Journal of Computer Science and Cybernetics, V.39, N.2 (2023), 101–124
DOI no 10.15625/1813-9663/17589

SIMILARITY ALGORITHMS FOR FUZZY JOIN COMPUTATION IN BIG DATA PROCESSING ENVIRONMENT

ANH CANG PHAN[1,*], THUONG CANG PHAN[2]

[1] *Vinh Long University of Technology and Education, 73 Nguyen Hue Street, Ward 2, Vinh Long City, Vinh Long Province, Viet Nam*

[2] *Can Tho University, 3/2 Street, Ninh Kieu District, Can Tho city, Viet Nam*

Abstract. Big data processing is attracting the interest of many researchers to process large-scale datasets and extract useful information for supporting and providing decisions. One of the biggest challenges is the problem of querying large datasets. It becomes even more complicated with similarity queries instead of exact match queries. A fuzzy join operation is a typical operation frequently used in similarity queries and big data analysis, in which similarity functions are the key for all fuzzy join algorithms. The functions are used to quantify the similarity of two join keys. Currently, there is very little research on similarity functions for parallel fuzzy joins, thus it poses significant barriers to the efforts of improving query operations on big data efficiently. As a result, this study overviews the similarity algorithms for fuzzy joins, in which the data at the join key attributes may have slight differences within a fuzzy threshold. We analyze six similarity algorithms including Hamming, Levenshtein, LCS, Jaccard, Jaro, and Jaro - Winkler, to show the difference between these algorithms through the three criteria: output enrichment, false positives/negatives, and the processing time of the algorithms. Experiments of fuzzy join algorithms are implemented in the Spark environment, a popular big data processing platform. The algorithms are divided into two groups for evaluation: group 1 (Hamming, Levenshtein, and LCS) and group 2 (Jaccard, Jaro, and Jaro - Winkler). For the former, Levenshtein has an advantage over the other two algorithms in terms of output enrichment, high accuracy in the result set (false positives/negatives), and acceptable processing time. In the letter, Jaccard is considered the worst algorithm considering all three criteria mean while Jaro - Winkler algorithm has more output richness and higher accuracy in the result set. The overview of the similarity algorithms in this study will help users to choose the most suitable algorithm for their problems.

Keywords. Fuzzy joins, similarity algorithms, set-similarity joins, big data processing, spark.

1. INTRODUCTION

Big data analysis is currently attracting the interest of researchers to extract valuable information from the huge amount of data that is rapidly increasing. The join operation

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

between large datasets is a typical operation used frequently in data processing. Join performs the association of data from different datasets according to a key join and returns a new dataset. Thus, joining two datasets is performed on tuples with the same key value. However, in practice as well as in science and technology, many concepts are not clearly defined. The data at the join key attributes may have slight differences, but it is still possible to ensure the join condition and returns the right results. Fuzzy Join is an operation that joins different datasets based on a degree of difference in a certain key attribute. Given two datasets $R(Enterprise)$ and $L(Employee)$ as in Table 1 and 2, the join computation between the two datasets $R \bowtie_{Ent\_Add=Emp\_Add} L$ will return a new dataset O as shown in Table 3.

Table 1: Dataset $R(Enterprise)$

Ent_Num	Ent_Name	Ent_Add
11663	More Power Industries	150 North Michigan Ave
11435	Group Services	605 3rd Acenue
13928	Liberty Trading	300 North Meridian Street

Table 2: Dataset $L(Employee)$

Emp_Num	Emp_Name	Emp_Add
10	Catherine Exelby	150 North Michigan Ave
22	Eileen Henderson	605 Third Avenue
65	Jorge Marin	300 North Meridian Street

Table 3: Dataset $O = R \bowtie_{Ent\_Add=Emp\_Add} L$

Ent_Num	Ent_Name	Ent_Add	Emp_Num	Emp_Name	Emp_Add
11663	More Power Industries	150 North Michigan Ave	10	Catherine Exelby	150 North Michigan Ave
13928	Liberty Trading	300 North Meridian Street	65	Jorge Marin	300 North Meridian Street

The result in Table 3 is the association of two datasets R and L with the join key being the addresses of the two datasets. However, with the naked eye, one result is missing because the address "605 Third Avenue" has a little difference in the dataset R. Thus, to have the desired results, it is necessary to perform a fuzzy join operation that accepts a certain possibility of variation in the join key.

In the literature, there have been studies examining set-similarity joins (fuzzy joins) on a single computer [1–5]. Finding pairs of strings with a specific amount of tokens in common was done using inverted-list-based algorithms proposed in [1]. In [2], a prefix filter was proposed to reduce the large number of pairs in data cleaning. In [3, 4], the length filter has been examined. The positional filter and the suffix filter were two additional filters that were suggested in [5]. Concerning parallel fuzzy joins, there have been some studies using the popular MapReduce model. Vernica et al. [6] performed set-similarity joins in parallel using MapReduce. They identified similar records based on string similarity with a set-similarity function, e.g., Jaccard with a similarity threshold of 0.80. Their approach to Jaccard similarity is known to be good in decreasing the running time when the similarity threshold is high. Experiments were conducted in Hadoop with two datasets, DBLP (1.2M publications) and CITESEERX (1.3M publications). Baraglia et al. [7] addressed the similarity self-join

problem to discover all pairs of objects according to a similarity function. They proposed two algorithms based on Prefix-Filtering: the first algorithm performed a Prefix-Filtering in two MapReduce jobs and the second used a remainder file to broadcast data likely to be used by every reducer. This approach is inspired by an earlier one proposed by Elsayed et. al. [8]. The authors ran experiments on a 5-node cluster with Hadoop configuration showing their improvements over the approach in [6]. Afrati et al. [9] evaluated several fuzzy join algorithms for finding all pairs of elements from an input set that meet a similarity threshold. They proposed to use a single MapReduce job and consider the cost of the algorithms including the execution cost of the mappers, the execution cost of the reducers, and the communication cost from the mappers to the reducers. The algorithms were presented in terms of Hamming distance and then extended to edit distance and Jaccard distance. The authors showed that non of the algorithm dominates the others when both communication and reducer costs are considered. Their algorithms came with complete theoretical analysis, but no experimental evaluation is provided, thus Ben Kimmett et al. [10] computed fuzzy joins of binary strings using Hamming Distance from the MapReduce algorithms proposed in [9]. Through experimental results, the authors showed different facets of the algorithms, and from their practical point of view, some algorithms were almost always preferable to others. In [11], Ben Kimmett et al. continued to provide an experimental evaluation of the fuzzy join algorithms proposed in [9] with the more challenging cases of edit and Jaccard distances. The authors provided details of adaptations needed to implement the algorithms based on these similarity measures. In [10, 11], the authors used the Hadoop platform for their experiments.

Das Sarma et al. [12] proposed ClusterJoin to split data into partitions using a set of strong candidate filters to keep each record distribution in a small number of partitions. The approach was compared with several similarity join methods with the distance functions of Euclidean, Cosine, Total Variation, and Jensen Shannon. Deng et al. [13] proposed a scalable string similarity joins, namely Mass-Join, based on MapReduce to solve the problem of low pruning power, many dissimilar pairs sharing the same token cannot be pruned. Experiments were conducted in Hadoop demonstrating both set-based similarity functions and character-based similarity functions. Their approach got similar results when evaluating the scale-up on two set-based similarity functions, e.g., Cosine and Jaccard, since the verification method took little time. Yu et al. [14] provided a survey on string similarity search and join. The authors presented the problem definitions and discussions on similarity search and join algorithms with their variants. They discussed the prefixed filtering method based on the similarity threshold of Jaccard, Cosine, Dice, and Edit distance. No experiment was provided in this study. Rong et al. [2] introduced FS-Join, a distributed similarity join algorithm, using a filter-and-verification framework. A new partitioning method was defined to support efficient string similarity joins. Experiments were set up on a Hadoop cluster running on the Amazon Elastic Compute Cloud with 11 nodes. Fier et al. [15] gave a survey on ten distributed set similarity join algorithms based on MapReduce. They performed a comparative evaluation on twelve datasets of varying sizes and characteristics. The tested algorithms used Jaccard similarity and were deployed on the same Hadoop cluster.

Yan et al. [16] proposed the Pada-Join algorithm for string similarity join on distributed systems with Spark. The authors conducted experiments to show performance differences between string similarity joins on distributed systems and multi-core systems. For rela-

tively small datasets, running on multi-core systems provides good scalability and avoids the overhead of network communication. Zhimin Chen et al. [17] described a scale-out fuzzy join operator that supports customization with a locality-sensitive-hashing based signature scheme. The evaluation of the design was done on the Azure Databricks version of Spark using several real-world and synthetic datasets. Through experimental results, they found that fuzzy join that uses locality-sensitive-hashing signature is significantly faster than a prefix filtering based technique and in case the broadcast fuzzy join is applicable, it is faster than the shuffle version. Tran Thi-To-Quyen et al. [18] proposed to integrate the Bloom filter in fuzzy joins to support fast similarity queries in reducing redundant data. The approach was done by maintaining a bit matrix, with a small false positive rate, and zero false negative rate. They provided experiments showing that the fuzzy filter join can eliminate redundant data, reduce computation costs, and avoid duplicate outputs.

In this study, we will give an overview of the similarity algorithms for join computation on large-scale datasets. The six algorithms considered in this study are Hamming, Levenshtein, LCS, Jaccard, Jaro, and Jaro - Winkler, which have been proposed for a long time. These are the basic and extensive similarity algorithms that are still used in many recent studies such as natural language processing [19–21], image processing [22], recommendation systems [23], and data clustering [24]. The algorithms are also used in recent research on fuzzy joins [9, 12, 13]. Basic algorithms are always necessary as long as they are extensive and effective in use. Thus, we present the fuzzy join calculation based on the similarity algorithms in a systematic manner and provide experiments for evaluation and comparison in a new platform of big data processing. Detailed descriptions of the six algorithms are presented along with the experiments implemented on Apache Spark. The three criteria used to evaluate these algorithms are output enrichment, false positives/negatives, and processing time. Based on the overview of the algorithms, depending on the desired fuzzy thresholds, users will choose the algorithms to suit their requirements. The arrangement of the paper is as follows. Section 2 presents the theoretical background related to the MapReduce model, Spark platform, and fuzzy join computation on a big data processing platform. Section 3 shows the six similarity algorithms that can be used in fuzzy join computation. We present the experiments in section 4 and conclude the paper in section 5.

2. BACKGROUND

In this chapter, we will describe the theoretical background of the MapReduce model, the Apache Spark platform, and the implementation of fuzzy join.

2.1. MapReduce

MapReduce [25] is a programming model providing an efficient data flow engine to improve the performance of data processing in a cluster environment. The model was developed and had been successfully used at Google for many different purposes. The motivation was to distribute computations across hundreds or thousands of machines to finish in a reasonable amount of time but hide the messy details of parallelization, fault tolerance, data distribution, and load balancing in a library. Two main functions in the programming paradigm are map and reduce. Mapping uses the input data to produce a set of intermediate key/value

pairs and reducing merges the intermediate values associating with a key to produce the results.

2.2. Apache spark

Apache Spark [26] is a large-scale data processing framework that can quickly perform task processing on big datasets, and can also distribute computation across multiple computers. Spark is compatible with multiple distributed file systems such as Hadoop HDFS[1], Cassandra[2], Hbase[3], and Amazon S3[4]. It allows the division of tasks into smaller pieces and runs in memory on different computing nodes to exploit the fast processing speed. Spark Core is the main component of Spark that supports the most basic functions such as scheduling tasks, managing memory, and error recovery. It provides concise and consistent APIs in Scala, Java, and Python.

A resilient distributed dataset (RDD) is a fundamental data structure of Spark. It is a fault-tolerant collection of elements partitioned across the nodes of the cluster that can be operated in parallel. RDDs support two types of operations: transformation and action. A transformation generates new RDDs from existing RDDs and an action returns a value after running a computation on RDDs. The RDDs will be recalculated every time a related action is performed, thus, an important capability of Spark is persisting (or caching) the RDDs in memory or on disk across operations. This is a key tool for iterative algorithms and fast interactive use. The RDDs will only calculate once and reuse in other actions. Spark uses a concept called "storage level" to manage the places where we can persist RDDs (on disk or in memory).

2.3. Fuzzy join

Fuzzy join (or set-similarity join) is a powerful operation that matches records in the input datasets with a given similarity function. Given two input datasets R and L, the fuzzy join operation will return all output records combining $x \in R$ and $y \in L$ such that $sim(x, y) >= \theta$, where sim is a similarity function and θ is a user-specified threshold. Commonly used similarity functions include Hamming, Levenshtein, Longest common sequence (LCS), Jaccard, Jaro, Jaro - Winkler, etc. Given the two input datasets R and L (as in Table 4), the fuzzy join operation with a similarity threshold of 1 provides the result set presented in Table 5.

In Table 5, the result set consists of 5 records, in which the records in line 1, line 4, and line 5 are the results of the correct match join operation. The remaining two records are the results of a fuzzy join with a threshold of 1. When performing the fuzzy join of two datasets based on the MapReduce model, the map phase will divide the input datasets of R and L for the mappers to process (Figure 1). As an example, the datasets are divided among four mappers, in which Mapper1 is assumed to process the first two records of dataset R, Mapper2 handles the remaining two records of dataset R, Mapper3 handles two records of dataset L, and Mapper4 processes a remaining record of dataset L. Then tuples with the

[1]https://hadoop.apache.org
[2]https://cassandra.apache.org
[3]https://hbase.apache.org
[4]https://aws.amazon.com/s3/

Table 4: Dataset R and L

R		L	
Join key R	Value R	Join key L	Value L
CS1_HC	1	CS1_HC	A
CS2_HC	2	CS2_HC	B
CS1_KT	3	CS1_KT	C
CS2_QL	4		

Table 5: Join results of R and L with a fuzzy threshold of 1

Join key R	Value R	Join key L	Value L
CS1_HC	1	CS1_HC	A
CS2_HC	2	CS1_HC	A
CS1_HC	1	CS2_HC	B
CS2_HC	2	CS2_HC	B
CS1_KT	3	CS1_KT	C

same join key will be sorted and shuffled to the same reducer before performing the join computation, and the results are returned after performing the join at the Reducers.

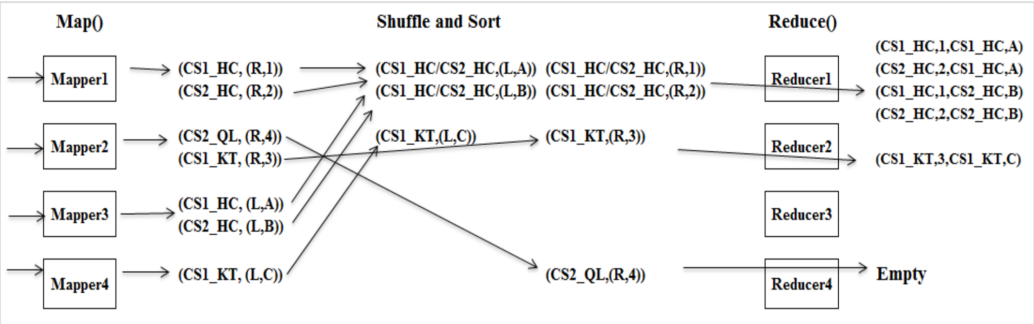


Figure 1: Fuzzy join based on MapReduce model

The implementation of Fuzzy Join in Spark is to use similarity algorithms to check the similarity between the join keys in two input datasets. Based on this similarity, the join keys are compared with a given fuzzy threshold to determine whether a join operation on the two data records can be performed. The fuzzy join process between two datasets in Spark is depicted schematically in Figure 2. The input data come from HDFS consisting of two datasets R and L. We generate two RDDs, rddDictionaryL, and rddDictionaryR corresponding with the two datasets. These RDDs contain strings of the join key field in the two input datasets (duplicates are removed). The next task is to create a listLibL of all data contained in the rddDictionaryL. The key/value pairs for R (rddFuzzyKeyR) are generated from rddDictionaryR and listLibL. Each key in the pairs will be checked for similarity with the keys in listLibL through a FuzzyGenerate function. A mapFuzzyKeyR aggregates all values of the same key from rddFuzzyKeyR. Then, we generate Lpairs and Rpairs from dataset L and mapFuzzyKeyR, respectively. Finally, the fuzzy join operation between Lpairs

and Rpairs will be done and stored results on HDFS. The difference between the similarity algorithms is only shown when generating rddFuzzyKeyR by the FuzzyGenerate function.

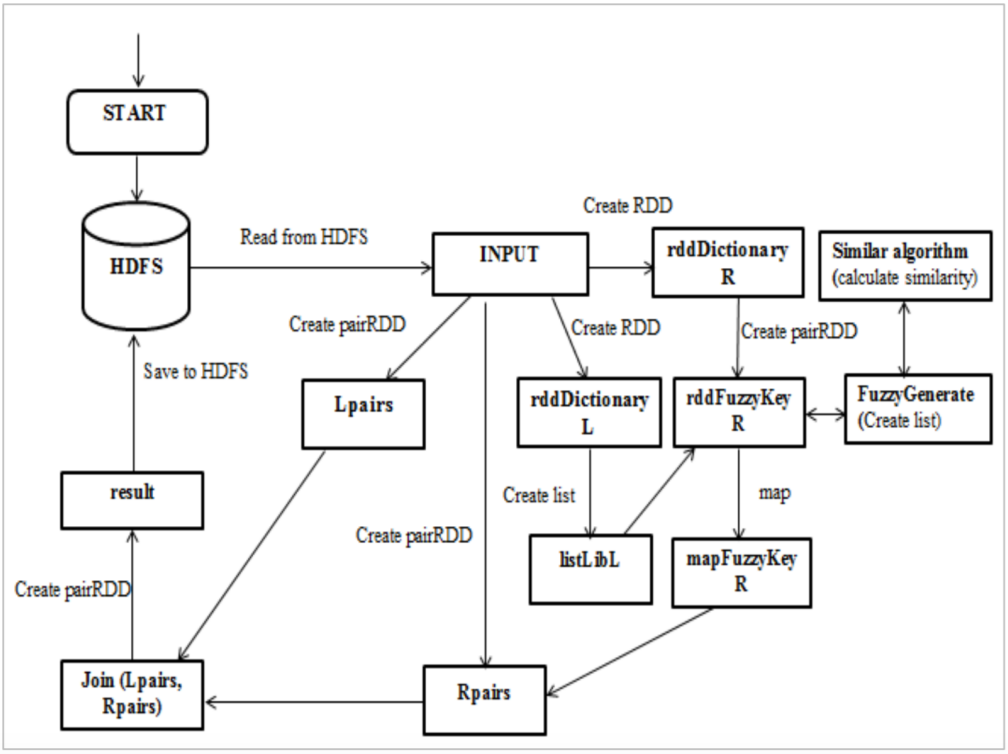


Figure 2: Fuzzy join schema in Spark

3. SIMILARITY ALGORITHMS

3.1. Hamming distance

Hamming distance [27] is named after the American mathematician Richard Hamming, who introduced the basic concept of Hamming codes, error detection, and correction codes in 1950 [28]. Hamming distance is frequently used in telecommunications to estimate faults by counting the amount of flipped bits in a binary word of a particular length. In fuzzy joins, Hamming distance has been used in several studies [3, 9, 14]. Let d be Hamming Distance between two strings of equal length, d will be the number of symbols in equivalent positions with different values of the two strings. Assuming there are two strings S_1= "010101" and S_2= "010010", we have $d = 3$ since the different symbols in the two strings are at positions 4, 5, and 6. The Hamming distance algorithm is presented in Algorithm 1.

The first thing to do is to convert the two strings to the same uppercase or lowercase and if the two strings have different lengths, they will need to be returned in the same length for comparison. Check the symbols in the corresponding positions of the two strings in turn, if the symbols are different, the value of d will be increased. In the end, the value of d received will be the Hamming Distance between the two strings. The algorithm performs two main tasks: checking the length of two strings and calculating the Hamming distance.

Algorithm 1 Hamming distance

Input: Two strings S_1 and S_2
Output: Hamming distance d

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: while |S1|! = |S2| do
4:    if (|S1| < |S2|) then
5:       S1 += ' ';
6:    else
7:       S2 += ' ';
8: int d = 0;
9: for all String i : S1 do
10:   if S1[i]! = S2[i] then
11:      d + +;
12: return d;
```

Checking the length of two strings in the worst case, i.e. one of the strings is empty, then the complexity, in this case, is $O(|S_1|)$ or $O(|S_2|)$. Calculating the Hamming distance of two strings with the same length has a complexity of $O(max(|S_1|, |S_2|))$, thus the complexity for the whole algorithm will be $O(max(|S_1|, |S_2|)$.

3.2. Levenshtein distance

The Levenshtein distance (or Edit distance) [29] is named after the Russian scientist Vladimir Levenshtein, who coined the concept in 1965. It is used in calculating the similarities and differences between two strings in the problem of constructing optimal codes capable of correcting deletions, insertions, and reversals. The distance measures how much a string need to the altered to become another string. It is used in the field of natural language processing [21, 30] and in the field of parallel fuzzy joins [9, 14]. The Levenshtein distance between two strings S_1 and S_2 is the minimum number of steps that turn S_1 into S_2 through three transformations: delete a symbol, add a symbol, and replace one symbol with another. The Levenshtein distance algorithm is presented in Algorithm 2.

Suppose n is the length of S_1 and m is the length of S_2, let $F[i][j]$ is the minimum number of transformations to make the string consist of i first symbols of S_1 $(S_1[1], S_1[2], ..., S_1[i])$ into a string consisting of j first symbols of S_2 $(S_2[1], S_2[2], ..., S_2[j])$.

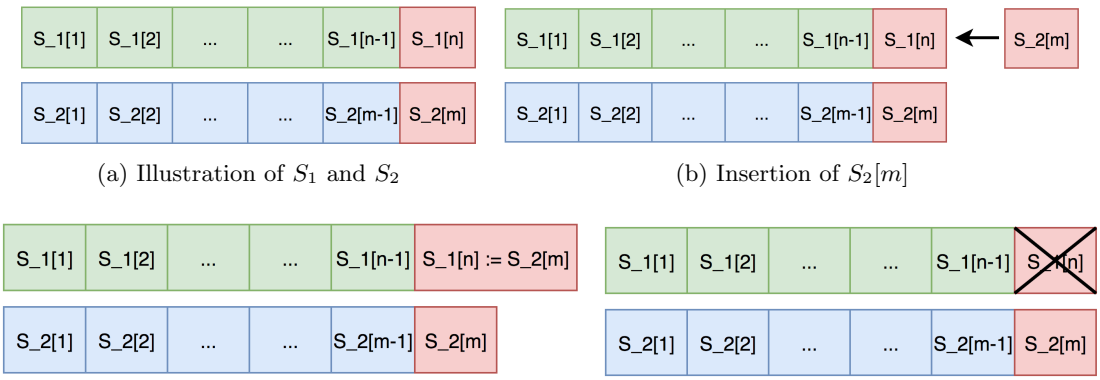
- In case $S_1[n] == S_2[m]$ (Figure 3a), we only need to convert $S_1[1], S_1[2], ..., S_1[n-1]$ into $S_2[1], S_2[2], ..., S_2[m-1]$. The value of F will be $F[n][m] = F[n-1][m-1]$.
- In case $S_1[n]! = S_2[m]$, we need to use one of the three transformations at position $S_1[n]$.
  - Insertion of $S_2[m]$ into the position after n of S_1 (Figure 3b). At this time, $F[n][m]$ will be equal to 1 (for the insertion) plus the number of transformations of the sequence $S_1[1], S_1[2], ..., S_1[n]$ into the sequence $S_2[1], S_2[2], ..., S_2[m-1]$: $F[n][m] = 1 + F[n][m-1]$.

Algorithm 2 Levenshtein distance

Input: Two strings S_1 and S_2
Output: Levenshtein distance $F[|S_1|][|S_2|]$

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: for (int i = 0; i < |S1|; i++) do
4:    F[i][0] = i;
5: for (int j = 0; j < |S2|; j++) do
6:    F[0][j] = j;
7: for (int i = 0; i < |S1|; i++) do
8:    for (int j = 0; j < |S2|; j++) do
9:       if (S1[i] == S2[i]) then
10:         F[i][j] = F[i − 1][j − 1];
11:      else
12:         F[i][j] = min(F[i − 1][j], F[i][j − 1], F[i − 1][j − 1]) + 1;
13: return F[|S1|][|S2|];
```

– Replacement of $S_1[n]$ by $S_2[m]$ (Figure 3c). At this time, $F[n][m]$ will be equal to 1 (for the replacement) plus the number of transformations of the sequence $S_1[1], S_1[2], ..., S_1[n-1]$ into the sequence $S_2[1], S_2[2], ..., S_2[m-1]$: $F[n][m] = 1 + F[n-1][m-1]$.
– Deletion of $S_1[n]$ (Figure 3d). At this time, $F[n][m]$ will be equal to 1 (for the deletion) plus the number of transformations of the sequence $S_1[1], S_1[2], ..., S_1[n-1]$ into the sequence $S_2[1], S_2[2], ..., S_2[m]$: $F[n][m] = 1 + F[n-1][m]$.
– Since $F[n][m]$ must be as small as possible, in this case, $F[n][m] = min(F[n][m-1], F[n-1][m-1], F[n-1][m]) + 1$.



(a) Illustration of S_1 and S_2
(b) Insertion of $S_2[m]$
(c) Replacement of $S_1[n]$
(d) Deletion of $S_1[n]$

Figure 3: Illustration of transformations in Levenshtein distance

The calculation of $F[i][j]$ is done by two nested for loops, thus the complexity for this task is $O(|S_1| * |S_2|)$. The complexity of the whole Levenshtein distance algorithm is $O(|S_1| * |S_2|)$.

3.3. Longest common subsequence (LCS)

The longest common subsequence (LCS) [31] of two or more strings is a useful measure of their similarity. LCS is used to calculate the similarity between texts [32], codes [33], and certain users in social media [34] to accelerate the similarity join. String S is the longest common subsequence (LCS) of strings S_1 and S_2 if S is a subsequence of S_1 and also a subsequence of S_2 of maximal length, i.e., there is no common subsequence of S_1 and S_2 that has greater length. Suppose n is the length of S_1 and m is the length of S_2, if the last two characters of the two strings are equal $S_1[n] = S_2[m]$, then that is also the last character of S. The remainder of S will be the longest common substring of $S_1[1], S_1[2], ..., S_1[n-1]$ and $S_2[1], S_2[2], ..., S_2[m-1]$. If the last two characters of S_1 and S_2 are different, then one of them does not belong to S (possibly both). Assuming $S_1[n] \notin S$, S will be the longest common subsequence of two strings $S_1[1], S_1[2], ..., S_1[n-1]$ and $S_2[1], S_2[2], ..., S_2[m]$. Conversely, if $S_2[m] \notin S$, then S will be the longest common subsequence of two strings $S_1[1], S_1[2], ..., S_1[n]$ and $S_2[1], S_2[2], ..., S_2[m-1]$. Let $F[i][j]$ be the length of the subsequence S considering the i^{th} element in S_1 and the j^{th} element in S_2, we define d as the difference between two strings S_1 and S_2 as follows:

- If $S_1[i] == S_2[j]$ then $F[i][j] = F[i-1][j-1] + 1$.
- If $S_1[i] != S_2[j]$ then $F[i][j] = \max(F[i-1][j], F[i][j-1])$.
- $d = \max(|S_1|, |S_2|) - F[n][m]$.

Algorithm 3 Longest common subsequence (LCS)

Input: Two strings S_1 and S_2
Output: Longest common subsequence (LCS) d

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: for (int i = 0; i < |S1|; i++) do
4:    F[i][0] = 0;
5: for (int j = 0; j < |S2|; j++) do
6:    F[0][j] = 0;
7: for (int i = 0; i < |S1|; i++) do
8:    for (int j = 0; j < |S2|; j++) do
9:       if (S1[i] == S2[i]) then
10:         F[i][j] = F[i-1][j-1] + 1;
11:      else
12:         F[i][j] = max(F[i-1][j], F[i][j-1]);
13: int d = max(|S1|, |S2|) - F[|S1|][|S2|];
14: return d;
```

The algorithm to calculate two strings similarity based on LCS is presented in Algorithm 3. The complexity to calculate the length of the longest common substring is $O(|S_1| * |S_2|)$ and the complexity to calculate the difference between two strings is $O(1)$. Thus, the complexity of the algorithm is $O(|S_1| * |S_2|)$.

3.4. Jaccard distance

Jaccard similarity [35] was proposed by Jaccard Paul in 1901, a French professor of Botany. This is a measure that emphasizes the similarity between finite sample sets and is defined as the size of the intersection divided by the size of the combination of the sample sets. The distance is commonly used in the problem of fuzzy joins as appearing in [6,9,11,13–15]. The Jaccard similarity is defined as in equation 1. For example, given the two strings S_1 = "$Bahamas$" and S_2 = "$Panama$", the Jaccard similarity of the two strings is 0.4 as in equation (1).

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{|S_1 \cap S_2|}{|S_1| + |S_2| - |S_1 \cap S_2|}. \tag{1}$$

Algorithm 4 Jaccard distance

Input: Two strings S_1 and S_2
Output: Jaccard similarity d

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: String str1 = new HashSet<>();
4: String str2 = new HashSet<>();
5: for (int i = 0; i < |S1|; i++) do
6:     str1+ = S1[i];
7: for (int j = 0; j < |S2|; j++) do
8:     str2+ = S2[j];
9: int mergeNum = 0;
10: int commonNum = 0;
11: for all (char ch1 : str1) do
12:     for all (char ch2 : str2) do
13:         if (ch1 == ch2) then
14:             commonNum++;
15: mergeNum = |str1| + |str2| − 2 ∗ commonNum;
16: double d = commonNum/mergeNum;
17: return d;
```

To calculate the Jaccard index, we need to find the intersection of the two strings S_1 and S_2. We use HashSets to hold the unique elements of each string. Through these HashSets, we will find the same elements of the two strings, and the union is calculated as the sum of the two strings' size minus the intersection found. The Jaccard index algorithm is presented in Algorithm 4 and the complexity for calculating the Jaccard index is $O(|S_1| * |S_2|)$.

3.5. Jaro similarity

Jaro similarity [36] was introduced in 1989 by Matthew A. Jaro, an American mathematician. This is a measure of the similarity of two strings with a value between 0 and 1, the higher the value, the more similar the strings are. If the measured value is 0, then the two strings have no similarities, and if it is 1, the two strings are exactly the same. Jaro distance has been used in the field of natural language processing [37–39]. Suppose d is the

Jaro similarity between two strings S_1 and S_2, d is calculated by equation 2, in which, m is the number of coincident characters and t is half the number of transpositions (changing the position of matching characters in two sequences). The coincident characters do not need to be in the same position in both strings, they just need to be the same and not far apart by $maxdist$ as in equation (3)

$$d = \begin{cases} 0 & \text{if} \quad m = 0 \\ \frac{1}{3}(\frac{m}{|S_1|} + \frac{m}{|S_2|} + \frac{m-t}{m}) & \text{otherwise,} \end{cases} \tag{2}$$

$$maxdist = [\frac{\max(|S_1|, |S_2|)}{2}] - 1. \tag{3}$$

Algorithm 5 Jaro similarity

Input: Two strings S_1 and S_2
Output: Jaro similarity d

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: if (S1 == S2) then
4:    return 1.0;
5: int maxdist = max(|S1|, |S2|)/2 − 1;
6: int m = 0;
7: for (int i = 0; i < |S1|; i++) do
8:    for (int j = max(0, i − maxdist); j < min(|S2|, i + maxdist + 1); i++) do
9:       if (S1[i] == S2[i] AND matchS2[j] == 0 ) then
10:         matchS1[i] = 1;
11:         matchS2[j] = 1;
12:         m++;
13:         break;
14: if (m == 0) then
15:    return 0.0;
16: int t = 0;
17: int point = 0;
18: for (int i = 0; i < |S1|; i++) do
19:    if (hashS1[i] == 1) then
20:       while (matchS2[point] == 0) do
21:          point++;
22:       if (S1[i]! = S2[point++]) then
23:          t++;
24: t/ = 2;
25: double d = ((m/|S1|) + (m/|S2|) + ((m − t)/m))/3.0;
26: return d;
```

To calculate d, we need to count the number of coincident characters of the two strings and calculate $maxdist$, which is the maximum allowed distance between the coincident characters in the two strings. The two lists $matchS_1$ and $matchS_2$ are used to mark coincident

characters in two strings, if $matchS_1[i] == 1$, then the i^{th} character in S_1 is the coincident character and if $matchS_1[i] == 0$, then the i^{th} character is not a coincident character. To mark the coincident characters on the lists, we check the match between the characters in S_1 with the characters in S_2 such that the distance between them does not exceed $maxdist$. During the marking process, we will count the number of coincident characters between the two strings. The counting of transpositions is done by checking the lists and counting the number of coincident characters in different positions in S_1 and S_2. Then, t will be half of the number of characters found. The Jaro similarity algorithm is presented in algorithm 5. In the algorithm, counting the number of coincident characters in two strings (in the worst case) has a complexity of $O(|S_1| * |S_2|)$, and counting the number of moves also falls in the range $O(|S_1| * |S_2|)$. Thus, the complexity of the Jaro similarity algorithm is $O(|S_1| * |S_2|)$.

3.6. Jaro - Winkler similarity

Jaro - Winkler similarity [40] is a modification of Jaro similarity proposed by William E. Winkler in 1990. The metric takes the Jaro similarity and increases the score if the characters at the start of both strings are the same. This modification helps Jaro-Winkler to be widely used in the field of string similarity search over its predecessor. The use of the Jaro-Winkler similarity search has been shown in [41–44]. Jaro Winkler similarity is defined as in equation 4, in which, d_j is the Jaro similarity, p is the scaling factor (0.1 by default), and l is the length of the common prefix at the start of the string (max of 4 characters).

$$d = d_j + l * p * (1 - d_j). \tag{4}$$

Algorithm 6 Jaro - Winkler similarity

Input: Two strings S_1 and S_2
Output: Jaro - Winkler similarity d

```
1: S1.toUpperCase();
2: S2.toUpperCase();
3: double p = 0.1;
4: double dj = JaroSimilarity(S1, S2);
5: double d = dj;
6: if (dj > 0.7) then
7:    int l = 0;
8:    for (int i = 0; i < min(|S1|, |S2|); i++) do
9:       if (S1[i] == S2[i]) then
10:         l++;
11:      else
12:         break;
13:   l = min(l, 4);
14:   d = dj + l * p * (1 − dj);
15: return d;
```

Jaro - Winkler similarity is only effective when the Jaro similarity (d_j) reaches a certain threshold. Based on the value of d_j, we will find the common prefix length at the beginning of each string. In the algorithm, the Jaro similarity calculation has a complexity of $O(|S_1|*|S_2|)$ while the Jaro - Winkler similarity calculation has a complexity of $O(\min(|S_1|, |S_2|))$. Thus, the complexity for the whole Jaro - Winkler similarity algorithm is $O(|S_1| * |S_2|)$.

4. EXPERIMENTS

4.1. Installation environment and data description

The experiments are conducted on a cluster of 9 computers (1 master and 8 computing nodes). Each computer uses Ubuntu 20.04 LTS operating system and is configured with 4 CPUs, 32GB RAM, and 70GB HDD. The environment is installed with the following software: Java 1.8, Hadoop 3.2.2, and Spark 3.2.0. Spark is configured to run in master mode with 8 executors, in which each executor has 3 CPUs and 30GB RAM. HDFS is configured to store input and output data.

The datasets used to run the experiments are standard data generated by the Purdue MapReduce Benchmarks Suite [45]. The datasets are all stored as plain text files, each line has more than one field separated by commas. The join key is the first column of both datasets. The number of records and the size of the datasets are described in Table 6.

Table 6: Experimental datasets

Test	Dataset L		Dataset R	
	Size	Num_record	Size	Num_record
1	512MB	1,341,628	512MB	1,341,283
2	2GB	5,366,662	1GB	2,681,966
3	3GB	8,049,298	2GB	5,364,698
4	3GB	8,049,298	4GB	10,730,763
5	6GB	16,098,601	4GB	10,730,763

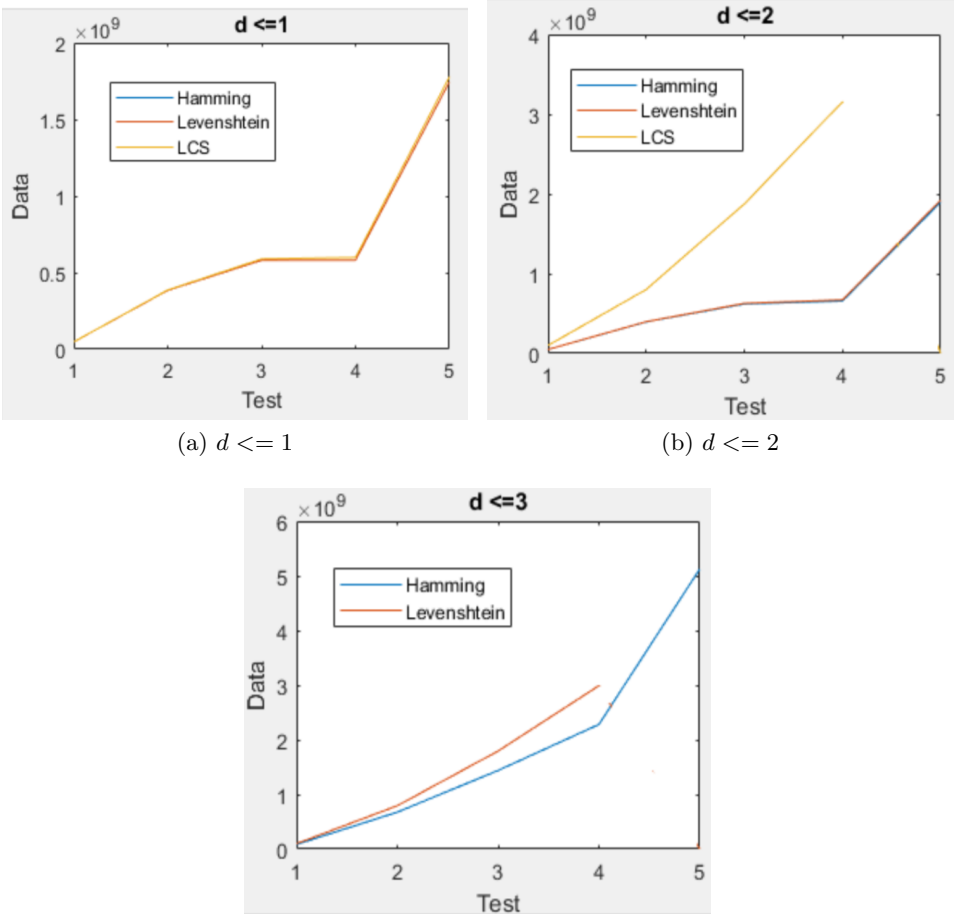
Experiments are conducted with six algorithms including Hamming, Levenshtein, LCS, Jaccard, Jaro, and Jaro - Winkler. At first, we join two datasets in the experiments with a distance of zero and the result is the same number of output records in all algorithms. This is done to ensure the correctness of the installed algorithms.

4.2. Results

First of all, in terms of the complexity of algorithms, Hamming's algorithm has the lowest complexity ($O(max(|S_1|, |S_2|))$) compared to the remaining algorithms ($O(|S_1| * |S_2|)$). However, when performing joins between large datasets in a distributed computing environment, the number of string pairs is very large, thus relying on the complexity of algorithms for performing fuzzy joins is not enough. Besides, since the algorithms are performed on different definitions of the similarity distance between two strings, the determination of a join will depend on the fuzzy threshold. Therefore, we will evaluate similarity algorithms for fuzzy joins based on the following criteria: output enrichment, number of false positives and false negatives generated in the result sets, and processing time of each algorithm. In this section, we will divide the algorithms into two groups to facilitate comparison. Group 1 includes Hamming, Levenshtein, and LCS (fuzzy threshold is 0 for exact match). Group 2 includes Jaccard, Jaro, and Jaro - Winkler (fuzzy threshold is 1 for exact match).

4.2.1. Output enrichment

The evaluation of the output enrichment is based on the number of output records generated in the result set.



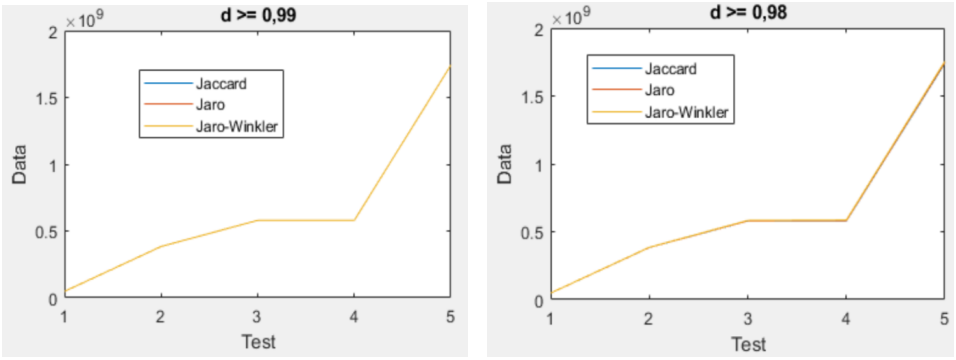
(a) $d <= 1$ (b) $d <= 2$

(c) $d <= 3$

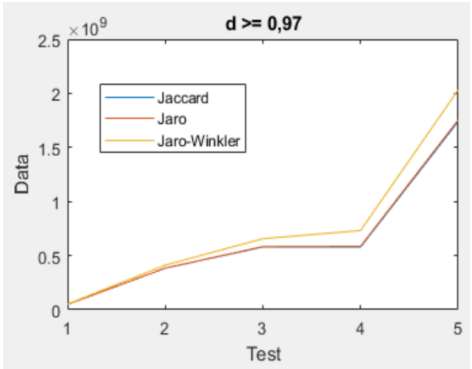
Figure 4: Number of output records for Hamming, Levenshtein, and LCS

Figure 4 presents the number of output records for algorithms in group 1 with the fuzzy threshold of 1, 2, and 3. With the fuzzy threshold of 1 (Figure 4a), the three algorithms in group 1 provide almost the same number of output records. Specifically, the number of records in the result set of Hamming and Levenshtein algorithms is the same in all testing datasets. Meanwhile, the number of records in the result set generated by the LCS algorithm is greater than those of the other two algorithms of the same group (Test 1: 320,305 records greater, Test 2: 2,589,272 records greater, Test 3: 500,763 records greater, Test 4: 8,444,040 records greater, and Test 5: 3,358,843 records greater). With the fuzzy threshold of 2 (Figure 4b), the number of output records increases quite a lot compared to the join result with a fuzzy threshold of 1. Output enrichment between the algorithms is shown more clearly, especially with the LCS algorithm. The number of output records in both Hamming and Levenshtein algorithms is almost equal, in which the Levenshtein algorithm has a number of output records slightly greater than that of the Hamming algorithm. The

algorithm that generates the greatest result set is the LCS algorithm and the larger the input datasets, the more records are generated in the result set (Test 3: more than Hamming 1,246,598,801 records and more than Levenshtein 1,253,795,848 records; Test 4, more than Hamming 2,497.614,463 records and more than Levenshtein 2,483,669,378 records). However, when running the LCS algorithm with Test 5, the LCS has a memory overflow because too much data is generated. Thus, in terms of output enrichment, LCS gives the best results with a fuzzy threshold of 2. With the fuzzy threshold of 3 (Figure 4c), the join computation can only be done on two algorithms, Hamming and Levenshtein. In all Test cases, the number of records obtained in the result set of the Levenshtein algorithm is larger than that of the Hamming algorithm. The difference is more pronounced when running with large input datasets (Test 3: more than Hamming 355,399,953 records, and Test 4: more than Hamming 709,149,035 records). However, with Test 5, the Levenshtein algorithm failed due to memory overflow. Thus, with the fuzzy threshold of 1, 2, and 3, the LCS algorithm has the best output enrichment performance, followed by the Levenshtein algorithm and finally the Hamming algorithm.



(a) $d >= 0.99$ (b) $d >= 0.98$



(c) $d >= 0.97$

Figure 5: Number of output records for Jaccard, Jaro, and Jaro - Winkler

Figure 5 presents the number of output records for algorithms in group 2 (Jaccard, Jaro, and Jaro - Winkler) with the fuzzy threshold is 0.99, 0.98, and 0.97. As shown in Figure 5a, with a fuzzy threshold of 0.99, the data received in the result set of the three algorithms in group 2 are equal. With a fuzzy threshold of 0.98 (Figure 5b), the number of output records

of the three algorithms has a difference but is not significant. With a fuzzy threshold of 0.97 (Figure 5c), the output enrichment is shown more clearly. However, the result set of Jaccard remains unchanged compared to the join computation with a fuzzy threshold of 0.98 and 0.99. The Jaro - Winkler algorithm has a clear difference compared to the other two algorithms, which is evident in Test 4 of 145,930,361 records more than Jaro and 151,801,312 records more than Jaccard and Test 5 of 278,269,366 records more than Jaro and 290,981,928 records more than Jaccard. Thus, in the second group of algorithms, the Jaro - Winkler algorithm has the best output enrichment performance, followed by the Jaro algorithm and finally the Jaccard algorithm.

4.2.2. False positives and false negatives

False positive is an incorrect record in the result set that is generated from the join operation between two rows in the two input datasets. In contrast, false negatives are two lines in the input data set that can produce a join result but are not included in the result set (missing correct data). Thus, to check the number of false positives and false negatives generated by the algorithms, we run experiments on small datasets to facilitate checking the results and comparing.

Table 7: Test 6

Dataset R	Dataset L
150 North Michigan Ave	150 North Michigan Ave
605 3rd Acenue	605 Third Avenue
300 North Meridian Street	300 North Meridian Street
400 High St SE	400 High Street S.E
2203 Rowan Street	2203 Rowen St

Table 7 presents the two datasets R and L in Test 6. As can be seen, the best result set of the fuzzy join computation should include 5 records. To receive this result (without false positive and false negative), we need different fuzzy thresholds in the six algorithms (Table 8). The fuzzy threshold will be set to be the same for two groups of algorithms to compare the number of false positives and false negatives received. With group 1 of Hamming, Levenshtein, and LCS, the fuzzy threshold is 5 and the fuzzy threshold will be 0.91 for group 2. At this moment, experimental results show that when the algorithms run on the same fuzzy threshold, there are differences in the result set. Specifically, the Hamming, Jaccard, and Jaro algorithms have 2 false negatives in the result set and no false positives. These algorithms have not met the desired result set requirements. The remaining algorithms return a result set that completely matches the requirement and does not return false negatives/positives. That means that Levenshtein, LCS, and Jaro - Winkler return a better result set when joining datasets on the same fuzzy threshold (compared to the same group of algorithms).

We conduct another experiment on Test 7 (Table 9). At this time, the fuzzy threshold between the corresponding lines in the two datasets R and L are also presented. In Table 9, the highlighted rows are the correct join results while the remaining rows are false positives. For each pair in the two datasets R and L, the similarity distance is calculated with the six algorithms. Determining the fuzzy threshold for these algorithms is difficult since the data

Table 8: Results of join computation in Test 6

Algorithms	Num_records	False positives	False negative	Fuzzy threshold
Hamming	5	0	0	<=12
Levenshtein	5	0	0	<=5
LCS	5	0	0	<=5
Jaccard	5	0	0	>=0.625
Jaro	5	0	0	>=0.8574
Jaro - Winkler	5	0	0	>=0.9144
Hamming	3	0	2	<=5
Levenshtein	5	0	0	<=5
LCS	5	0	0	<=5
Jaccard	3	0	2	>=0.91
Jaro	3	0	2	>=0.91
Jaro - Winkler	5	0	0	>=0.91

in the two datasets have a huge difference. For example, with Hamming algorithm, if the fuzzy threshold is 28, it will ensure to provide all expected results, but will generate the false positives of 8; and if the fuzzy threshold is 11, it will produce 4 false positives and 2 false negatives.

Table 9: Test 7

Records in R	Records in L	Fuzzy threshold					
		Hamming	Levenshtein	LCS	Jaccard	Jaro	Jaro - Winkler
Antigua and	Antigua &	11	3	3	0.9090	0.8444	0.9066
Bahamas	Panama	3	3	3	0.2857	0.7460	0.7460
Bosnia and Herzegovi na	Bosnia & Herzegov ina	15	3	3	0.8666	0.8413	0.9048
Cabo Verde	Cape Verde	2	2	2	0.7	0.7416	0.7933
Central African Republic	Central African Rep.	5	5	5	0.7857	0.9138	0.9483
China including Hong Kong	Hong Kong	22	16	16	0.5	0.5244	0.5244
Congo	Tonga	2	2	2	0.5	0.7333	0.7333
Cook Islands (NZ)	Cook Islands	4	5	5	0.7692	0.9019	0.9411
Côte d'Ivoire	Cote d'Ivoire	1	1	1	0.9090	0.9487	0.9538
Federated States of Micronesia	United States	28	20	20	0.5714	0.5675	0.5675
Gambia	Zambia	1	1	1	0.6666	0.8888	0.8888
Myanmar/ Burma	Panama	11	8	8	0.3333	0.5747	0.5747
North Macedonia	New Caledonia	14	6	6	0.5714	0.7641	0.7876
Saint Kitts and Nevis	Saint Kitts & Nevis	9	3	3	0.8181	0.9348	0.9609
South Sudan	South Korea	5	5	4	0.5833	0.7575	0.8545
São Tomé and Príncipe	Sao Tome & Principe	14	6	6	0.7058	0.7325	0.7593
Timor- Leste	Sierra Leone	10	7	7	0.5	0.5237	0.5237
Trinidad and Tobago	Trinidad & Tobago	10	3	3	0.9090	0.8756	0.9254
Vatican City State	Vatican	9	11	11	0.6	0.7962	0.8777

An experiment is conducted on Test 7 with the selection of the fuzzy threshold for each algorithm so that the number of false negatives generated in the result set is zero. The results of the experiment are shown in Table 10. The Hamming and Jaccard algorithms have a high number of false positives in the result set and a high fuzzy threshold (false positives is 6, a fuzzy threshold of 15 for Hamming and 0.5 for Jaccard). The remaining algorithms have lower fuzzy thresholds and fewer false positives in the result set. Thus, in considering false positives/negatives in the result set, the Hamming and Jaccard algorithms are somewhat less advantageous than the remaining algorithms.

Table 10: Results of join computation in Test 7

Algorithms	Num_records	False positives	False negative	Fuzzy threshold
Hamming	17	6	0	<=15
Levenshtein	15	4	0	<=6
LCS	15	4	0	<=6
Jaccard	17	6	0	>=0.5
Jaro	15	4	0	>=0.7325
Jaro - Winkler	15	4	0	>=0.7333

4.2.3. Processing time

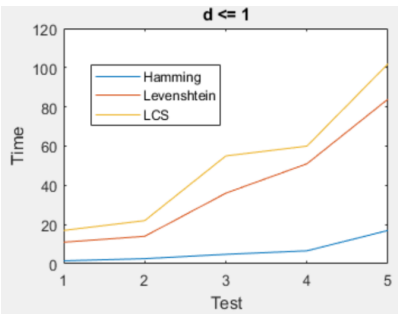
Figure 6 shows the processing time of the join computation based on the first group of algorithms (Hamming, Levenshtein, and LCS). The join computation is done with the fuzzy threshold of 1, 2, and 3. The processing time is proportional to the size of the two input datasets, and the larger the input datasets, the higher the processing time difference of the algorithms. With a fuzzy threshold of 1, the Hamming algorithm on Test 1 needs to run in 1.5 minutes while the Levenshtein algorithm executes in 11 minutes and the LCS algorithm runs in 22 minutes. With Test 5, the Hamming algorithm executes in 17 minutes while the Levenshtein algorithm runs in 84 minutes and the LCS algorithm runs in 102 minutes. The Levenshtein algorithm is more than 5 times slower than Hamming and the LCS algorithm is more than 6 times slower than Hamming. The similarity algorithms for join computation with a fuzzy threshold of 2 tend to be the same as with a fuzzy threshold of 1, in which LCS continues to be the algorithm with the longest processing time. With a fuzzy threshold of 3, the Hamming algorithm continues to give a much better running time than the Levenshtein algorithm. Thus, in all Test cases and with 3 fuzzy thresholds, Hamming has the least running time, followed by Levenshtein and finally LCS.

Figure 7 shows the processing time of the join computation based on the second group of algorithms (Jaccard, Jaro, and Jaro - Winkler). The join computation is done with the fuzzy threshold of 0.99, 0.98, and 0.97. The processing time of the algorithms with fuzzy thresholds of 0.99, 0.98, and 0.97 does not change significantly. Jaccard has a very slow processing time in all Test cases, about 4 times slower than the other two algorithms. With all three fuzzy thresholds, the processing time of this algorithm is almost unchanged. The Jaro algorithm has the least processing time, followed by the Jaro -Winkler algorithm. With a fuzzy threshold of 0.99, the processing time of the two algorithms is almost equal. With the fuzzy thresholds of 0.98 and 0.97, the difference between the processing time of the two algorithms is not significant. Thus, in the second group of algorithms, the two algorithms Jaro and Jaro - Winkler have the best processing time and the Jaccard algorithm has the worst running time.

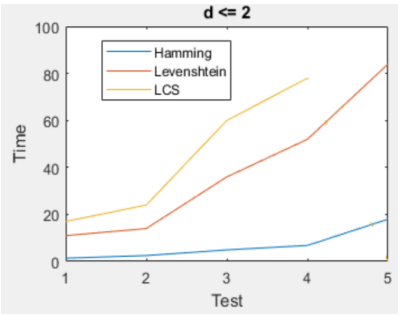
4.2.4. Comparison

A brief comparison of the algorithms is presented in Table 11. This comparison depends on three criteria: output enrichment, false positives/negatives, and processing time.

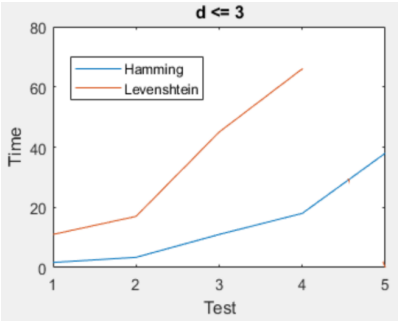
Among the algorithms belonging to the first group, the Levenshtein algorithm has an advantage over the other two algorithms. This algorithm has high accuracy in the result



(a) $d <= 1$



(b) $d <= 2$



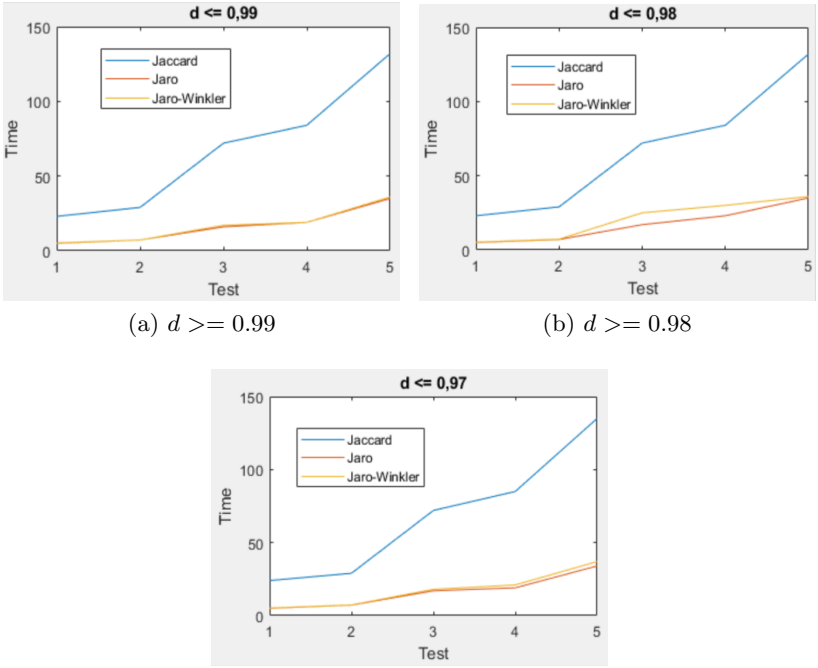
(c) $d <= 3$

Figure 6: Processing time (minutes) of Hamming, Levenshtein, and LCS

Table 11: Comparison of the six algorithms

Algorithms	False positives/negatives	Output enrichment	Processing time
Hamming	Much	Little	Fast
Levenshtein	Little	Average	Acceptable
LCS	Little	Much	Slow
Jaccard	Much	Little	Very slow
Jaro	Little	Average	Pretty fast
Jaro - Winkler	Little (Less than Jaro)	Average (more than Jaro)	Pretty fast (slower than Jaro)

set, runs with large input datasets, and has a good output enrichment performance and an acceptable processing time. The Hamming algorithm has a fast processing time and can run with large input datasets, but it has limited output richness and a lack of accuracy in the result set. The LCS algorithm has high accuracy in the result set and a good output enrichment performance, but the processing time is very slow and it can work with small input datasets. In group 2, the Jaccard algorithm is considered the worst algorithm in all aspects. It has a very slow processing time and limited output richness on the same fuzzy thresholds. The result set of this algorithm has not had high accuracy. The Jaro algorithm has a faster running time and the Jaro - Winkler algorithm has more output richness and higher accuracy. However, the difference between these two algorithms is not much. Thus, considering all three criteria, in group 1, the priority order of algorithms is arranged as follows: Levenshtein algorithm, followed by Hamming algorithm, and finally LCS algorithm.



(a) $d >= 0.99$ (b) $d >= 0.98$

(c) $d >= 0.97$

Figure 7: Processing time (minutes) of Jaccard, Jaro, and Jaro - Winkler

Similarly in the second group, the priority order of algorithms are arranged as follows: the Jaro - Winkler algorithm, followed by the Jaro algorithm, and finally the Jaccard algorithm. Depending on each criterion, users will choose a suitable algorithm for fuzzy join on large datasets.

5. CONCLUSION

In this study, we present an overview of similarity algorithms that can be used in fuzzy join operations (Hamming, Levenshtein, LCS, Jaccard, Jaro, and Jaro - Winkler). These algorithms are used to check the similarity between two strings so that comparing the join keys will yield results within the desired fuzzy threshold. We implement the experiments on the Apache Spark platform and use the Puma benchmark datasets in fuzzy join computation. The evaluation of algorithms is based on several criteria: output enrichment, false positives/negatives, and the processing time of the algorithms. Based on these criteria, Levenshtein and Jaro - Winkler are two algorithms that can be prioritized over the rest. Both two algorithms work with large input datasets, have higher accuracy in the result set, have a short processing time, and have a relative output richness. Depending on the desired fuzzy thresholds, users will choose the algorithms to suit their requirements. The study described similarity algorithms for fuzzy join and presented experiments installed in the Spark environment. We give three criteria to evaluate algorithms to help users choose the suitable algorithm when performing fuzzy join on large datasets. The performance of similarity between the two strings in this study considers the character sequence. It can be extended by looking at the word sequence, and even at the semantic level.

REFERENCES

[1] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 743–754.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 5–5.

[3] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proceedings of The 32nd International Conference on Very Large Data Dases*, 2006, pp. 918–929.

[4] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of The 16th International Conference on World Wide Web*, 2007, pp. 131–140.

[5] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient similarity joins for near-duplicate detection," *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 3, pp. 1–41, 2011.

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

[7] R. Baraglia, G. D. F. Morales, and C. Lucchese, "Document similarity self-join with mapreduce," in *2010 IEEE International Conference on Data Mining*. IEEE, 2010, pp. 731–736.

[8] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with mapreduce," in *Proceedings of ACL-08: HLT, short papers*, 2008, pp. 265–268.

[9] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman, "Fuzzy joins using mapreduce," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 498–509.

[10] B. Kimmett, V. Srinivasan, and A. Thomo, "Fuzzy joins in mapreduce: an experimental study," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1514–1517, 2015.

[11] B. Kimmett, A. Thomo, and V. Srinivasan, "Fuzzy joins in mapreduce: Edit and jaccard distance," in *2016 7th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE, 2016, pp. 1–6.

[12] A. Das Sarma, Y. He, and S. Chaudhuri, "Clusterjoin: A similarity joins framework using mapreduce," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1059–1070, 2014.

[13] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng, "Massjoin: A mapreduce-based method for scalable string similarity joins," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 340–351.

[14] M. Yu, G. Li, D. Deng, and J. Feng, "String similarity search and join: a survey," *Frontiers of Computer Science*, vol. 10, pp. 399–417, 2016.

[15] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag, "Set similarity joins on mapreduce: An experimental survey," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1110–1122, 2018.

[16] C. Yan, X. Zhao, Q. Zhang, and Y. Huang, "Efficient string similarity join in multi-core and distributed systems," *PloS One*, vol. 12, no. 3, p. e0172526, 2017.

[17] Z. Chen, Y. Wang, V. Narasayya, and S. Chaudhuri, "Customizable and scalable fuzzy join for big data," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2106–2117, 2019.

[18] T. Thi-To-Quyen, P. Thuong-Cang, A. Laurent, and L. d'Orazio, "Optimization for large-scale fuzzy joins using fuzzy filters in mapreduce," in *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE, 2020, pp. 1–8.

[19] S. Jimenez, C. J. Becerra, A. F. Gelbukh, and F. A. González, "Generalized mongue-elkan method for approximate text string comparison." in *CICLing*, vol. 9. Springer, 2009, pp. 559–570.

[20] J. M. Keil, "Efficient bounded jaro-winkler similarity based search," *BTW 2019*, 2019.

[21] E. Anzén, "The viability of machine learning models based on levenstein distance and cosine similarity for plagiarism detection in digital exams," 2018.

[22] M. Jain, R. Benmokhtar, H. Jégou, and P. Gros, "Hamming embedding similarity-based image classification," in *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, 2012, pp. 1–8.

[23] S. Bag, S. K. Kumar, and M. K. Tiwari, "An efficient recommendation generation using relevant jaccard similarity," *Information Sciences*, vol. 483, pp. 53–64, 2019.

[24] D. B. Bisandu, R. Prasad, and M. M. Liman, "Data clustering using efficient similarity measures," *Journal of Statistics and Management Systems*, vol. 22, no. 5, pp. 901–922, 2019.

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

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[27] A. Sgarro, "A fuzzy hamming distance," *Bulletin mathématique de la Société des Sciences Mathématiques de la République Socialiste de Roumanie*, pp. 137–144, 1977.

[28] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[29] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics Doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.

[30] C. Arora, M. Sabetzadeh, A. Goknil, L. C. Briand, and F. Zimmer, "Change impact analysis for natural language requirements: An NLP approach," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*. IEEE, 2015, pp. 6–15.

[31] D. Mount, "CMSC 251: Algorithms1 spring 1998," University Lectures, 1998.

[32] C.-S. Tasi, Y.-M. Huang, C.-H. Liu, and Y.-M. Huang, "Applying VSM and LCS to develop an integrated text retrieval mechanism," *Expert Systems with Applications*, vol. 39, no. 4, pp. 3974–3982, 2012.

[33] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014, pp. 305–314.

[34] X. Zhou and L. Chen, "Event detection over twitter social media streams," *The VLDB Journal*, vol. 23, no. 3, pp. 381–400, 2014.

[35] P. Jaccard, "Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 241–272, 1901.

[36] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *Journal of the American Statistical Association*, vol. 84, no. 406, pp. 414–420, 1989.

[37] C. Varol and S. Hari, "Detecting near-duplicate text documents with a hybrid approach," *Journal of Information Science*, vol. 41, no. 4, pp. 405–414, 2015.

[38] R. Aziz and M. W. Anwar, "URDU spell checker: A scarce resource language," in *Intelligent Technologies and Applications: Second International Conference, INTAP 2019, Bahawalpur, Pakistan, November 6–8, 2019, Revised Selected Papers 2*. Springer, 2020, pp. 471–483.

[39] V. Wijaya, A. Erwin, M. Galinium, and W. Muliady, "Automatic mood classification of indonesian tweets using linguistic approach," in *2013 International Conference on Information Technology and Electrical Engineering (ICITEE)*. IEEE, 2013, pp. 41–46.

[40] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage," ERIC, Institute of Education Sciences of the United States Department of Education, Tech. Rep., 1990.

[41] Y. Wang, J. Qin, and W. Wang, "Efficient approximate entity matching using jaro-winkler distance," in *Web Information Systems Engineering–WISE 2017: 18th International Conference, Puschino, Russia, October 7-11, 2017, Proceedings, Part I*. Springer, 2017, pp. 231–239.

[42] K. Dreßler and A.-C. N. Ngomo, "Time-efficient execution of bounded jaro-winkler distances." in *OM*, 2014, pp. 37–48.

[43] W. W. Cohen, P. Ravikumar, S. E. Fienberg *et al.*, "A comparison of string distance metrics for name-matching tasks." in *IIWeb*, vol. 3, 2003, pp. 73–78.

[44] I. E. Agbehadji, H. Yang, S. Fong, and R. Millham, "The comparative analysis of smith-waterman algorithm with jaro-winkler algorithm for the detection of duplicate health related records," in *2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*. IEEE, 2018, pp. 1–10.

[45] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," Electrical and Computer Engineering, Purdue University, Tech. Rep., 2012.

Received on October 16, 2022
Accepted on June 01, 2023