algorithm. It enables to reduce the network traffic and cost time.

Scalable Load Balancing for MapReduce-based Record Linkage [27] is another algorithm for load balancing, devoted explicitly to the sketch-based approximate data profiles. Dynamic Resource Allocation for MapReduce with Data Skew (DREAMS) [10] is a data assigning scheme, based on runtime partitioning skew mitigation. This scheme is achieved by adjusting the task runtime resource allocation. Locality-Aware and Fairness-Aware Key Partitioning (LEEN) [48] algorithm handle the partitioning skew in network bandwidth dissipation during the shuffling phase. Fast and Scalable MapReduce Similarity Join (FACET) [46] has been developed for self- join case only and solve the vector join problems on large datasets. It is unable to handle the data skewness. MapReduce Frequency Adaptive (MRFA) [24] is a skew insensitive join algorithm which is widely used in different algorithms.

Hierarchical segment tree index (HS-Tree) [47] has been developed to handle two variants such as threshold-based string similarity search and top-k string similarity search. But it does not handle the data skewness problem.

MapReduce Frequency Adaptive Group by-Join (MRFAG) [49] was proposed as a solution to handle the skewness problem by using group by joins using MapReduce framework based on distributed histograms and a critical randomized redistribution approach. It is an extended version of the MRFA algorithm. Efficient parallel set similarity to join in MapReduce [25] method is divided into three stages and is a dynamic approach to reduce data duplication. It is incapable of handling data skewness problem in basic record join operations and has lower pruning power due to the generation of many false positives.

To improve on state of the art, a hybrid approach to handle the data skewness in character-based string similarity join has been proposed in this paper. This approach has been divided into three stages. The first stage is dedicated to preprocessing and calculates the global minimum and maximum length of the strings. The second stage is committed to the MapReduce Framework of pass join, and the last stage contains an algorithm MRFA, which is designed for string similarity join namely MRFA-SSJ.

4. Proposed approach

This section presents the proposed approach, and it is dealing with basic record join (BRJ) method. The reason for the poor performance of BRJ is due to the skewness, and it affects the workload balance when it deals with a large amount of data (as described in Efficient Parallel Set-Similarity Join) [25]. A MapReduce framework is proposed based on Pass Join name as "MR-Pass Join" following the MRFA-SSJ algorithm to handle the skewness effect in a join operation. MR-Pass Join gives the list of pairs of records which satisfy the edit distance threshold. BRJ has been used to calculate the entire records of the pairs along with the edit distance value. However, the performance of BRJ is affected due to skewness in the data. The problem overcome by the MRFA algorithm, it is adapting to the BRJ framework by incorporating the similarity join concept. Further, the proposed work is specified in the following three stages along with algorithms and examples.

Let's take an example referred in Table 1. The table contains records from relations R and S. The tag of a record has the following values:

1. R_tag=This indicates that the record belongs to relation R
2. S_tag=This indicates that the record belongs to relation S
3. Pair_tag=This indicates that the record defines an RID pair (which will be computed in stage 2 and is used in stage 3).

Tag	RID	Key (the 1st string of the entire record – used for joining)	Value (the rest of the record)
R_tag	1	A	& a mobility sales & service
R_tag	2	Aa	20meetings 20in 20charlotte
R_tag	3	Aaa	and grand canyon discounts
R_tag	4	aaacookcountyconsolidation.com	Null
R_tag	5	Aaacoman	Null
S_tag	1	a & a mobility sales & service	& a mobility sales & service
S_tag	2	aa 20meetings 20in 20charlotte	20meetings 20in 20charlotte
S_tag	3	aaa and grand canyon discounts	and grand canyon discounts
S_tag	4	aaacookcountyconsolidation.com	Null

Table 1.
Example of Sample Input.

4.1 Stage 1

Stage 1 computes the distribution of data from the relations R and S. While the records from R can merely be hashed to the reducers in the next stage, a record from S needs to go to all the reducers containing those records of R with which it can possibly be joined. This stage outputs a partition list, which is explained in the reducer phase. The input for this stage will only have records from the relation R. The working of the mapper and reducer are described in Sections 4.1.1 and 4.1.2.

4.1.1 Mapper-1. The map phase is used to find the local maximum and minimum length of strings from R that will be passed to each reducer in the next stage. The mappers will compute the maximum and minimum of the subset of data that they have received. Let the numReducers be the number of reducers that shall be used in the next stage. Every mapper has a buffer of the same size as numReducers. The buffer maintains the minimum and maximum lengths of strings that are passed to the reducers. The buffer is initialized to contain sentinel values corresponding to each reducer. This is done by using the hash-based distribution method to determine the string presence in which reducer. Firstly, note down the length of the join key as follows:

Length=len(key), where key is the join key

Then, use the same formula, as used by Hadoop, to determine the reducer destination, red_dest as follows:

red_dest=(hash(key) & sys.maxsize) % numReducers

Once the reducer destination has been calculated, further update the values of the minimum and maximum lengths of strings in the buffer corresponding to that reducer.

4.1.1.1 Algorithm 1.1 (Mapper-1). This algorithm is dedicated to the mapper phase of Hadoop. It gets the dataset R as input. The mapper aims to compute the minimum and maximum lengths of all the strings that will be passed to each reducer in the next stage. For every <key, value> pair in a subset of data it receives, the mapper computes to which reducer the string will be the map to (based on hash partitioning the key). It uses the reducer destination along with the length of the string to compute the least and greatest length of strings passed to the reducer.

```
Input: <key,value> pairs from relation R
numReducers = number of reducers for the next stage
buffer – stores the minimum and maximum lengths of strings
   to be passed to the reducers in the next stage
   for string r in input:
length = r.length
reducer_destination = (hash(key) &sys.maxsize) %
   numReducers
   compare length with the max and min lengths for
   reducer_destination in the buffer and update accordingly
emit buffer
```

4.1.2 Reducer-1. This reducer calculates the global minimum and maximum lengths of strings that will be passed to each reducer in the next stage. Only one reducer has been used for stage1. It receives as input values of the form <red_dest, min_length, max_length>, where red_dest is the reducer number, min_length is the local minimum length of the strings passed to that reducer (local minimum as it calculated by a mapper using only a subset of the data), max_length is local maximum length of the strings passed to that reducer.

The reducer calculates the overall minimum and maximum lengths of strings corresponding to each reducer, which is shown in Figure 1. It outputs a partition list, which has values of the

form <red_dest, global_min_length, global_max_length>, where red_dest is the reducer number, global_min_length is the global minimum length of the strings passed to that reducer, global_max_length is global maximum length of the strings passed to that reducer.

4.1.2.1 Algorithm 1.2 (Reducer-1). This algorithm shows how the maximum and minimum length of strings has been calculated. Here, only 1 reducer has been used. It takes the minimum and maximum lengths computed by the mappers for the subset of dataset R. Then it calculates the overall minimum and maximum lengths for each reducer in the next stage. It outputs the partition list in the form of <red_destination, global min_length, global max_length>.

```
Input: Set of <red_dest, min_length, max_length>
   For each reducer:
global_min = minimum (min_length from the list
   corresponding to that reducer)
global_max = maximum (max_length from the list
   corresponding to that reducer)
emit <red_dest, global_min, global_max>
```

The inputs to stage 1 are the records from relation R, identified by the tag 'R_tag'. Each mapper works on a subset of the input and outputs the local least and greatest length of strings for a particular reducer. Let the number of reducers in the stage 2 be three: red0, red1, red2. Only the records from the R relation are sent to the mapper. So, the input for the mapper is as shown in Table 2: -

Mapper computes two things for each string: the reducer destination, red_dest, i.e., the reducer to which the string will map to in the next phase (based on hash-based shuffling done by Hadoop) and the length of the key. For instance, consider the string with RID 4. Assume that red_dest for this string is red0. The length of this string is 30.

Thus, for each red_dest, the mapper computes the lengths of the strings that are being mapped to it. The list used by the mapper to calculates the local minimum and maximum lengths for each reducer. For instance, consider the output of the 1st mapper in the above Figure 1 "red0 1 150". red0 is the red_dest, 1 is the local minimum length, and 150 is the local maximum of the strings that are mapped to red0.

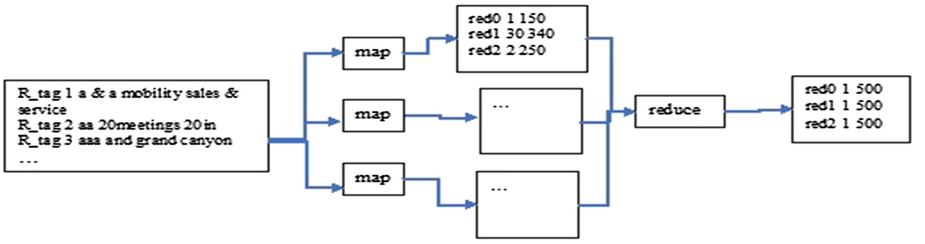

Figure 1.
Working of Stage 1.

Tag	RID	Key (the 1st string of the entire record – used for joining)	Value (the rest of the record)
R_tag	1	A	& a mobility sales & service
R_tag	2	Aa	20meetings 20in 20charlotte
R_tag	3	Aaa	and grand canyon discounts
R_tag	4	aaacookcountyconsolidation.com	Null
R_tag	5	Aaacoman	Null

Table 2.
Input to the Mapper.

The output of each mapper is passed to the reducer. The reducer computes the global minimum and maximum lengths of the strings for each red_dest. For example, suppose for red0, the outputs given by the mappers are:

red0 1 150

red0 30 200

red0 45 500

Then the global minimum is 1 and global maximum is 500. Hence the output of the reducer is:-

red0 1 500

The final output of the reducer i.e. the list of global min and max lengths for each of the reducers is the partition list which will be required in the next stage.

4.2 Stage 2

The aim of stage 2 (MapReduce framework of candidate pair generation algorithm as MR-Pass Join) is to distribute the data appropriately and to generate and verify all candidate pairs. This stage receives all records from relations R and S. More details of the mapper and reducer phase are explained in Sections 4.2.1 and 4.2.2.

4.2.1 Mapper-2. The mapper of stage 2 uses the partition list which was generated by the reducer in the previous stage. The partition list specifies the minimum and maximum lengths of the strings from R that will be passed to the reducers of stage 2 in Figure 2. This list needs to ensure that all the strings from S that could be joined with a string from R are also passed to the same reducer that contains that string.

The point to consider, strings from both the relations R and S are distributed as follows: if the record belongs to the relation R (by checking the relation tag), the reducer destination is computed by the hash-based distribution technique as used by Hadoop (discussed previously in stage 1). If the record belongs to the relation S, then first the length of the join key is calculated. The record from S is passed to all the reducers for which length of its join key belongs to the range [min_len + T, max_len + T], where min_len, max_len are the least and greatest length of all strings passed to that reducer and T is the edit distance threshold. The min_len and max_len of the corresponding reducer are found from the partition list.

After deciding upon as to which reducer the record from S must be passed, the mapper proceeds to find the segments of the join key. If there are segments, then pass the list of segments as well as the original join key and the rest of the records to all the reducers, which have been calculated above.

4.2.1.1 Algorithm 2.1 (Mapper-2). This algorithm gets datasets R and S as inputs. The partition list computed in stage 1 is broadcasted to all the mappers. For strings belonging to R, the reducer is computed according to the hash distribution based on the key. The key value pair is then emitted along with the reducer number. For strings in S, it uses the partition list to determine all the reducers to which the string should be passed to (based on the length of the

Figure 2.
Execution of work of stage 2.

join key). The key-value pair, the list of segments of the key along with the reducer number is emitted for all the reducers to which the string should be shuffled to.

```
Input: <key, value> pairs from relation R and S and partition
   list from stage 1
for every <key,value> pair X in input:
if X belongs to relation R:
red_dest is as computed using the hash-based distribution
   technique
emit <red_dest,X>
else:      //from relation S
length = X.length
for every red_dest in partition list:
if length is in [min_len – T, max_len + T]:
addred_dest to red_dest_list
segments = list of T + 1 segments of X.key by even partition
   scheme
for every red_dest in red_dest_list:
emit <red_dest,X,segments>
```

T, r_len + T], where T is the edit distance threshold, we further check to see whether the records from S and R form a candidate pair. If they form a candidate pair, then there is a need to verify it. If the pair is verified, the RIDs (record IDs) of both the records, with the RID of the record from R preceding the RID of the record from S, are printed followed by the value of the edit distance between the join keys of the two records as shown in Figure 2.

4.2.1.2 Algorithm 2.2 (Reducer-2). The output from the mapper already has the reducer destination (red_dest). The partition method uses this to distribute the data accordingly. The data distributed among reducer by using the red_dest computed by the mapper for every <key, value> pair. This algorithm gets the output of the mapper as input. For every string in R, it compares with all strings in S (with the length within the constraints) and checks if it is a candidate pair by the multi-match aware method. If it is a candidate pair, it verifies the pair. If it satisfies the edit distance threshold, then the RIDs of the 2 strings along with their edit distance value is emitted.

```
buffer – to store segment list of strings from relation S
for every < key,value> pair X in input:
for every <key,value> pair X in input:
If X ϵ S:
add X.segments to buffer
else:
for every string s in buffer such that X.length – T  < s.
   length < X.length + T:
Generate substrings of X.key using the substring selection
   methods described
if s and X.key form a candidate pair using Multi-match aware
   method:
verify (s,X.key)
if accepted:
Emit <s.RID, X.RID, edit_distance>
```

From Figure 2, the input to the map phase are records from both relations R and S. Using the partition list generated by the stage 1, the mapper computes the reducer(s) to which the record from S should be passed. For records belonging to S, the mapper also emits the list of segments of the join key along with the start position. The reducer after receiving a subset of the relation R along with the required strings from S generates candidate pairs and verifies them.

Thus, the final output is a set of RID (record IDs) pairs with the value of edit distance between them. Consider the example 1, tuple 9, Record from the relation S, as in Table 3:-

The length of the key is 30. Let the edit distance be $T = 5$. An even partition scheme will be followed by dividing this string into segments. According to this scheme, $k = 30 - [3 - /(5 + 1))*(5 + 1)) = 0]$. Thus, the length of the $T + 1 = 5 + 1 = 6$ segments will be $[(30/5 + 1))] = 5 = 5$. For each segment, the mapper outputs a pair: the start position (the position from which the segment begins) and the segment itself. So, for this example string the segment list from the mapper is {(0, aaaco), (5, okcou), (10, ntyco), (15, nsoli), (20, datio), (25, n.com)} (we have considered strings to start from the index 0). This procedure is illustrated in algorithm 2.1.

The reduce phase (algorithm 2.2) is responsible for generating substrings of the strings from relation R for each segment of the strings from S. As an example, let us take the string "aaacoman" belonging to the relation R. For each segment of the complete segment list (pl. refers Table 3) the substrings of this string have been computed. So, for the 1st segment with start position=0, the list of substrings generated for this segment are {'aaaco'}. Similarly, for the 2nd segment with start position=5, the list of substrings is {'oman', 'man', 'an']}. For our example, no substrings are generated for the rest of the segments.

When a pair is identified to be a candidate one, it is sent for verification. As explained, the extension-based verification and sharing-computation algorithms are used to verify the pair. If they pass the edit distance threshold then, the generated output of strings would be recorded ids. In case of our example, the pair, although being a candidate one, fails to satisfy the edit distance threshold.

Stages 1 and 2 are dedicated to the MR Pass Join which is a MapReduce version of the pass join. It can handle the strings of both the length viz short and long, while existing techniques can handle one type at a time. The proposed method uses 2-phase verification; merge key-value pairs & light-weight filter unit for redundancy control. It is the better method for candidate pair generation than the existing methods in the literature. As well as it is a scalable solution for the string similarity join.

4.3 Stage 3

In this stage, the basic record join algorithm has been used, as this is the recommended alternative as opposed to one phase record join. However, it is robust to the data skewness. So here the aim is to use MRFA algorithm concepts to handle data skewness [24].This stage receives the RID pairs generated by the previous stage along with the records of R and S. In the map phase, for each record, a <RID, record> pair is emitted. For each RID pair, two <key, value> pairs are emitted, where each pair uses one of the RIDs of the RID pairs as the key. The value for both the pairs is the RID pair along with the value of the edit distance computed in the previous stage.

According to the original algorithm presented by Vernica, R.; Carey, M. J & Li, C; in [25] every reducer will receive exactly one record, and the rest of the data will be RID pairs along

Tag	RID	Key	Value
S_tag	4	aaacookcountyconsolidation.com	Null

Table 3.
Record from relation S.

with the similarity function value. At this point, there is a likelihood of having skewness since one record can have join results with many other records, i.e., there may be many RID pairs for a single record. Thus, there is a high possibility of skewness occurrence in stage 3. Here, MRFA [24] is used to deal with the skewness at this point. Since there is a possibility, the number of RID pairs for one record can be huge, therefore use of the MRFA algorithm to compute the frequencies of these RID pairs. Further, it will determine whether there is a need to divide the RID pairs into buckets or not. Hence the frequencies are computed. If the frequency is higher than a specific threshold frequency [50,51], then these RID pairs will be divided into buckets and distributed to different reducers. The tuple containing the entire record of the relationship will also be replicated to all the reducers which have the RID pair bucket as shown in Figure 3.

After such distribution is done, for each RID pair <key, value> pairs are emitted where the key is the RID pair, and the value is the record (either from relation R or S) and the edit distance of the RID pair. The next phase uses an identity mapper which emits the previous <key, value> pairs. This list is then sorted according to the key (which is the RID pair). The reducer then merges the two records, which have the same key, along with the edit distance value and output results of stage 2 (see Figure 4).

RID pairs have the tag RID_pair_tag. If the number of RID pairs for one record exceeds the user-defined threshold frequency, then they are divided into buckets (in the example above, the RID pairs are divided into four buckets, and each bucket is distributed to one reducer). The record, which has the tag record_tag, is replicated across all the reducers to which the buckets of the RID pairs have been distributed, as shown in the Figure 3. RID_pair_tag and record_tag is used to differentiate between the record and the RID pairs. Since stage 3 is very complicated, four mapper-reducer phases have been applied in this. All the phases are described through the algorithms.

In Figure 3, four reducers i_0, i_1, i_2 and i_3 have been considered. RID_pair_tag indicates that the record is an RID pair, while record_tag indicates that the record belongs to relation R or S. In both the cases value is the entire record.

4.3.1 Algorithm 3.1 (Mapper-Reducer-3(i)). This is a map phase only. It takes datasets R and S and the RID pairs emitted from the previous stage as input. For every record it emits the

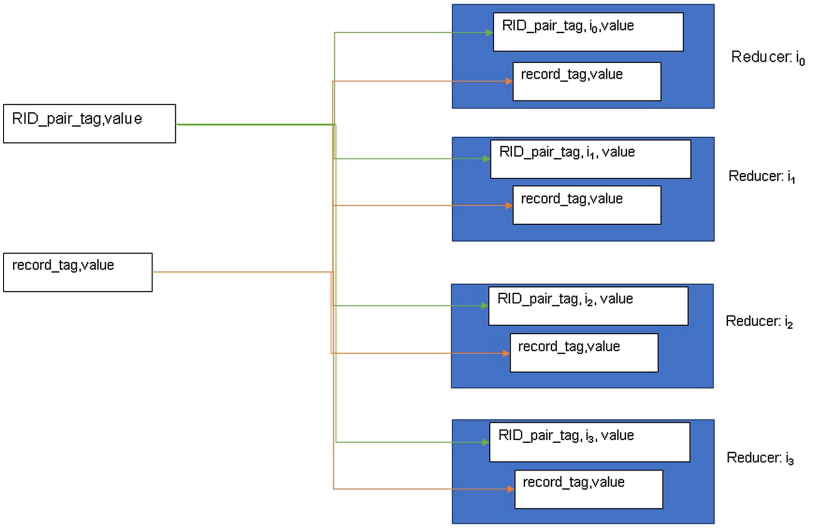

Figure 3.
Workflow of stage 3.

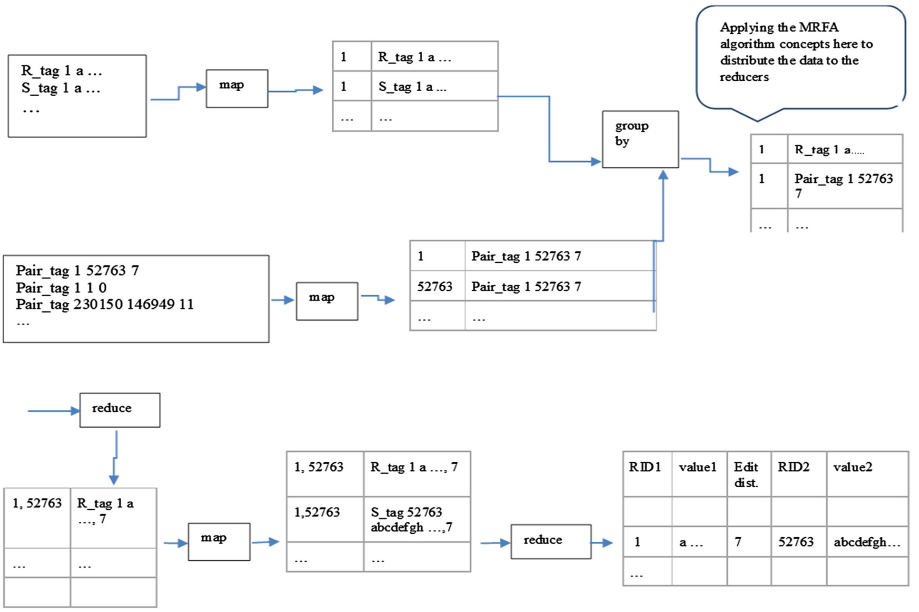

Figure 4.
Basic record join used with MRFA-SSJ to combat skewness in string similarity join.

same with a record tag, while for every RID pair it emits the RID pair twice, each with one of the RIDs as their key.

```
Input: Records from relation R and S and RID pairs list from
   previous stage
for <key,value> pair X in input
if X is a record:
emit <Record_tag, X.RID, X>
else:            // RID pair
emit <Pair_tag, X.RID1,X>
emit <Pair_tag, X.RID2,X>
```

4.3.2 Algorithm 3.2 (Mapper-Reducer-3(ii)). In this phase the frequency of the RID pairs that have the same key is calculated. The mapper emits '1' for each RID pair and the reducer sums up all the frequencies for each key.

```
//Using MRFA to distribute data so as to handle skewness
   calculates the frequencies of RID pairs
MRFA-SSJ mapper-3(ii)
for <key,value > pair X in input:
if X is an RID pair:
emit <X.RID, X.tag, 1, X.value>
MRFA-SSJ reducer- 3(ii)
for every key K in input:
frequency = sum of frequencies corresponding to that key
emit <K, frequency>
```

4.3.3 Algorithm 3.3 (Mapper-Reducer-3(iii)). Here, using the frequencies computed in the previous MapReduce phase, it is determined whether the RID pairs of the same key are to be split into buckets, based on a user-defined threshold or not. If the RID pairs are split into buckets, then the corresponding record from the datasets R and S are replicated to all the reducers to which the buckets are sent. The partitioner distributes the data according the reducer destination thus computed. The reducer emits all the RID pairs along with the corresponding the record from R or S that matches one of the RIDs.

3.3.1. MRFA-SSJ mapper 3(iii)

```
/*this function will determine whether or not to split the data
   into buckets and will also compute the number of buckets,
   if required*/
Hash_Pointer = 1
Partition_Pointer = 2/*THRESHOLD FREQ = T: user-defined
   frequency*/
if freq < = T
Emit <join_key, Hash_ Pointer,1>
Else
Nb_buckets = ⌈(frequency/THRESHOLD_FREQ)⌉
Emit <join_key,Partition_ Pointer,Nb_buckets>
```

3.3.2 MRFA-SSJ reducer 3(iii)

```
/*it will generate the reducer destination for each record in a
   random manner*/
Hash_Pointer = 1
Partition_Pointer = 2
if the tuple contains the RID pair
if freq_Pointer = = hash_Pointer
Emit <join_key, −1, record>
else
random_dest = (random_integer + SRAND(Nb_buckets))%
   Nb_buckets
Flag_Pointer = Partition_Pointer
Emit <join_key,Flag_ Pointer,random_dest,record>
if the tuple contains the record
if Nb_buckets = = 1
Emit <join_key, −1, record>
Else: for i = 0 to Nb_buckets − 1
rand_dest ← (random integer + i)% Nb buckets
Flag_Pointer ← Replication_Pointer
Emit < join key,Random _dest,Flag_ Pointer , value>
Else: For record in buff:
If record.RID matches any one of the RIDs in X: Emit <X.RID1,
   X.RID2, record, X.edit_distance>
```

4.3.4 Algorithm 3.4(Mapper-Reducer-3(iv)). 4.3.4.1 MRFA-SSJ partitioner. The output from the mapper already has the reducer destination. The partition technique uses this method distribute the data accordingly.

```
for <key,value> pair Xin input:
reducer_dest = X.random_dest
ifreducer_dest ! = (−1)
returnreducer_dest % numReducers
Else
returnHashCode(join_key) % numReducers
Reducer
buff = buffer that stores records from the relations
For <key,value> X in input:
If X is a record:
buff.append(X)
```

Finally, the last phase of mapper-reducer in stage 3 is dedicated to find out the identity mapper for every RID pair. It is used to divide the keys in between map-reduce. An identity mapper is used which maps the output from the previous phase. The reducer, for every RID pair merges the corresponding record values. It emits the two records along with the edit distance value.

```
3.4.1 Mapper -3(iv)
//Identity mapper
for <key,value> X in input:
emit X
3.4.2 Reducer – 3(iv)
for every RID pair X
record1, record2 = 2 record values corresponding to X
emit <record1, record2, X.edit_distance>
```

Here, stage 3 has been illustrated by using Example1 as in Tables 1, 4 and 5: -

- For records, the mapper outputs in the form <key, value> pairs: for example – <1, R_tag 1 a and a mobility sales & service>.
- Consider the RID pair, "Pair_tag 1 52,763 7". For such RID pair, the output as shown in Table 5:

Tag	RID 1	RID 2	Edit distance
Pair_tag	1	52,763	7
Pair_tag	1	1	0
Pair_tag	230,150	146,949	11

Table 4. RID pairs generated from stage 2.

Key	Value
1	Pair_tag 1 52,763 7
52,763	Pair_tag 1 52,763 7

Table 5. output of RID pairs.

Now, once the RID pairs generate the records are joined together, at the same time there is a need to get ensure to avoid skewness which can hamper the performance. One record can be paired with many records which can cause skewness. For this MapReduce phase, algorithm 3.2 has been used. Using the concepts from MRFA, distribute the output of stage 2 on the basis of frequency. Then compute the total number of RID pairs for 1 record. If this frequency is greater than the user-defined threshold frequency then the entire set of RID pairs is divided into buckets, which are distributed to different reducers. The record is then replicated across all those reducers for the join operation. Then for RID pairs, the record is set as the value while the RID pair acts as the key. This is one half of the result which is computed by the MapReduce phase in algorithm 3.3. To combine the two halves, group all the <key, value> pairs according to the keys (here the key is the RID pair) explained in algorithm 3.4.

Two <key, value> pairs sharing the same key are thus the two halves of the result. The two values (i.e. records) are then combined to give the final results.

Stage 3 is dedicated to the most crucial and important part of this proposed technique. It is not affected method as other existing algorithms in literature are sensitive to data skewness. Analysis has been done by following the ZipF law. The detailed explanation and comparative study has been shown in Section 4 (pl.refer Experimental Analysis).

4.3.5 Illustration of the proposed approach. The algorithm begins with Stage 1, which computes how the data from relation R will be distributed. It outputs the minimum and maximum lengths of the strings to each reducer. So, for instance, if the shortest string received by the reducer and its length is 20 and the longest string is of length 500, then the output will be the reducer number followed by the values of the minimum and maximum lengths of the strings (using algorithms 1.1 and 1.2).

For the second stage, the map phase process strings from S as follows: Consider the string from relation S "aaacookcountyconsolidation.com". Its length is 30. Let the edit distance, $T = 5$. An even partition scheme will be followed to divide this string into segments. According to this scheme, $k = 30 - [30/(5+1))*(5+1)] = 0]$. Thus the length of the $T + 1 = 5 + 1 = 6$ segments will be $[(30/(5+1))] = 5$. For each segment, the mapper outputs a pair: the start position (the position from which the segment begins) and the segment itself. So for this example string the segment list from the mapper is {(0, aaaco), (5, okcou), (10, ntyco), (15, nsoli), (20, datio), (25, n.com)} (here, the considered strings has been started from the index 0). This procedure is illustrated in Algorithm 2.1.

The reduce phase (algorithm 2.2) is responsible for generating substrings of the strings from relation R for each segment of the strings from S. As an example, consider the string "aaacoman" belonging to the relation R. For each segment of the used segment list has been computed. So, for the 1st segment with start position=0, the list of substrings generated for this segment is {'aaaco'}. Similarly, for the 2nd segment with start position=5, the list of substrings is {'oman', 'man', 'an']}. For our example, no substrings are generated for the rest of the segments.

As and when the list of substrings for each segment is generated the next step is to check whether the segment matches any one of the substrings in the list or not. In this example, the 1st segment matched the substring that was generated. Hence these two strings have been concluded to be a candidate pair. When a pair is identified to be a candidate one, it is sent for verification. As explained, the extension-based verification and sharing computation algorithms are used to verify the pair. If they pass the edit distance threshold, then the record IDs of both the strings is the output result. In case of our example, the pair, although being a candidate one, fails to satisfy the edit distance threshold.

Now, once the RID pairs generate the records they are joined together at the same time there is a need to handle skewness which can hamper the performance. One record can be paired with many records which can cause skewness. For this MapReduce phase, algorithm 3.2 has been used. Using the concepts from MRFA, distribute the output of stage 2 based on frequencies. Then compute the total number of RID pairs for one record. If this frequency is greater than the user-defined threshold frequency then the entire set of RID pairs are divided into the number of buckets, which are distributed to different reducers. Now, records get start replicating across all those reducers for the join operation. Then for RID pairs, the record is set as the value while the RID pair acts as the key. This is one half of the result which is computed by the MapReduce phase in algorithm 3.3. To combine the two halves, group all the <key, value> pairs according to the keys (here the key is the RID pair) explained in algorithm 3.4. Two <key, value> pairs sharing the same key are thus the result will come in two halves. Further, these two halves (i.e. records) have been combined for the final outcome. After this illustration, experiment analysis (pl. refers to section 4) follows the detailed setup for the proposed approach with graphical results and comparative study of the compared techniques.

5. Experimental analysis

In this section, we are analyzing the performance of the proposed method. We have compared our work with the best methods, namely: Improved Repartition Join [52], Standard Repartition Join [52] and MRFA Join algorithm [24]. We used the set of 15 node clusters, where every node consists of Intel Xenon processor E5520 and contains 3GHz with four cores, 12GB of RAM and five 200GB Hard disks. We had also set up the extra node for running the master node, to manage Hadoop jobs as well as HDFS working.

We have installed the Hadoop 2.6.0, and every node consists of the Ubuntu 9.04, 64-bit, server edition operating system, Java 1.6 with a 64-bit server JVM. We have done some changes in a default Hadoop configuration just to maximize the parallelism and minimize the running time; the default block size of each block is 64 bit, but we set the block size of HDFS to 128 bit, and allocate 1GB of virtual memory to each node and 1GB of virtual memory to each map/reduce task. It runs eight maps, and eight reduce tasks in parallel on each node. As well as we had set the replication factor to 1 and disable the unpredictable task execution feature.

We had followed the Zipf law to study the effect of data skewness on the proposed method during the join operation. To examine the impact of skewness, it was incorporated artificially in the join operations by following the Zipf distribution law [53,54]. The factor of Zipf law varies from 0 to 1. Here “0” stands for the uniform data distribution which carries zero skewness whereas “1” stands for highly skewed data. In this experiment, we fix the size of the input up to 10 GB of data on the right side of the table and one GB of data in the left side table. So, the range of final result varies from 5 GB to 40 GB records.

We have analyzed during the experiment that our proposed algorithms outperform other existing algorithms which we have compared for low to medium skewness. As confirmed from the series of experiments performed on Query log, DBLP datasets and for more realistic results we have used the real IPS and cookies dataset. It can be shown that the proposed method is better than the other existing methods available in the literature for handling data skewness in big data for the Basic Record Join method. Further, the proposed method has been compared with the existing techniques namely: Improved Repartition Join [52], Standard Reparation Join [52] and MRFA Join [24], with respect to the join processing time and reduce shuffle time in Figures 5 and 6 respectively.

When skewness is high, all the existing algorithms are not able to perform except our proposed algorithm. In the skew factor ranging from 0.6 to1, it shows that the job failed for Improved repartition join and Standard repartition join due to the lack of memory as well as it

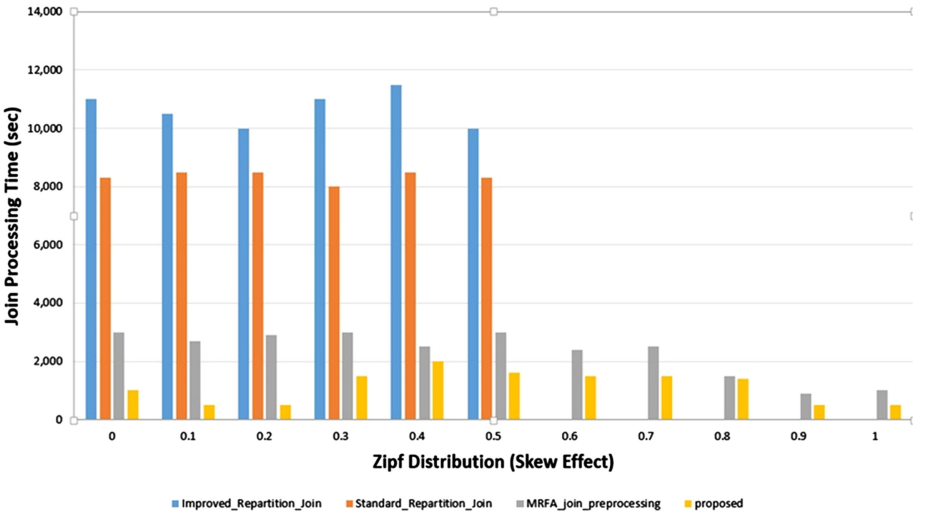

Figure 5.
Job processing time with respect to skew effect in data and the time taken by techniques to process it completely.

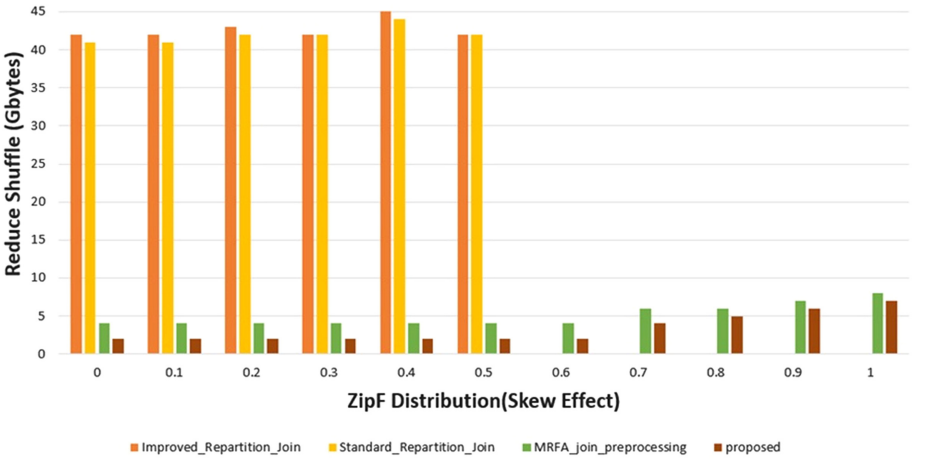

Figure 6.
Shuffle time with respect to the skewness in data.

affects the scalability of the algorithm. Both of the join algorithms are skew sensitive algorithms whereas, MRFA sustains for the entire range of skewness factor, but it does not ensure the scalability. While it is compared with the proposed method, it has been noticed that the proposed method takes very less time to process the jobs at high data skewness. Hence the proposed method is skew-insensitive for character-based string similarity join. Comparison of different techniques has been made in Table 6.

6. Conclusion

In this paper, a new approach has been presented for handling data skewness in character-based string similarity join using the MapReduce framework, and we have proposed an algorithm namely MR- Pass Join for handling this task. After performing the experiments, we observed that the candidate pair generation method is neither generating false positive nor false negative results. We found that the proposed algorithm ensures scalability with reasonable query processing time

Characteristics	MRSim Join	V-Smart Join	Mass Join	Improved Repartition Join	Standard Repartition Join	MRFA	Proposed (MRFA-SSJ)
Data Skew	Insensitive	Insensitive	Insensitive	Sensitive	Sensitive	Less Sensitive	Insensitive
Scalability	Less	Moderate	Less	Less	Less	Below Moderate	High
Memory Requirement	Lack of Memory	Moderate Amount of Memory	Lack of Memory	Lack of Memory	Lack of Memory	Moderate Amount of Memory	High Usage of Memory

Table 6.
Comparison of different Techniques.

for string similarity database join. To obtain the results of the proposed approach MRFA-SSJ has been compared with the best-known solutions namely Improved Repartition Join, Standard Repartition Join, and MRFA Join. Our results are found to be better, and result analysis has been done to the Zipf law in respect of reducing shuffling time and joins processing time.

The proposed algorithm is highly skew insensitive, and it is a scalable method with respect to the high usage of memory. Proposed algorithm survives till the highest value of the skew factor "1" while the existing algorithms survive during the experiment till skew factor 0.5 in respect of Zipf distribution. As per our analysis, present algorithms are not able to give good results as they are skew sensitive.

This analysis has been done on the set-up of the set of 15 cluster nodes, where we provide 1 GB virtual memory to every node and five hard disks of 200 GB. Size of map nodes in Hadoop extended from 64 bit to 128 bit for better analysis. We had also fixed the size of input 10GB in both, right & left table after the analysis the outcome varies from 5 GB to 40 GB.

It has been noticed that the proposed algorithm can handle highly skewed data whereas other algorithms cannot, and hence it is a skew insensitive approach. The proposed algorithm is also found to be scalable and better in load balancing. It outperforms the existing methods which fail to handle data skewness in similarity joins.

References

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

[2] A. Metwally, C. Faloutsos, V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors, Proc. VLDB Endow. 5 (8) (2012) 704–715.

[3] M. Wang, T. Nie, D. Shen, Y. Kou, G. Yu, Intelligent similarity joins for big data integration, in: Web Information System and Application Conference (WISA), 10th, IEEE, 2013, pp. 383–388.

[4] C. Batini, A. Rula, M. Scannapieco, G. Viscusi, From data quality to big data quality, J. Database Manage. 26 (1) (2015) 60–82.

[5] L. Kolb, A. Thor, E. Rahm, Load balancing for mapreduce-based entity resolution, in: 28th International Conference on Data Engineering (ICDE).2012, IEEE, 2012, pp. 618–629.

[6] P. Budikova, M. Batko, D. Novak, P. Zezula, Inherent fusion: towards scalable multi-modal similarity search, J. Database Manage. (JDM) 27 (4) (2014) 1–23.

[7] M.S. Al-kahtani, L. Karim, An efficient distributed algorithm for big data processing, Arab. J. Sci. Eng. (2017) 1–9.

[8] I. Elgedawy, NASEEB: an Escrow-based approach for ensuring data correctness over global clouds, Arab. J. Sci. Eng. 39 (12) (2014) 8743–8764.

[9] M. Sedighizadeh, M. Ghalambor, A. Rezazadeh, Reconfiguration of radial distribution systems with fuzzy multi-objective approach using modified big bang-big crunch algorithm, Arab. J. Sci. Eng. 39 (8) (2014) 6287–6296.

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

[11] Z. Liu, Q. Zhang, M.F. Zhani, R. Boutaba, Y. Liu, Z. Gong, Dreams: dynamic resource allocation for mapreduce with data skew, in: IFIP/IEEE International Symposium on Integrated Network Management (IM), 2015, pp. 18–26.

[12] J. Wang, J. Feng, G. Li, Trie-join: efficient trie-based string similarity joins with edit-distance constraints, Proc. VLDB Endow. 3 (1–2) (2010) 1219–1230.

[13] C. Xiao, W. Wang, X. Lin, Ed-join: an efficient algorithm for similarity joins with edit distance constraints, Proc. VLDB Endow. 1 (1) (2008) 933–944.

[14] G. Li, D. Deng, J. Wang, J. Feng, Pass-join: a partition-based method for similarity joins, Proc. VLDB Endow. 5 (3) (2011) 253–264.

[15] J. Qin, W. Wang, Y. Lu, C. Xiao, X. Lin, Efficient exact edit similarity query processing with the asymmetric signature scheme, in: Proceedings of the ACM SIGMOD International Conference on Management of data, 2011, pp. 1033–1044.

[16] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering?: An adaptive framework for similarity join and search, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2012, pp. 85–96.

[17] W. Wang, J. Qin, C. Xiao, X. Lin, V. Shen, Chunk Join: an efficient algorithm for edit similarity joins, in: IEEE Transactions on Knowledge and Data Engineering, 2013, pp. 1916–1929.

[18] I. Kamel, Z. Al Aghbari, T. Awad, MG-join: detecting phenomena and their correlation in high dimensional data streams, Distrib. Parallel Databases 28 (1) (2010) 67–92.

[19] D. Deng, G. Li, S. Hao, J. Wang, J. Feng, Massjoin: a mapreduce-based method for scalable string similarity joins, in: Data Engineering (ICDE) 30th International Conference, 2014, pp. 340–351.

[20] G. Li, D. Deng, J. Feng, A partition-based method for string similarity joins with edit-distance constraints, ACM Trans. Database Syst. (TODS) 38 (2) (2013) 9–18.

[21] J. Wang, G. Li, J. Feng, Extending string similarity join to tolerant fuzzy token matching, ACM Trans. Database Syst. (TODS) 39 (1) (2014) 7–52.

[22] C. Xiao, W. Wang, X. Lin, J.X. Yu, G. Wang, Efficient similarity joins for near-duplicate detection, ACM Trans Database Syst. (TODS) 36 (3) (2011) 15.

[23] A. Das Sarma, Y. He, S. Chaudhuri, Clusterjoin: a similarity joins framework using map-reduce, Proc. VLDB Endowment. 7 (12) (2014) 1059–1070.

[24] M.A.H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using mapreduce, Proc. Comput. Sci. 29 (2014) 145–158.

[25] R. Vernica, M.J. Carey, C. Li, Efficient parallel set-similarity joins using MapReduce, in: Proceedings of the ACM SIGMOD International Conference on Management of data, 2010, pp. 495–506.

[26] C. Zhang, F. Li, J. Jestes, Efficient parallel knn joins for large data in mapreduce, in: Proceedings of the 15th International Conference on Extending Database Technology, 2012, pp. 38–49.

[27] W. Yan, Y. Xue, B. Malin, Scalable load balancing for mapreduce-based record linkage, IEEE 32nd International, in: Performance Computing and Communications Conference (IPCCC), 2013, pp. 1–10.

[28] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune in action: mitigating skew in mapreduce applications, Proc. VLDB Endow. 5 (12) (2012) 1934–1937.

[29] Y. Kwon, K. Ren, M. Balazinska, B. Howe, J. Rolia, Managing skew in Hadoop, IEEE Data Eng. Bull. 36 (1) (2013) 24–33.

[30] F. Afrati, N. Stasinopoulos, J.D. Ullman, A. Vassilakopoulos, Sharesskew: An algorithm to handle skew for joins in mapreduce, arXiv (2015) 1–12.

[31] Z. Tang, W. Ma, K. Li, K. Li, A data skew oriented reduce placement algorithm based on sampling, in: IEEE Transactions on Cloud Computing, 2016, pp. 1–14.

[32] M. Abdel-Basset, G. Manogaran, D. El-Shahat, S. Mirjalili, A hybrid whale optimization algorithm based on local search strategy for the permutation flow shop scheduling problem, Future Generation Comput. Syst. 85 (2018) 129–145.

[33] M.G. Abdel-Basset, L. Abdel-Fatah, S. Mirjalili, An improved nature inspired meta-heuristic algorithm for 1-D bin packing problems, Person. Ubiquit. Comput. (2018) 1–16.

[34] M.M.G. Abdel-Basset, A. Gamal, F. Smarandache, A hybrid approach of neutrosophic sets and DEMATEL method for developing supplier selection criteria, in: Design Automation for Embedded Systems, 2018, pp. 1–22.

[35] Mohamed Abdel-Basset, M. Gunasekaran, M. Mohamed, F. Smarandache, A novel method for solving the fully neutrosophic linear programming problems, Neural Comput. Appl. (2018) 1–11.

[36] Mohamed Abdel-Basset, M. Gunasekaran, A.E. Fakhry, I. El-Henawy, 2-Levels of clustering strategy to detect and locate copy-move forgery in digital images, Multimedia Tools Appl. (2018) 1–19.

[37] M. Abdel, M. Mohamed, Internet of Things (IoT) and its Impact on supply chain: a framework for building smart, secure and efficient systems, Future Gener. Comput. Syst. (2018) 1–15.

[38] M. Basset, G. Manogaran, M. Mohamed, E. Rushdy, Internet of things in smart education environment: supportive framework in the decision-making process, e4515, in: Concurrency and Computation: Practice and Experience, 2018, pp. 1–12.

[39] M.M.G. Abdel, H. Rashad, A.N.H. Zaied, A comprehensive review of quadratic assignment problem: variants, hybrids and applications, J. Ambient Intel. Human. Comput. (2018) 1–24.

[40] MohamedAbdel-Basset, M. Gunasekaran, M. Mohamed, N. Chilamkurti, Three-way decisions based on neutrosophic sets and AHP-QFD framework for supplier selection problem, Future Gener. Comput. Syst. (2018) 1–39.

[41] M. Bamha, G. Hains, A skew-insensitive algorithm for join and multi-join operations on shared nothing machines, in: International Conference on Database and Expert Systems Applications, 2000, pp. 644–653.

[42] M. Bamha, in: An, optimal skew-insensitive join and multi-join algorithm for distributed architectures, Springer, Berlin Heidelberg, 2005, pp. 616–625.

[43] Y.N. Silva, J.M. Reed, L.M. Tsosie, MapReduce-based similarity join for metric spaces, in: Proceedings of the 1st International Workshop on Cloud Intelligence, 2012, pp. 3–10.

[44] Y.U. Minghe, L.I. Guoliang, D.E.N.G. Dong, F.E.N.G. Jianhua, String similarity search and join: a survey, Front. Comput. Sci. 10 (3) (2016) 399–417.

[45] P. Selvaramalakshmi, S.H. Ganesh, J.J. Manoharan, Survey of string similarity join algorithms on large scale data, Int. J. Innov. Eng. Technol. (IJIET) (2016) 100–104.

[46] B. Yang, H.J. Kim, J. Shim, D. Lee, S.G. Lee, Fast and scalable vector similarity joins with MapReduce, J. Intel. Inform. Syst. 46 (3) (2016) 473–497.

[47] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, J. Feng, A unified framework for string similarity search with edit-distance constraint, VLDB J. 26 (2) (2017) 249–274.

[48] T. Seidl, S. Fries, B. Boden, MR-DSJ: distance-based self-join for large-scale vector data analysis with MapReduce, BTW 214 (2013) 37–56.

[49] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Handling partitioning skew in mapreduce using leen, Peer-to-Peer Network. Appl. 6 (4) (2013) 409–424.

[50] M. Bamha, M.A.H. Hassan, Scalability and optimisation of groupby-joins in mapreduce, in: Technical report LIFO, Universited'Orleans, France, 2015, pp. 1–20.

[51] M. Bamha, G. Hains, Frequency-adaptive join for shared nothing machines, Parallel Distrib. Comput. Pract. 2 (3) (1999) 333–345.

[52] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, Y. Tian, A comparison of join algorithms for log processing in mapreduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 975–986.

[53] Z. Bi, C. Faloutsos, F. Korn, The DGX distribution for mining massive, skewed data, in: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2001, pp. 17–26.

[54] J. Lin, The curse of zipf and limits to parallelization: a look at the stragglers problem in mapreduce, in: 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009, pp. 57–62.

Corresponding author
Oscar Castillo is the corresponding author and can be contacted at: ocastillo@tectijuana.mx