SIMILARITY ALGORITHMS FOR FUZZY JOIN COMPUTATION IN BIG DATA PROCESSING ENVIRONMENT

ANH CANG PHAN\textsuperscript{1,\ast}, THUONG CANG PHAN\textsuperscript{2}

\textsuperscript{1}Vinh Long University of Technology and Education, Viet Nam
\textsuperscript{2}Can Tho University, 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 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

\textsuperscript{\ast}Corresponding author.

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

© 2023 Vietnam Academy of Science & Technology
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 \( R \bowtie_{\text{Ent_Add}=\text{Emp_Add}} L \) will return a new dataset \( O \) as shown in Table 3.

### Table 1: Dataset \( R(Enterprise) \)

<table>
<thead>
<tr>
<th>Ent_Num</th>
<th>Ent_Name</th>
<th>Ent_Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>11663</td>
<td>More Power Industries</td>
<td>150 North Michigan Ave</td>
</tr>
<tr>
<td>11435</td>
<td>Group Services</td>
<td>605 3rd Avenue</td>
</tr>
<tr>
<td>13928</td>
<td>Liberty Trading</td>
<td>300 North Meridian Street</td>
</tr>
</tbody>
</table>

### Table 2: Dataset \( L(Employee) \)

<table>
<thead>
<tr>
<th>Emp_Num</th>
<th>Emp_Name</th>
<th>Emp_Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Catherine Exelby</td>
<td>150 North Michigan Ave</td>
</tr>
<tr>
<td>22</td>
<td>Eileen Henderson</td>
<td>605 Third Avenue</td>
</tr>
<tr>
<td>65</td>
<td>Jorge Marin</td>
<td>300 North Meridian Street</td>
</tr>
</tbody>
</table>

### Table 3: Dataset \( O = R \bowtie_{\text{Ent_Add}=\text{Emp_Add}} L \)

<table>
<thead>
<tr>
<th>Ent_Num</th>
<th>Ent_Name</th>
<th>Ent_Add</th>
<th>Emp_Num</th>
<th>Emp_Name</th>
<th>Emp_Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>11663</td>
<td>More Power Industries</td>
<td>150 North Michigan Ave</td>
<td>10</td>
<td>Catherine Exelby</td>
<td>150 North Michigan Ave</td>
</tr>
<tr>
<td>13928</td>
<td>Liberty Trading</td>
<td>300 North Meridian Street</td>
<td>65</td>
<td>Jorge Marin</td>
<td>300 North Meridian Street</td>
</tr>
</tbody>
</table>

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 [5,10,12,32,41]. Finding pairs of strings with a specific amount of tokens in common was done using inverted-list-based algorithms proposed in [32]. In [12], a prefix filter was proposed to reduce the large number of pairs in data cleaning. In [5,10], the length filter has been examined. The positional filter and the suffix filter were two additional filters that were suggested in [41]. Concerning parallel fuzzy joins, there have been some studies using the popular MapReduce model. Vernica et al. [37] 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. [9] 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. [19]. The authors ran experiments on a 5-node cluster with Hadoop con-
figuration showing their improvements over the approach in [37]. Afrati et al. [1] 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 re-
ducers, 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 the-
oretical analysis, but no experimental evaluation is provided, thus Ben Kimmett et al. [28]
computed fuzzy joins of binary strings using Hamming Distance from the MapReduce algo-
rithms proposed in [1]. 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 [29], Ben Kimmett et al. continued to provide an experimental eval-
uation of the fuzzy join algorithms proposed in [1] 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 [28,29], the authors used the Hadoop
platform for their experiments.

Das Sarma et al. [15] 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. [17] 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. [43] 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. [12] 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. [20] 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. [42] 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. [13] 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. [35] 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 [4, 25, 27], image processing [23], recommendation systems [8], and data clustering [11]. The algorithms are also used in recent research on fuzzy joins [1, 15, 17]. 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 [16] 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 [44] 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\textsuperscript{1}, Cassandra\textsuperscript{2}, Hbase\textsuperscript{3}, and Amazon S3\textsuperscript{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 $\text{sim}(x, y) \geq \theta$, where $\text{sim}$ is a similarity function and $\theta$ 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.

<table>
<thead>
<tr>
<th>$R$</th>
<th>$L$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Join key</strong> $R$</td>
<td><strong>Value</strong> $R$</td>
</tr>
<tr>
<td>CS1_HC</td>
<td>1</td>
</tr>
<tr>
<td>CS2_HC</td>
<td>2</td>
</tr>
<tr>
<td>CS1_KT</td>
<td>3</td>
</tr>
<tr>
<td>CS2_QL</td>
<td>4</td>
</tr>
</tbody>
</table>

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

\textsuperscript{1}https://hadoop.apache.org
\textsuperscript{2}https://cassandra.apache.org
\textsuperscript{3}https://hbase.apache.org
\textsuperscript{4}https://aws.amazon.com/s3/
Table 5: Join results of $R$ and $L$ with a fuzzy threshold of 1

<table>
<thead>
<tr>
<th>Join key $R$</th>
<th>Value $R$</th>
<th>Join key $L$</th>
<th>Value $L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1_HC</td>
<td>1</td>
<td>CS1_HC</td>
<td>A</td>
</tr>
<tr>
<td>CS2_HC</td>
<td>2</td>
<td>CS1_HC</td>
<td>A</td>
</tr>
<tr>
<td>CS1_HC</td>
<td>1</td>
<td>CS2_HC</td>
<td>B</td>
</tr>
<tr>
<td>CS2_HC</td>
<td>2</td>
<td>CS2_HC</td>
<td>B</td>
</tr>
<tr>
<td>CS1_KT</td>
<td>3</td>
<td>CS1_KT</td>
<td>C</td>
</tr>
</tbody>
</table>

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 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](image)

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.
3. SIMILARITY ALGORITHMS

3.1. Hamming distance

Hamming distance [33] is named after the American mathematician Richard Hamming, who introduced the basic concept of Hamming codes, error detection, and correction codes in 1950 [21]. 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 [1, 5, 43]. 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. 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 \( O(max(|S_1|, |S_2|)) \).
Algorithm 1 Hamming distance

**Input:** Two strings $S_1$ and $S_2$

**Output:** Hamming distance $d$

1. $S_1.toUpperCase();$
2. $S_2.toUpperCase();$
3. **while** $|S_1| = |S_2|$ **do**
   4. **if** ($|S_1| < |S_2|$) **then**
      5. $S_1 += ' ';$
   6. **else**
      7. $S_2 += ' ';$
8. **int** $d = 0;$
9. **for** **all** String $i : S_1$ **do**
10. **if** $S_1[i]! = S_2[i]$ **then**
11. $d += 1;$
12. **return** $d$;

3.2. Levenshtein distance

The Levenshtein distance (or Edit distance) [30] 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 [4,6] and in the field of parallel fuzzy joins [1,43]. 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.

Algorithm 2 Levenshtein distance

**Input:** Two strings $S_1$ and $S_2$

**Output:** Levenshtein distance $F[|S_1|][|S_2|]$

1. $S_1.toUpperCase();$
2. $S_2.toUpperCase();$
3. **for** (int $i = 0; i < |S_1|; i + +$) **do**
4. $F[i][0] = i;$
5. **for** (int $j = 0; j < |S_2|; j + +$) **do**
6. $F[0][j] = j;$
7. **for** (int $i = 0; i < |S_1|; i + +$) **do**
8. **for** (int $j = 0; j < |S_2|; j + +$) **do**
9. **if** ($S_1[i] == S_2[i]$) **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** $F[|S_1|][|S_2|]$;
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 \( S_1 \) (\( S_1[1], S_1[2], \ldots, S_1[i] \)) into a string consisting of \( j \) first symbols of \( S_2 \) (\( S_2[1], S_2[2], \ldots, S_2[j] \)).

- In case \( S_1[n] = S_2[m] \) (Figure 3a), we only need to convert \( S_1[1], S_1[2], \ldots, S_1[n-1] \) into \( S_2[1], S_2[2], \ldots, S_2[m-1] \). The value of \( F \) will be \( F[n][m] = F[n-1][m-1] \).
- In case \( S_1[n] \neq S_2[m] \), we need to use one of the three transformations at position \( S_1[n] \).

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

### 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 [34], codes [26], and certain users in social media [45] to accelerate the similarity join. String \( S \) is the longest

---

**Figure 3:** Illustration of transformations in Levenshtein distance
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) $d$

```java
1: $S_1$.toUpperCase();
2: $S_2$.toUpperCase();
3: for (int $i = 0; i < |S_1|; i += 1$) do
4:   $F[0][j] = 0$;
5: for (int $j = 0; j < |S_2|; j += 1$) do
6:   $F[0][j] = 0$;
7: for (int $i = 0; i < |S_1|; i += 1$) do
8:   for (int $j = 0; j < |S_2|; j += 1$) do
9:     if ($S_1[i] == S_2[i]$) 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:   end if
14: end for
15: end for
16: 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 $O(|S_1| \times |S_2|)$ and the complexity to calculate the difference between two strings is $O(1)$. Thus, the complexity of the algorithm is $O(|S_1| \times |S_2|)$.

### 3.4. Jaccard distance

Jaccard similarity [22] 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 [1,17,20,29,37,43]. The Jaccard similarity is defined as in equation 1. For example, given the two strings $S_1 = \text{"Bahamas"}$ and $S_2 = \text{"Panama"}$, 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: $S_1\.toUpperCase();$
2: $S_2\.toUpperCase();$
3: String $str_1 = new HashSet <>();$
4: String $str_2 = new HashSet <>();$
5: for (int $i = 0; i < |S_1|; i += ) do
6:  $str_1+ = S_1[i];$
7: for (int $j = 0; j < |S_2|; j += ) do
8:  $str_2+ = S_2[j];$
9: int $mergeNum = 0;$
10: int $commonNum = 0;$
11: for all (char $ch_1 : str_1) do
12:  for all (char $ch_2 : str_2) do
13:    if ($ch_1 == ch_2) then
14:      $commonNum += ;$
15: $mergeNum = |str_1| + |str_2| - 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 [24] 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 [7,36,39]. 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)

\[
d = \begin{cases} 
0 & \text{if } m = 0 \\
\frac{1}{3} \left( \frac{m}{|S_1|} + \frac{m}{|S_2|} + \frac{m-t}{m} \right) & \text{otherwise},
\end{cases}
\]

\[
\text{maxdist} = \left\lceil \frac{\max(|S_1|, |S_2|)}{2} \right\rceil - 1.
\]

Algorithm 5 Jaro similarity

\textbf{Input:} Two strings \( S_1 \) and \( S_2 \)

\textbf{Output:} Jaro similarity \( d \)

1. \( S_1\).toUpperCase();
2. \( S_2\).toUpperCase();
3. if \((S_1 == S_2)\) then
   4. \textbf{return} 1.0;
5. int \( m = 0; \)
6. \textbf{for} (int \( i = 0; i < |S_1|; i++ \)) \textbf{do}
7. \textbf{for} (int \( j = \max(0, i - \text{maxdist}); j < \min(|S_2|, i + \text{maxdist} + 1); i++ \)) \textbf{do}
8. \quad if \((S_1[i] == S_2[i] \text{ AND } \text{matchS}_2[j] == 0 )\) then
9. \quad \qquad \text{matchS}_1[i] = 1;
10. \quad \qquad \text{matchS}_2[j] = 1;
11. \quad \qquad \quad m + +;
12. \quad \quad \textbf{break};
13. \textbf{if} \((m == 0)\) then
14. \quad \textbf{return} 0.0;
15. \quad \textbf{return} 0.0;
16. \textbf{if} \((\text{point} == 0)\) then
17. \textbf{for} (int \( i = 0; i < |S_1|; i++ \)) \textbf{do}
18. \textbf{if} \((\text{hashS}_1[i] == 1)\) then
19. \quad \textbf{while} \((\text{matchS}_2[\text{point}] == 0)\) \textbf{do}
20. \quad \quad \text{point} + +;
21. \quad \quad \quad \textbf{if} \((S_1[i]! == S_2[\text{point} + +])\) then
22. \quad \quad \quad \quad t + +;
23. \quad \quad \quad \textbf{t/} = 2;
24. \quad \quad \textbf{return} d = \left( \frac{m/|S_1| + m/|S_2| + (m-t)/m}{3} \right);\)
25. \textbf{return} \( d \);

To calculate \( d \), we need to count the number of coincident characters of the two strings and calculate \( \text{maxdist} \), which is the maximum allowed distance between the coincident characters in the two strings. The two lists \( \text{matchS}_1 \) and \( \text{matchS}_2 \) are used to mark coincident characters in two strings, if \( \text{matchS}_1[i] == 1 \) then the \( i^{th} \) character in \( S_1 \) is the coincident character and if \( \text{matchS}_1[i] == 0 \) 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 \( \text{maxdist} \).
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 $O(|S_1| \times |S_2|)$, and counting the number of moves also falls in the range $O(|S_1| \times |S_2|)$. Thus, the complexity of the Jaro similarity algorithm is $O(|S_1| \times |S_2|)$.

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 [2, 14, 18, 38]. 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).

$$d = d_j + l \times p \times (1 - d_j).$$

Algorithm 6 Jaro - Winkler similarity

**Input:** Two strings $S_1$ and $S_2$

**Output:** Jaro - Winkler similarity $d$

1: $S_1$.toUpperCase();
2: $S_2$.toUpperCase();
3: double $p = 0.1$;
4: double $d_j = JaroSimilarity(S_1, S_2)$;
5: double $d = d_j$;
6: if $(d_j > 0.7)$ then
7: int $l = 0$;
8: for (int $i = 0; i < \min(|S_1|, |S_2|); i + +)$ do
9: if ($S_1[i] == S_2[i]$) then
10: $l + +$;
11: else
12: break;
13: $l = \min(l, 4)$;
14: $d = d_j + l \times p \times (1 - d_j)$;
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 $O(|S_1| \times |S_2|)$ 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 $O(|S_1| \times |S_2|)$. 
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 [3]. 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

<table>
<thead>
<tr>
<th>Test</th>
<th>Dataset L</th>
<th>Dataset R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Num_record</td>
</tr>
<tr>
<td>1</td>
<td>512MB</td>
<td>1,341,628</td>
</tr>
<tr>
<td>2</td>
<td>2GB</td>
<td>5,366,662</td>
</tr>
<tr>
<td>3</td>
<td>3GB</td>
<td>8,049,298</td>
</tr>
<tr>
<td>4</td>
<td>3GB</td>
<td>8,049,298</td>
</tr>
<tr>
<td>5</td>
<td>6GB</td>
<td>16,098,601</td>
</tr>
</tbody>
</table>

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.

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.

\[ d > = 0.99 \]

\[ d > = 0.98 \]

\[ d > = 0.97 \]

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

<table>
<thead>
<tr>
<th>Dataset $R$</th>
<th>Dataset $L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>150 North Michigan Ave</td>
<td>150 North Michigan Ave</td>
</tr>
<tr>
<td>605 3rd Avenue</td>
<td>605 Third Avenue</td>
</tr>
<tr>
<td>300 North Meridian Street</td>
<td>300 North Meridian Street</td>
</tr>
<tr>
<td>400 High St SE</td>
<td>400 High Street S.E</td>
</tr>
<tr>
<td>2203 Rowan Street</td>
<td>2203 Rowen St</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Num_records</th>
<th>False positives</th>
<th>False negative</th>
<th>Fuzzy threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&lt;=12</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&lt;=5</td>
</tr>
<tr>
<td>LCS</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&lt;=5</td>
</tr>
<tr>
<td>Jaccard</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&gt;=0.625</td>
</tr>
<tr>
<td>Jaro</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&gt;=0.8574</td>
</tr>
<tr>
<td>Jaro - Winkler</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&gt;=0.9144</td>
</tr>
<tr>
<td>Hamming</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>&lt;=5</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&lt;=5</td>
</tr>
<tr>
<td>LCS</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&lt;=5</td>
</tr>
<tr>
<td>Jaccard</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>&gt;=0.91</td>
</tr>
<tr>
<td>Jaro</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>&gt;=0.91</td>
</tr>
<tr>
<td>Jaro - Winkler</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>&gt;=0.91</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Records in R</th>
<th>Records in L</th>
<th>Hamming</th>
<th>Levenshtein</th>
<th>LCS</th>
<th>Jaccard</th>
<th>Jaro</th>
<th>Jaro - Winkler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antigua and</td>
<td>Antigua &amp;</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>0.9090</td>
<td>0.8444</td>
<td>0.9066</td>
</tr>
<tr>
<td>Bahamas</td>
<td>Panama</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0.2857</td>
<td>0.7460</td>
<td>0.7460</td>
</tr>
<tr>
<td>Bosnia and</td>
<td>Bosnia &amp;</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>0.8066</td>
<td>0.8413</td>
<td>0.9048</td>
</tr>
<tr>
<td>Herzegovina</td>
<td>Herzegovina</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cabo Verde</td>
<td>Cape Verde</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.7</td>
<td>0.7416</td>
<td>0.7933</td>
</tr>
<tr>
<td>Central African Republic</td>
<td>Central African Rep.</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.7857</td>
<td>0.9138</td>
<td>0.9483</td>
</tr>
<tr>
<td>China</td>
<td>Hong Kong</td>
<td>22</td>
<td>16</td>
<td>16</td>
<td>0.5</td>
<td>0.5244</td>
<td>0.5244</td>
</tr>
<tr>
<td>Congo</td>
<td>Tonga</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0.5</td>
<td>0.7333</td>
<td>0.7333</td>
</tr>
<tr>
<td>Cook Islands</td>
<td>Cook Islands</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>0.7092</td>
<td>0.9019</td>
<td>0.9411</td>
</tr>
<tr>
<td>Côte d'Ivoire</td>
<td>Cote d'Ivoire</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.9090</td>
<td>0.9487</td>
<td>0.9528</td>
</tr>
<tr>
<td>Federated States of Micronesia</td>
<td>United States</td>
<td>28</td>
<td>20</td>
<td>20</td>
<td>0.5714</td>
<td>0.6016</td>
<td>0.6016</td>
</tr>
<tr>
<td>Gambia</td>
<td>Zambia</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.6666</td>
<td>0.8888</td>
<td>0.8888</td>
</tr>
<tr>
<td>Myanmar/</td>
<td>Panama</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>0.3333</td>
<td>0.5474</td>
<td>0.5474</td>
</tr>
<tr>
<td>Burma</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>North</td>
<td>New Caledonia</td>
<td>14</td>
<td>6</td>
<td>6</td>
<td>0.5714</td>
<td>0.7641</td>
<td>0.7876</td>
</tr>
<tr>
<td>Macedonia</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Saint Kitts</td>
<td>Saint Kitts &amp;</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0.8181</td>
<td>0.9348</td>
<td>0.9699</td>
</tr>
<tr>
<td>and Nevis</td>
<td>Nevis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>South Sudan</td>
<td>South Korea</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>0.6833</td>
<td>0.7515</td>
<td>0.8545</td>
</tr>
<tr>
<td>São Tomé and</td>
<td>São Tomé &amp;</td>
<td>14</td>
<td>6</td>
<td>6</td>
<td>0.7686</td>
<td>0.7345</td>
<td>0.7553</td>
</tr>
<tr>
<td>Principe</td>
<td>Principe</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timor- Leste</td>
<td>Sierra Leone</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>0.5</td>
<td>0.5237</td>
<td>0.5237</td>
</tr>
<tr>
<td>Trinidad and</td>
<td>Trinidad &amp;</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td>0.9990</td>
<td>0.8756</td>
<td>0.9254</td>
</tr>
<tr>
<td>Tobago</td>
<td>Tobago</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vatican City</td>
<td>Vatican</td>
<td>9</td>
<td>11</td>
<td>11</td>
<td>0.6</td>
<td>0.7962</td>
<td>0.8777</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Num_records</th>
<th>False positives</th>
<th>False negative</th>
<th>Fuzzy threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming</td>
<td>17</td>
<td>6</td>
<td>0</td>
<td>&lt;=15</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>&lt;=6</td>
</tr>
<tr>
<td>LCS</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>&lt;=6</td>
</tr>
<tr>
<td>Jaccard</td>
<td>17</td>
<td>6</td>
<td>0</td>
<td>&gt;=0.5</td>
</tr>
<tr>
<td>Jaro</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>&gt;=0.7325</td>
</tr>
<tr>
<td>Jaro - Winkler</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>&gt;=0.7333</td>
</tr>
</tbody>
</table>

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 set, runs with large input datasets, and has a good output enrichment performance and an
Figure 6: Processing time (minutes) of Hamming, Levenshtein, and LCS

Table 11: Comparison of the six algorithms

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

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. 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


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