Efficient Record Linkage in Large Data Sets

Paper: 113
Liang Jin, Chen Li, Sharad Mehrotra
Department of Information and Computer Science
University of California, Irvine, CA 92697, USA
{liangj,chenli,sharad}@ics.uci.edu

Abstract

This paper describes an efficient approach to record linkage. Given two lists of records, the record-linkage problem consists of determining all pairs that are similar to each other, where the overall similarity between two records is defined based on domain-specific similarities over individual attributes constituting the record. The record-linkage problem arises naturally in the context of data cleansing that usually precedes data analysis and mining. The scalability issue of record linkage has previously been addressed in [15], where the authors proposed a sorted-neighborhood approach to improve performance. Since that original work, the repertoire of database techniques dealing with multidimensional data sets has significantly increased. Specifically, many effective and efficient approaches for distance-preserving transforms and similarity joins have been developed. Based on these advances, we explore a novel approach to record linkage. For each attribute of records, we first map values to a multidimensional Euclidean space that preserves domain-specific similarity. Many mapping algorithms can be applied, and we use the Fastmap approach [12] as an example. Given the merging rule that defines when two records are similar based on their attribute-level similarities, a set of attributes are chosen along which the merge will proceed. A multidimensional similarity join (using the algorithm proposed by Hjaltason and Samet [16]) over the chosen attributes is used to determine similar pairs of records. Our extensive experiments using real data sets show that our solution has very good efficiency and accuracy.

1 Introduction

The record-linkage problem – identifying and linking duplicate records – arises in the context of data cleansing, which is a necessary pre-step to many database applications. Databases frequently contain approximately duplicate fields and records that refer to the same real-world entity, but are not identical. As the following example illustrates, variations in representation may arise from typographical errors, misspellings, abbreviations, as well as other sources. This problem is especially severe when data to be stored within databases is automatically extracted from unstructured or semi-structured documents or web pages [4].

EXAMPLE 1.1 A hospital of a medical school has a database with thousands of patient records.[1] Every year it receives data from other sources, such as the government or local organizations. The data includes all kinds of information about patients, such as whether a patient has moved to a new place. It is important for the hospital to link the records in its own database with the data from other sources, so that they can collect more information about patients. However, usually the same information (e.g., name, SSN, address, telephone number) can be represented in different formats. For instance, a patient name can be represented as "Tom Franz" or "Franz, T." or other forms. In addition, there could be typos in the data. For example, name "Franz" may be mistakenly recorded as "Frans". The main task here is to link records from different databases in the presence of mismatched information. □

With the increasing importance of data linkage in a variety of data-analysis applications, developing effective and efficient techniques for record linkage has emerged as an important problem [26, 27]. It is further evidenced by the emergence of numerous organizations (e.g., Trillium, FirstLogic, Vality, DataFlux) that are developing specialized domain-specific record-linkage and data-cleansing tools.

Three primary challenges arise in the context of record linkage. First, it is important to determine similarity functions that can be used to link two records as duplicates. Such a similarity function consists of two levels. First, similarity metrics need to be defined at the level of each field to determine similarity of different values of the same field. Next, field-level similarity metrics need to be combined to determine overall similarity between two records. At the field level, a typical choice is string edit distance, particularly if the primary source of errors are typographic. However, as the example above illustrated, domain-specific similarity measures (e.g., a different function for people's names, addresses, paper references, etc.) are more relevant. At the record level, merging rules that combine field-level similarity measures into an overall record similarity need to be developed. Approaches based on binary classifiers, expectation maximization (EM) methods and support vector machines have been proposed in the literature [4, 7, 35].

The second challenge in record linkage is to provide user-friendly interactive tools for users to specify different transformations, and use the feedback to improve the data quality. Recently a few works [11, 13, 30] have been conducted to solve this problem. The third challenge is that of scale. A simple solution is to use a nested-loop approach to generate the Cartesian product of records, and then use the similarity function(s) to compute the distance between each pair of records. This approach is computationally prohibitive as the two data sizes become large.

[1]For confidentiality reasons, we omit the real name of the hospital.

The scalability issue of the record linkage has previously been studied in [15]. The proposed approach first merges two given lists of records, then sorts records based on lexicographic ordering for each attribute (or a "key" as defined in that paper). A fixed-size sliding window is applied, and records within a window are checked to determine if they are duplicates using a merging rule for determining similarity between records. Notice that, in general, records with similar values for a given field might not appear close to each other in lexicographic ordering. For example, the string edit distance of "Anderson" and "Zanderson" is 1, but their names can be very far from each other in the sorted list. As a result, they might not appear in the same sliding window. The effectiveness of the approach is based on the expectation that if two records are duplicates, they will appear lexicographically close to each other in the sorted list based on at least one key. Even if we choose multiple keys to do the search, the approach is still susceptible to deterministic data-entry errors, e.g., the first character of a key attribute is always erroneous. It is also difficult to determine a value of the window size that provides a "good" tradeoff between accuracy and performance. In addition, it might not be easy to choose good keys to bring similar records close to each other.

Since the original work in [15], many new data-management techniques have a direct bearing on the record-linkage problem. In particular, techniques to map arbitrary similarity spaces into similarity/distance-preserving multidimensional Euclidean spaces have been developed. Furthermore, many efficient multidimensional similarity joins have been studied. In this paper, we develop an efficient strategy to record linkage that exploits these advances. We first consider the situation that a record consists of a single attribute. The record-linkage problem then reduces to linking similar strings in two string collections based on a given similarity metric. We propose a two-step solution for duplicate identification. In the first step, we combine the two sets of records, and map them into a high-dimensional Euclidean space. (A similar strategy in a different context is proposed in [29], where authors map sequences of events to a Euclidean space.) In general, many mapping techniques can be used. As an example, in this paper we focus on the FastMap algorithm [12] due to its simplicity and efficiency. We develop a linear algorithm called "StringMap" to do the mapping, and the algorithm works independently of the specific distance metric.

In the second step, we find similar object pairs in the Euclidean space whose distance is no greater than a new threshold. This new threshold is closely related to the old threshold of string distances. Again, many similarity-join algorithms can be used in this step. In this paper we use the algorithm proposed by Hjaltason and Samet [16] as an example. For each object pair in the result of the second step, we check their corresponding strings to see if their original distance is within the original threshold, and find those similar-record pairs. Our approach has the following advantages. (1) It is "open," since many mapping functions can be applied in the first step, and many high-dimensional similarity-join algorithms can be used in the second step. (2) It does not depend on a specific similarity function of strings. (3) Our extensive experiments using real large data sets show that this approach has very good efficiency and accuracy (greater than 99%).

We next study mechanisms for record linkage when many attributes are used to determine overall record similarity. Specifically, we consider merging rules expressed as logical expressions over similarity predicates based on individual attributes. Such rules would be generated, for example, if a classifier such as a decision tree was used to train an overall similarity function between records using a labeled training set. Given a merging rule, we choose a set of attributes over which the similarity join is performed (as discussed in the single-attribute case), such that similar pairs can be identified with minimal cost.

The rest of the paper is organized as follows. Section 2 gives the formulation of the problem. Section 3 presents the first step of our solution, which maps strings to objects in a Euclidean space. Section 4 presents the second step that conducts a similarity join in the high-dimensional space. Section 5 studies how to solve the record-linkage problem if we have a merging rule over multiple attributes. In Section 6 we give the results of our extensive experiments. We conclude the paper in Section 7.

2 Problem Formulation

2.1 Distance Metrics of Attributes

Consider two relations R and S that share a set of attributes $A_1, \ldots, A_p$. Each attribute A_j has a metric M_j that defines the difference a value of $R.A_j$ and a value of $S.A_j$.[2] There are many ways to define the similarity metric at the attribute level, and domain-specific similarity measures are critical. We take two commonly-used metrics as examples: edit distance and q-gram distance.

Edit distance, a.k.a. Levenshtein distance [25], is a common measure of textual similarity. Formally, given two strings s_1 and s_2, their edit distance, denoted $\Delta_e(s_1, s_2)$, is the *minimum* number of edit operations (insertions, deletions, and substitutions) of single characters that are needed to transform s_1 to s_2. For instance,

$$\Delta_e(\text{“Harrison Ford”}, \text{“Harison Fort”}) = 2.$$

[2]In [15], “keys” constructed from multiple fields are used to represent the similarity between records. These keys can be viewed as attributes in our setting.

In particular, we can remove the third character "r" in the first string, and substitute the last character "d" with "t" to transform it to the second string. Similarly, Δ_e("Jack Lemmon","Jack Lemon") = 1, and Δ_e("Anderson","Zandersson") = 2. It is known that the complexity of computing the edit distance between strings s_1 and s_2 is $O(|s_1| \times |s_2|)$, where $|s_1|$ and $|s_2|$ are the lengths of s_1 and s_2, respectively [36].

Given a string s and an integer q, the set of *q-grams of s*, denoted $G(s)$, is obtained by sliding a window of length q over the characters of string s. For instance:

G("Harrison Ford")={'Ha', 'ar', 'rr', 'ri', 'is', 'so', 'on', 'n ', ' F', 'Fo', 'or', 'rd'}.
G("Harison Fort")={'Ha', 'ar', 'ri', 'is', 'so', 'on', 'n ', ' F', 'Fo', 'or', 'rt'}.

The *relative q-gram distance* between two strings s_1 and s_2, denoted $\Delta_q(s_1, s_2)$, is defined as:

$$\Delta_q(s_1, s_2) = 1 - \frac{|G(s_1) \cap G(s_2)|}{|G(s_1) \cup G(s_2)|}$$

For example, Δ_q("Harrison Ford", "Harison Fort") = 1 - $\frac{10}{13} \approx 0.23$. Clearly the smaller the relative q-gram distance between two strings is, the more similar they are.

2.2 Similarity Merging Rule

Given distance metrics for attributes $A_1, A_2, \ldots, A_p$, there is an *overall function* that determines whether or not two records are to be considered as duplicates. Such a function may either be supplied manually by a human (e.g., an analyst in the data analysis), or alternatively, learned automatically using a classification technique. While the specifics of the method used to learn such a function are not of interest to us in this paper, we do make an assumption that the function is captured in the form of a rule discussed below. The form of rules considered include those that could be learned using inductive rule-based techniques such as decision trees. Furthermore, this form of merging rules are consistent with the merging functions considered in [15].

Let r and s be two records whose similarity is being determined. A merging rule for records r and s is of the the following disjunctive format.

$$\begin{array}{ll} & M_1(A_1) \leq \delta_{1,1} \wedge \quad \ldots \quad \wedge M_p(A_p) \leq \delta_{1,p} \\ \vee & M_1(A_1) \leq \delta_{2,1} \wedge \quad \ldots \quad \wedge M_p(A_p) \leq \delta_{2,p} \\ & \vdots \\ \vee & M_1(A_1) \leq \delta_{k,1} \wedge \quad \ldots \quad \wedge M_p(A_p) \leq \delta_{k,p} \end{array} \qquad \text{(Merging Rule)}$$

For each conjunct $M_j(A_j) \leq \delta_{i,j}$ ($i = 1, \ldots, k$, and $j = 1, \ldots, p$), the value $\delta(i, j)$ is a threshold using the metric function M_j on attribute A_j. The conjunct means that two records r and s from the two relations should satisfy the condition $M_j(r.A_j, s.A_j) \leq \delta_{i,j}$.

For instance, given three attributes about papers, *title*, *author*, and *year*, suppose we use the edit distance Δ_e as a metric for attributes *author* and *year*, and the relative q-gram function Δ_q as a metric for attribute *title*. Then we could have the following rule:

$$\begin{array}{lllll} & \Delta_q(title) \leq 0.10 & \wedge\ \Delta_e(name) \leq 4 & \wedge\ \Delta_e(year) \leq 1 \\ \vee & \Delta_q(title) \leq 0.15 & \wedge\ \Delta_e(name) \leq 2 & \wedge\ \Delta_e(year) \leq 2 \end{array} \qquad \text{(Query } Q_1\text{)}$$

The record-linkage problem studied in this paper is to find pairs of records (r, s) from relations R and S, such that each pair satisfies the merging rule defined above. The quadratic nested-loop solution is not desirable, since as the size of the data set increases, this solution becomes computationally prohibitive. For instance, in our experiments, it took more than 6 hours to use this approach on two single-attribute relations, each with just $10K$ names, assuming the edit-distance function is used (see Section 6 for our experimental setting). Since by definition, the record-linkage problem is based on *approximate* matching of those individual metrics for different attributes, an efficient solution that might miss some pairs (but with a high accuracy) might be more preferable.

In this paper, we first consider the case of a single attribute in Sections 3 and 4, then study the case of multiple attributes in Section 5. Table 1 summarizes some symbols used in this paper.

Symbol	Meaning
R, S	two relations for record linkage
M	metric function in the original space
Δ_e	edit distance
Δ_q	relative qgram distance
δ	threshold in the original string metric space
δ'	new threshold in the mapped Euclidean space
d	dimensionality of the Euclidean space

Table 1: Symbols.

3 Step 1: Mapping Strings to Euclidean Space

We first consider the single-attribute case, where R and S share one attribute A. Thus the record-linkage problem reduces to linking similar strings in R and S based on a given similarity metric. Formally, given a predefined threshold value δ, we want to find pairs of strings (r, s) from R and S respectively, such that according to the metric M of attribute A, the distance of r and s is within δ, i.e.,

$$M(r.A, s.A) \leq \delta.$$

Such a string pair is called a *similar-string pair*; otherwise, it is called a *dissimilar-string pair*. Our proposed approach has two steps. In the first step, we map strings to objects in a multidimensional Euclidean space, such that the mapped space preserves the original string distances. In the second step, a multi-dimensional similarity join is conducted in the Euclidean space. In this section we discuss the first step.

3.1 StringMap: Mapping Strings to Objects

In the first step, we combine the strings from the two relations into one set, and embed them into a Euclidean space. Formally, the process is the following.

> Given N objects $O_1, \ldots, O_N$ and their distances (e.g., an $N \times N$ distance matrix), find N points $P_1, \ldots, P_N$ in a d-dimensional Euclidean space, such that the distances are maintained as well as possible.

Many mapping/embedding functions can be used, such as multidimensional scaling [23, 37], FastMap [12], Lipschitz [5], SparseMap [18], and MetricMap [34]. These algorithms have different properties in terms of their efficiency, contractiveness, and stress. (See [17] for a good survey.) In this paper we use the FastMap algorithm because of its efficiency and distance-preserving capability. Due to space limitation, here we briefly review the algorithm. (See [12] for the details of FastMap.) Its main idea is to find d mutually-orthogonal axes. It iteratively chooses two objects (called "pivot objects") to form an axis. For each axis, FastMap projects all objects onto this axis by computing their coordinates using their distance matrix. It also computes the remaining distance matrix after the projections.

We modify the FastMap slightly and propose an algorithm called "StringMap," as shown in Figure 1 (a).[3] StringMap iterates to find pivot strings to form d orthogonal directions, and computes the coordinates of the N strings on the d axes. The function $ChoosePivot(int\ h,\ Metric\ M)$ selects two strings to form an axis for the d-th dimension. These two strings should be as far from each other as possible, and the function iterates m times to find the seeds. A typical m value could be 5 (as in [12]). The algorithm assumes the dimensionality d of the target space, and we will discuss how to choose a good d value shortly.

One important function is

$$GetDistance(int\ a,\ int\ b,\ int\ h,\ Metric\ M)$$

[3]Notice that the algorithm does not require the complete distance matrix to be given, since it only computes those distances when necessary.

```
Algorithm StringMap
Input: • $N$ strings: $t[1,\ldots,N]$.
       • $d$: Dimensionality of Euclidean space.
       • $M$: Metric function on strings.
Output: $N$ corresponding objects in the new space.
Variables: • PA[1,2][1,...,N]: $2 \times d$ pivot strings.
           • coord[1,...,N][1,...,d]: object coordinates.

Method:
 for (h = 1 to d) {
    ($p_1$,$p_2$) = ChoosePivot(h,M); // choose pivot strings
    PA[1,h]= $p_1$; PA[2,h] = $p_2$; // store them
    dist = GetDistance($p_1$, $p_2$, $h$, M);
    if (dist == 0) {
        // set all coordinates in the $h$-th dimension to 0
        for (i = 1 to N) { coord[i,h] = 0 };
        break;
    }

    // compute coordinates of strings on this axis
    for (i = 1 to N) {
        x = GetDistance(t[i], $p_1$, h, M);
        y = GetDistance(t[i], $p_2$, h, M);
        coord[i,h] = (x*x + dist*dist - y*y) / (2*dist);
    }
 }
```

(a) The main algorithm

```
// choose two pivot strings on the $h$-th dimension
Function ChoosePivot(int h, Metric M)
{
    seed $s_a$ = a random string from $t[1],\ldots,t[N]$;
    for (i = 1 to m) { // a typical m value could be 5
        // use function GetDistance(.,.,h,M)
        // to compute distances
        seed $s_b$ = a farthest point from $s_a$;
        seed $s_a$ = a farthest point from $s_b$;
    }
    return ($s_a$, $s_b$);
}

// get distance of two strings (indexed by $a$ and $b$)
// after they are projected onto the first $h-1$ axes
Function GetDistance(int a, int b, int h, Metric M)
{
    A = t[a]; B = t[b]; // get strings
    dist = M(A, B); // get original metric distance
    for (i = 1 to h - 1) {
        // get their difference on dimension i
        w = coord[a,i] - coord[b,i];
        dist = $\sqrt{|dist \times dist - w \times w|}$;
    }
    return ( dist );
}
```

(b) Functions

Figure 1: Algorithm StringMap.

which computes the distance between strings (indexed by a and b) after they are mapped to the first $h-1$ axes. As shown in Figure 1 (b), it iterates over the $h-1$ dimensions, and does the computation using only the two strings and their already-computed coordinates on the $h-1$ dimensions.

There are a few observations about the algorithm.

1. The StringMap algorithm removes the recursion in FastMap.
2. In the last line of the algorithm, the computation of $coord[i,h]$ is not symmetric with respect to values x and y.
3. In function *GetDistance*(), it is known that $dist * dist - w * w$ can be negative [34] . In StringMap, we take the square root of the *absolute* value to compute the new distance. In our experiments we tried other ways to deal with this case, as described in [17]. Our experimental results indicated that the approach of taking the absolute value can keep the distance well.

All the steps in the StringMap algorithm are linear on the number of strings N. It can be shown that the complexity of the StringMap algorithm is:

$$O(d^2 \times m \times N)$$

assuming it takes $O(1)$ time to compute $M(a, b)$. Notice that a major cost in the algorithm is spent in function $ChoosePivot()$. We can reduce the cost in the function as follows. At each step in the function, we want to find a new string that is as far from a string (e.g., s_a) as possible. Instead of scanning the whole N strings, we can just do sampling to find a string that is very far from s_a. Or we can just stop once we find a string that is "far enough" from s_a, i.e., their distance is above certain value. See [20] for an approximation algorithm for finding this pair efficiently.

3.2 Choosing Dimensionality d

A good dimensionality value d used in algorithm StringMap should have the property that after the mapping, similar strings can be differentiated from those dissimilar ones. On the one hand, the dimensionality d cannot be too small, since otherwise those dissimilar pairs will not "fall apart" from each other. In particular, the distances of similar-string pairs are too close to those of dissimilar ones. On the other hand, d cannot be too high either. There are mainly two reasons. First, the complexity of the StringMap algorithm (see above) is linear to d^2. Second, since we need to do a similarity join in the second step, we want to avoid the curse of dimensionality [3, 10]. In particular, as d increases, it becomes more time-consuming to find object pairs whose distance is within a new threshold. We choose a dimensionality d as follows.

1. Randomly select a small number (say, 1K) of strings from the data sets R and S. Use the nested-loop approach to find all similar-string pairs within threshold δ in the selected strings.
2. Run StringMap using different d values. (Typically d is between 5 and 30.) For each d, compute the new distances of the similar-string pairs. Find their largest new distance w.
3. Find a dimensionality d with a low *cost*:
$$cost = \frac{\text{\# of object pairs within distance } w}{\text{\# of similar-string pairs}} \tag{1}$$
Intuitively, the cost is the average number of object pairs we need to retrieve in step 2 for each similar-string pair, if w is used as the new threshold δ' for selecting similar-object pairs in the mapped space.

Notice that we only use the pairs of selected strings to compute the cost. This value measures how well a new threshold $\delta' = w$ differentiates the similar-string pairs from those dissimilar pairs.

In particular, the string pairs whose new distance is within δ' will be retrieved in step 2. Thus the lower the cost is, relatively the fewer object pairs need to be retrieved in step 2 whose original distance is more than δ. Figure 5 in Section 6.1 shows that typically a good dimensionality value is between 15 and 25.

4 Step 2: Finding Similar-Object Pairs in Euclidean Space

In the second step, we find object pairs whose Euclidean distance is within a new threshold δ'. For each candidate pair, we check the distance of their original strings to see it is within the original threshold δ. In this section we study how to select the new threshold and how to do the similarity join.

4.1 Choosing New Threshold δ'

The selection of the new threshold δ' depends on the mapping function. For instance, we can set $\delta' = \delta$ if the mapping function is *contractive*. That is, for any two strings r and s, we have

$$M(r, s) \leq M'(r', s')$$

where $M(r, s)$ is their distance in the original metric space, and $M'(r', s')$ is the new Euclidean distance of the corresponding objects r' and s' in the new space. In general, suppose there are two constants $c_1, c_2 \geq 1$, such that for any two strings r and s, we have:[4]

$$\frac{1}{c_1} \cdot M(r, s) \leq M'(r', s') \leq c_2 \cdot M(r, s)$$

Then we can just set $\delta' = c_2 \cdot \delta$. Properties of different mapping functions are studied in [17].

Ideally, the threshold δ' should be set to a maximal value of the new distance between any two similar-string pairs in the original space. Then it will guarantee no false dismissals. However, this maximal value could be either too expensive to find (we do not want to have a nested-loop procedure), or the theoretical upper bound could be too large. Since it is acceptable to miss a few pairs, we would like to choose a threshold such that for *most* of the similar-string pairs, their new distances are within this threshold. As shown in our experimental results, even though a theoretical upper bound could be large, most of the new distances could be within a much smaller threshold.

In our approach to selecting the dimensionality d, the threshold w can be used to identify similar pairs in the mapped space. Therefore, here is how we choose the threshold δ'. We randomly select

[4]This equation measures the *distortion* of the mapping [17].

a small number (say, 1K) of strings from data sets R and S. (Notice these selected strings might be different from those used to decide the dimensionality d in Section 3.2.) We find all the similar-string pairs in these selected strings, and compute their new Euclidean distances after StringMap. We choose their maximal new distance as the new threshold δ'. We may do this sampling multiple times (on different sets of selected strings), and choose δ' as the maximal new distance of those similar-string pairs. In Section 6.2 we will give experimental results on choosing δ'.

4.2 Finding Object Pairs within δ'

We want to find all those object pairs whose new distance is within this new threshold δ'. Similarity joins over multidimensional spaces have been studied in [1, 6, 9, 21, 22, 24, 28, 33, 32]. Many algorithms can be used in this step. In this paper we use a simplified version of the algorithm in [16] as an example. We could instead have chosen any of those algorithms for our purpose. We chose the algorithm in [16] due to its simplicity and availability of the code. This approach suffices for our purpose, since our intent is to establish a base line for our approach of mapping record linkage into a similarity-join problem.

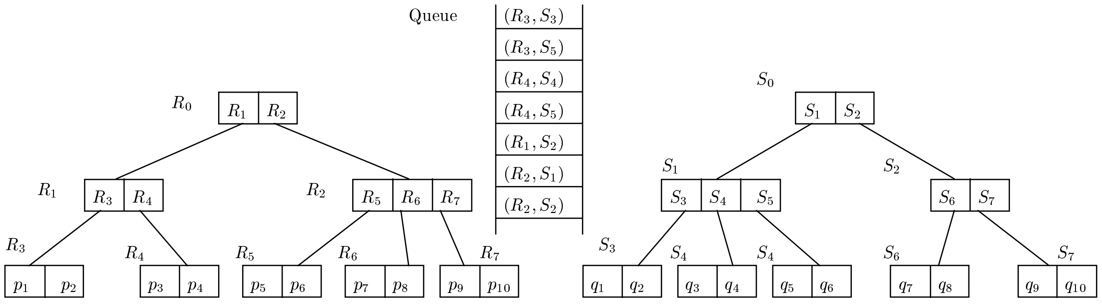


Figure 2: Finding similar-object pairs using R-trees.

Due to space limitation, we briefly explain the main idea of the algorithm. (See [16] for details.) We first build two R-trees for the mapped objects of the two string sets, respectively. We traverse the two trees from the roots to the leaf nodes to find those pairs of objects within distance δ'. As we do the traversal, a queue is used to store pairs of nodes (internal nodes or leaf nodes) from the two trees.[5] We only insert those node pairs that can potentially yield object pairs that satisfy the condition. Those node pairs that cannot produce results are pruned in the traversal, i.e., they are never inserted into the queue.

Take Figure 2 as an example. Initially, a pair of the root nodes (R_0, S_0) is inserted into the

[5]Since we just want to find those object pairs whose distance is within δ', we do *not* need a *priority* queue. A priority queue based on object-pair distances is necessary in [16], since they want to find the "top-k" object pairs with the smallest distances.

queue. At each step, we dequeue the head pair (R_i, S_j). If both nodes are internal nodes (i.e., hyper-rectangle regions), we consider all the pairs of their children. For each pair (R_a, S_b), we compute their "distance," which is a *lower bound* of all the distances of their child objects. (The case of a node-object pair is handled similarly.) For instance, we can use the MINDIST function [31] to compute the distance between two nodes. Then we can prune node pairs as follows: if the distance of a node pair is greater than δ', we do not insert this pair into the queue. The reason is that the lower-bound property guarantees that these two nodes cannot generate object pairs whose distance is within δ'. On the other hand, if the distance of two nodes is within δ', we insert this pair into the queue.

For instance, when we consider the two child nodes of each of the two root nodes in the figure, we have four pairs:

$$(R_1, S_1), (R_1, S_2), (R_2, S_1), (R_2, S_2)$$

Suppose each of them has a MINDIST distance within δ', then we insert them into the queue. We remove the pair (R_1, S_1) from the queue, and consider pairs of their child nodes:

$$(R_3, S_3), (R_3, S_4), (R_3, S_5), (R_4, S_3), (R_4, S_4), (R_4, S_5)$$

For each pair, we compute their MINDIST. If, for example, the distance between R_3 and S_4 is greater than δ', we will not consider this pair, since they cannot generate object pairs that have a distance within δ'. In other words, all the pairs of their descendants are greater than δ'.

Suppose only the following pairs have a MINDIST distance within δ':

$$(R_3, S_3), (R_3, S_5), (R_4, S_4), (R_4, S_5)$$

Then we insert these four pairs into the queue. The status of the queue is shown in the figure. Eventually we have a pair of *objects* from the queue. Then we compute their Euclidean distance to check if it is within δ'. If so, we compute the original metric distance of their original strings. We output this pair of strings if their original metric distance is within the original threshold δ.

Different strategies can be used to traverse the two R-trees, such as depth first and breadth first. Our experiments show that the depth-first traversal strategy has several advantages.

1. It can effectively reduce the queue size, since pairs of objects in the leaf nodes can be processed early, and they can be dequeued from the queue. Thus the memory requirement of the algorithm tends to be small. In our experiments, when each data set had about 27K strings, the breadth-first strategy required about 1.2GB memory, while the depth-first strategy only requested about 30MB memory.

2. It also reduces the time that the first pair is output, since it can reach the leaf nodes more quickly than the breadth-first strategy. If we use the breadth-first traversal strategy, we need to generate a very large number of pairs before processing some pairs of leaf nodes.

4.3 Comparison with the Approach in [14]

Steps 1 and 2 discussed so far can be viewed as a mechanism for computing similarity join over strings where any arbitrary distance function can be used to determine the similarity between strings. If we were to constrain our problem to only consider edit distance as a distance metric, we could instead use the qgram-based approach described in [14].

In [14], the authors first pad each string on both sides by a set of special characters, and then use a sliding window of size q over the augmented string to compute substrings of size q (referred to as q-grams). The number of resulting q-grams depends upon the length of the string and the number q. An additional column is added to the relation scheme to store the q-grams corresponding to the string. Thus, a tuple is replicated as many times as there are q-grams for the string contained in the tuple. While the relation size increases by a factor of number of q-grams for a given string, the advantage of the approach is that, instead of computing an expensive predicate (i.e., edit distance between strings) in a nested-loop join, by using q-grams, the original string-join query can be rewritten as an equi-join query over q-grams with a suitable aggregation over the string identifier. This way a cheaper equi-join (possibly using hash join or sort-merge approach) can be evaluated on the q-grams, and the expensive edit distance is computed only on strings that satisfy the filter imposed by q-grams. The effectiveness of the approach is predicated on the hope that the penalty in terms of increase in size of the database is offset by the savings due to pruning the number of expensive edit-distance computations and the benefit of hash-join over the nested-loop approach.

While the approach described in [14] is attractive, our approach has several advantages in the context of the record-linkage problem. First, the distance metrics used at individual attributes might not be edit distance. As argued earlier, domain-specific methods work better in identifying similar records. Furthermore, the algorithm in [14] is geared towards finding all the string pairs that are within a fixed edit-distance threshold. Since the record-linkage problem, by definition, is based on approximate matching, solutions that might miss some such pairs (say those that obtain around 99% accuracy) but result in significant time savings might be more preferable. As we will see in Section 6.3, our algorithm works very efficiently, while still achieving a very high accuracy. In addition, the implementation of the qgram-based approach inside a database might be less efficient than a direct implementation using other languages (e.g., the C language). Finally, in the record-linkage context, catalogs to be merged rarely have a single string attribute – rather records may consist of multiple string attributes. The data-size increase will grow multiplicatively with the number of string attributes in the relation, thus the efficiency of the q-gram approach can deteriorate. In contrast, the string-join approach proposed in this paper (as shown in the next section) can be easily combined with an appropriate multi-attribute merging algorithm without much overhead.

5 Combining Multiple Attributes

In this section we discuss how to join over multiple attributes efficiently where the merging rule is of the disjunctive normal form (DNF) discussed in Section 2.2. We first study how a single disjunct (in the form of a conjunctive clause) can be evaluated, then describe the more general case when the merging rule consists of multiple conjunctive clauses.

5.1 Single Conjunctive Clause

For a single conjunctive clause, we can process the most "selective" attribute to find the candidate pairs that satisfy this conjunct condition, and then check other conjunct conditions. For instance, consider the following clause.

$$\Delta_q(\mathit{title}) \leq 0.15 \ \wedge \ \Delta_e(\mathit{name}) \leq 3 \ \wedge \ \Delta_e(\mathit{year}) \leq 1 \qquad \text{(Query } Q_2\text{)}$$

We could first do a similarity search to find all the record pairs that satisfy the first condition, $\Delta_q(\mathit{title}) \leq 0.15$. For each of the returned candidate pairs, we check if it satisfies the other two conditions on the *name* and *year* attributes. Alternatively, we can choose either *name* or *year* to do the similarity join. Our experiments show that the step of testing other attributes takes relatively much less time than the step of finding the candidate record pairs, thus we mainly focus on the time of doing the similarity join that finds the candidate pairs. We can use existing techniques on estimating the performance of spatial joins (e.g., [2, 19]), and choose the attribute that takes the least time to do the corresponding similarity join. (This attribute is called the *most selective attribute* for this conjunctive clause.) Notice that similar to [15], we could also search along multiple attributes of the conjunction to improve accuracy. Since the mapping in Step 1 does not guarantee that all the relevant string pairs will be found, using multiple attributes may improve accuracy. However, as will be shown in the experimental section, since our strategy for single attributes is able to identify matching string pairs at a very high accuracy (over 99%), traversals along multiple attribute is not necessary. In contrast, since string lexicographic ordering may not preserve domain-specific similarity as well, to get the same degree of accuracy, the approach in [15] requires traversals along multiple keys.

5.2 Disjunctive Clauses

The problem becomes more challenging in the case of multiple conjunctive clauses. Take query Q_1 in Section 2.2 as an example. We have at least the following different approaches to answering this query.

1. Approach A: Find all record pairs that satisfy the first conjunctive clause by doing a similarity search using the conjunct $\Delta_q(title) \leq 0.10$. Find all record pairs satisfying the second conjunctive clause by doing a similarity search using the conjunct $\Delta_e(name) \leq 2$. Take the union of these two sets of results.

2. Approach B: Do a similarity search to find record pairs that satisfy $\Delta_q(title) \leq 0.15$ in the second conjunctive clause. These pairs also include all the pairs satisfying the first conjunctive clause, since $\Delta_q(title) \leq 0.10$ implies $\Delta_q(title) \leq 0.15$. Among all these pairs, find those satisfying the merging rule.

Approach A needs to do two similarity searches, while approach B requires only one. However, both similarity searches in approach A could be more selective than the single similarity search in approach B. Which approach is better depends on the data set.

The example shows that to answer a conjunctive clause, we can choose at most one conjunct in it to do a similarity join. In addition, once we choose a conjunct $M_j(A_j) \leq \delta(i,j)$ to do a similarity join, we do not need to do a similarity join for any other conjunctive clause that has a conjunct $M_j(A_j) \leq \delta(k,j)$, where $\delta(k,j) \leq \delta(i,j)$. The reason is that a superset of the results for the conjunct $M_j(A_j) \leq \delta(k,j)$ has been returned by the similarity search. As the number of attributes and the number of conjunctive clauses increase, there could be an exponential number of possible ways to answer the query. In fact, assuming the time of doing a similarity search for each conjunct is given, we can show that the problem of finding an optimal solution (with a minimal total time) to answer the query is NP-complete. (The NP-hardness can be proven by reducing the problem to the NP-complete set-cover problem [8].)

Since it could be too time-consuming to find an optimal solution, we propose two heuristic-based algorithms for finding a good one.

- Algorithm 1: Treat all the conjunctive clauses separately. For each of them, choose the most selective attribute to do the corresponding similarity join. If we choose the same attribute A_j in two conjunctive clauses, and their corresponding thresholds $\delta_{i,j} \leq \delta_{k,j}$, then we only choose the threshold $\delta_{k,j}$ to do the similarity search for the second clause, saving one similarity search for the first clause. Take the union of results of all conjunctive clauses.

- Algorithm 2: For each attribute, choose the largest threshold among all its conjuncts. Among all the largest thresholds of different attributes, choose the most selective one to do a similarity join. Among the results, find the record pairs that satisfy the merging rule.

For instance, consider the query Q_1 in Section 2.2. Suppose Algorithm 1 chooses $\Delta_q(title) \leq 0.10$ as the most selective condition for the first clause, and $\Delta_e(name) \leq 2$ for the second one. Thus it will produce the approach A above. For Algorithm 2, the largest thresholds of the three attributes *title*, *name*, and *year* are 0.15, 4, and 2, respectively. Suppose $\Delta_q(title) \leq 0.15$ is the most selective one. This algorithm will produce approach B as the solution. In general, Algorithm 1 works better than Algorithm 2 if doing multiple similarity searches with small thresholds is more efficient than one with a large threshold.

6 Experiments

In this section we present our extensive experimental results to evaluate our solution. The following are three main sources we used.

1. **Source 1** consists of 54,000 movie star names collected from The Internet Movie Database.[6] The length of each name varies from 5 to 20, and their average length is about 12. Figure 3(a) shows the distribution of the string lengths.

2. **Source 2** is from the Die Vorfahren Database, a database of mostly Pomeranian surnames and geographic locations.[7] This 2001 DV database experiment contains of 133,101 full names that have appeared in the *Die Pommerschen Leute* Newsletter, Die Vorfahren section over the 19.5 years of its publication. The lengths of names are less than 40, and their mean length is around 15. Figure 3(b) shows the distribution of the string lengths.

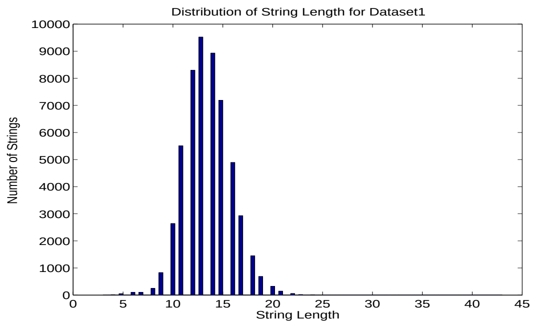
3. **Source 3** is from the publications in DBLP.[8] We randomly selected 20,000 papers in proceedings. We use this data source to show how to do data linkage in multiple-attribute cases. Figure 4 shows the distributions of the three attributes. (The distribution of the title lengths

[6]`http://www.imdb.com/`

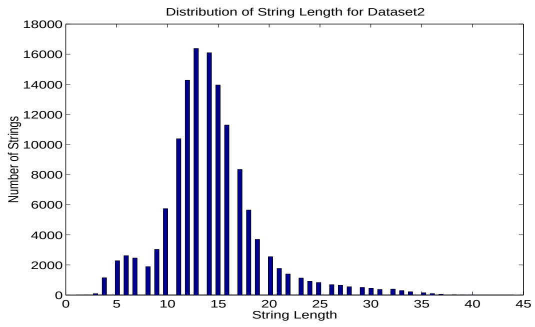
[7]`http://feefhs.org/dpl/dv/indexdv.html`

[8]`http://www.informatik.uni-trier.de/ ley/db/index.html`

is not “smooth,” and we believe the reason is that titles consist of words, whose lengths are not “smoothly” distributed.)

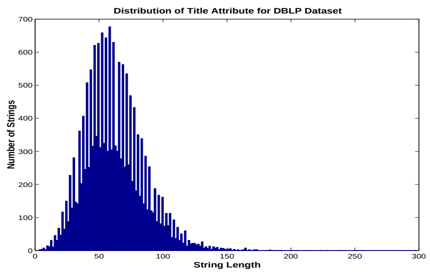


(a) Source 1 (IMDB)

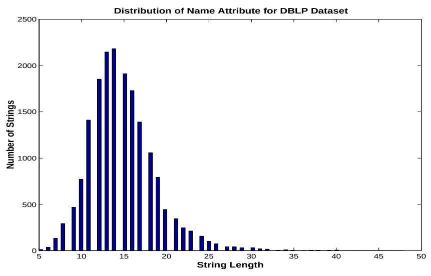


(b) Source 2 (Die Vorfahren)

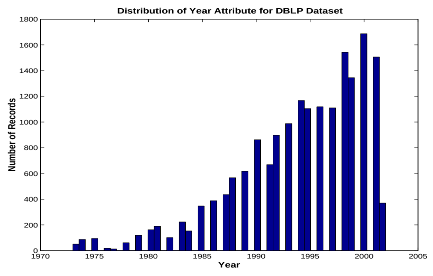
Figure 3: String-length distribution of sources 1 and 2.



(a) Title (string length)



(b) Name (string length)



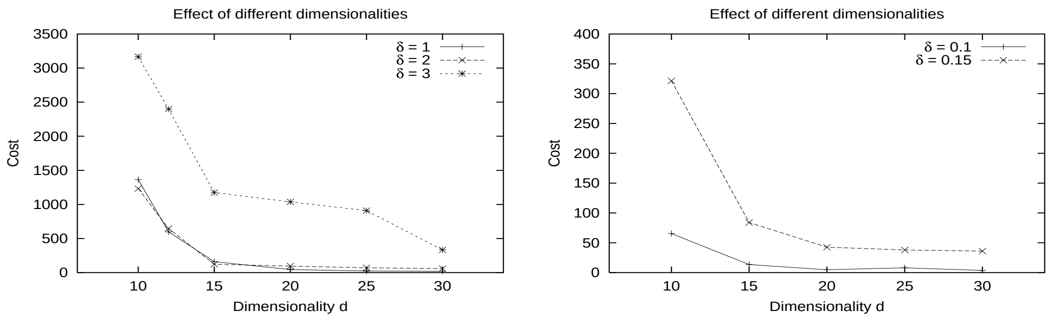
(c) Year (value)

Figure 4: Distributions of source 3 (DBLP).

All the experiments were run on a PC, with a 1.5GHz Athlon CPU and 512MB memory. The operating system is Windows 2000, and the complier is gnu C++ running in cygwin. This setting shows that our approach does not have specific hardware and software requirements. We used 8192 as the page size to build R-trees. Most of our experimental results are similar for three sources. Due to space limitation, we mainly report the results on data source 1.

6.1 Choosing Dimensionality d

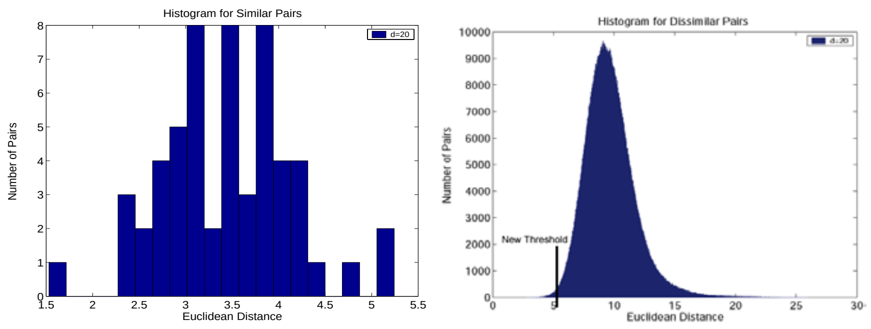
As discussed in Section 3.2, we need to choose a good dimensionality d for the StringMap algorithm. To select a d value, we randomly selected 2K strings from Source 1 to form two data sets with the same size, and ran StringMap using different dimensionalities. For edit distance, we considered the case where $\delta = 1$, 2, and 3. For the q-gram metric, we considered $\delta = 0.1$ and 0.15.



(a) Edit Distance (b) Q-gram Distance

Figure 5: Costs of different dimensionalities.

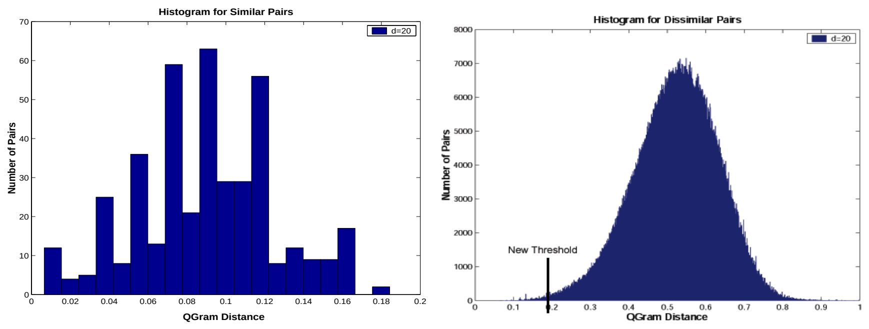
Figures 5 (a) and (b) show costs for different dimensionalities for edit distance and q-gram distance, respectively. (See Section 3.2 for the definition of "cost.") It is shown that the cost decreased with the increase of dimensionality. That is, the larger the dimensionality, the fewer extra object pairs we need to retrieve in step 2 for each similar-string pair, while the original strings of these objects have a distance greater than δ. On the other hand, due to the complexity of StringMap and the curse of dimensionality, d cannot be high either. The results show that $d = 20$ is a good dimensionality for both metrics.



(a) Similar pairs (b) Dissimilar pairs

Figure 6: Histograms of new Euclidean distances: edit distance, $\delta = 2$, $d = 20$.

Figures 6 (a) and (b) show the distributions of the object-pair distances after StringMap, for the edit-distance metric. We randomly sampled 2K from Source 1 to form two data sets with the same size. We chose $\delta = 2$ for similar-string pairs, and set $d = 20$. The figures show that after StringMap, there is a new threshold to differentiate similar pairs from *most* dissimilar pairs. In particular, all the sampled similar-string pairs had new object-pair distances within 5.5, while most of dissimilar-string pairs had their object-pair distances larger than 5.5.



(a) Similar pairs (b) Dissimilar pairs

Figure 7: Histograms of new Euclidean distances: q-gram distance, $\delta = 0.2$, $d = 20$.

Figures 7 (a) and (b) show the distributions of the object-pair distances after StringMap, for the q-gram distance metric. We chose $\delta = 0.2$ for similar-string pairs, and set $d = 20$. We have the similar observations as for the edit-distance metric. In particular, the new distances of all the similar-string pairs are within 0.2.

6.2 Choosing Threshold δ'

As discussed in Section 4.1, we selected new threshold δ' for the second step as follows. For each source, we randomly selected 2K strings, and partitioned the strings into two data sets equally. We used the nested-loop approach to find all the similar-string pairs in the selected strings. We ran StringMap with $d = 20$, and traced the new Euclidean distances of these sample similar-string pairs. We did this sampling 10 times, and chose δ' as the largest new object-pair distance of the sampled similar-string pairs.

	edit distance $\delta = 1$	edit distance $\delta = 2$	edit distance $\delta = 3$	q-gram distance $\delta = 0.2$
Source 1	4.5	5.9	7.735	0.2
Source 2	5.36	7.0	10	0.2

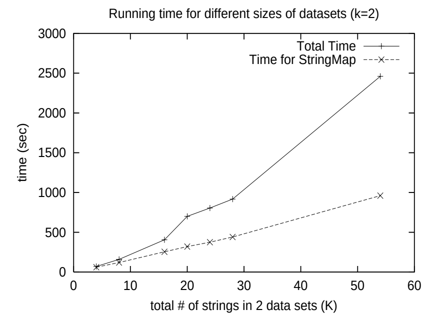
Table 2: Threshold δ' used in step 2 ($d = 20$).

Table 2 shows the δ' values used in step 2 for two different metrics. Notice that when using the edit-distance metric, since the two sources have different strings with different length distributions, it is not surprising that we need to choose different δ' values by sampling. For the q-gram metric, we set $\delta' = 0.2$ for both data sources 1 and 2.

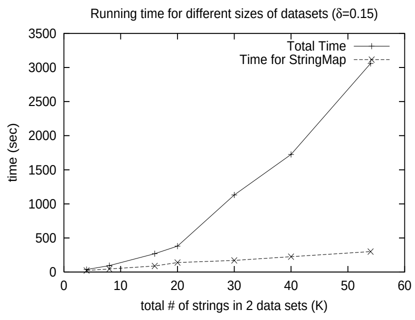
6.3 Running Time

In order to measure the performance of our approach, we ran our algorithm on different data sizes. In each case, we chose the same number of strings in both data sets. We chose dimensionality $d = 20$, and let the total size of strings vary from 2K, 4K, 8K, 16K, to 54K from Source 1. We measured the corresponding running time.

Figure 8(a) shows the time of the complete algorithm (including two steps), and the time of the StringMap step, assuming we use the edit-distance metric. Their gap is the time of the second step that did the R-tree similarity join. The figure shows that as the data sizes increased, both the StringMap time and the total time grew. Our approach is shown to be very efficient and scalable. For instance, when the total number of strings is $54K$, it took the approach only 41 minutes to find the similar-string pairs. (It look almost one week for the nested-loop approach to finish.) The figures also indicate that other similarity-join techniques may be used in the second step to improve its performance. Figure 8(b) shows the times if we used the q-gram distance. The times are similar to those of edit distance.



(a) Edit distance $\delta = 2$



(b) Q-gram distance $\delta = 0.2$

Figure 8: Running time (d=20).

In the case where the edit-distance metric is used, the approach in [14] can be used to find all the string pairs whose edit distance is within a given threshold. We implemented this approach using Oracle 8.1.7 on the same PC, and let the database use indexes to run the query. We selected subsets of strings from Source 1, and let both sets have the same number of strings. In our approach we chose threshold $\delta = 2$, dimensionality $d = 20$, and new threshold $\delta' = 5.9$. Figure 9 shows the performance difference between these two approaches. The figure shows that our approach can substantially reduce the time of finding similar-string pairs. Notice that even though our approach cannot guarantee to find all such pairs, as we will see shortly, it can still achieve a very high accuracy. In the context of record linkage, an efficient approximate-search algorithm with a very high accuracy might be more preferable.

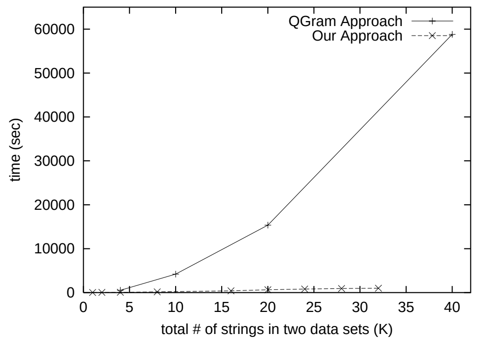
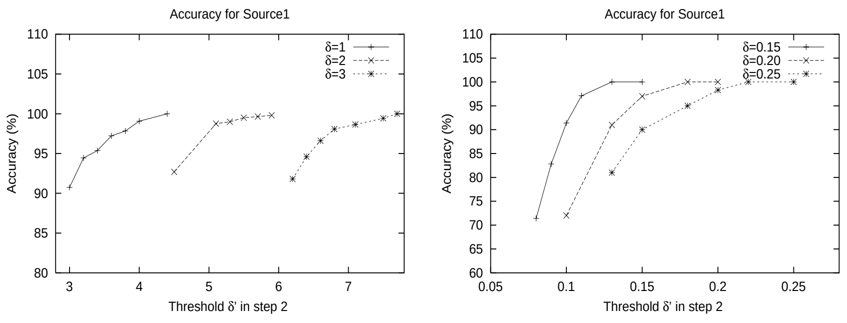


Figure 9: Our approach (approximate search) versus the qgram approach in [14] (exact search).

6.4 Accuracy

We want to know how well our approach can find all the similar-string pairs. (Ideally we want to find all of them!) In particular, we are interested in the accuracy, i.e., ratio of similar-string pairs found among all similar-string pairs.



(a) Edit Distance

(b) Q-gram distance

Figure 10: Accuracy versus threshold δ' (d=20).

Figure 10 (a) shows the accuracy of data source 1, with different threshold δ' values in the second step, using the edit-distance metric. As the δ' value increased, the accuracy also increased, and it quickly got very close to 100%. For instance, in the case where $\delta = 2$, the accuracy reached 99% when $\delta' = 5.6$. When we further increased the threshold, the accuracy continued to grow close to 100%. Figures 10 (b) shows similar accuracy results when we used the q-gram distance on data source 1.

We also implemented the sliding-window approach in [15]. Without loss of generality, we used attributes as keys in the sliding-window approach, and the condition is $\Delta_e(name) \leq 2$ for data source 1. We chose different window sizes, and for each of them, the time and accuracy were measured.

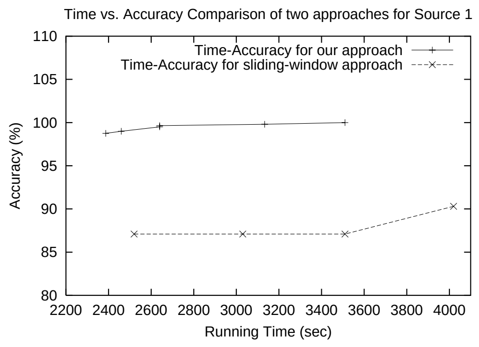


Figure 11: Our approach versus the sliding-window approach in [15].

Figure 11 shows the accuracy and time for these two approaches. It shows that our approach can achieve a very high accuracy given a time limit. The primary reason is that our mapping function provides very good distance/similarity preservation. Since lexicographic ordering does not preserve edit distances as well, the approach discussed in [15] needs to consider a very large window size (and hence cost) to obtain competitive degree of accuracy.

6.5 Results on Multiple Attributes

Now we report our experimental results for the multiple-attribute case on data source 3.

Single Conjunctive Clause

We evaluated the single conjunctive query Q_2 in Section 5.1. There are three attributes to perform a similarity join: *title*, *name*, and *year*. Our experimental results showed that the attribute *year* is not a good one to do a similarity join, since too many candidate pairs were generated. Thus we answered the query by conducting a similarity join using attributes *title* and *name*, respectively.

Similarity-join Attribute	Time (sec)	Final Result Size
name	1140	700
title	1540	702

Table 3: Results of similarity join using different attributes.

Table 3 gives the results. It is shown that for the thresholds in the query, doing a similarity join on *name* is more efficient than on *title*. Notice that the result size is different for these two similarity searches, since both of them are approximate. The small difference (only two pairs) between them again shows that our approach has a very high accuracy (more than 99%).

Disjunctive Clause

We used the query Q_1 in Section 2.2 as an example of disjunctive clauses. We implemented the two approaches in Section 5.2. For algorithm 1 that produces approach A, we chose the conjunct $\Delta_q(title) \leq 0.10$ to do the similarity search for the first clause, and conjunct $\Delta_e(name) \leq 2$ for the second one. After getting the result pairs for each clause, we took a union of the two sets and produced the final result. Table 4 shows the results. The total time for approach A was 1820 seconds, and the size of the final result set was 619. Notice that the results of the two clauses had overlapped record pairs. The accuracy of taking this approach is more than 99%.

Selected Condition	Time (sec)	Number of Record Pairs
Clause 1, $\Delta_q(title) \leq 0.10$	1060	406
Clause 2, $\Delta_e(name) \leq 2$	760	518
Total	1820	619 (pairs above overlap)

Table 4: Result of approach A for disjunctive clause.

For algorithm 2 that produces approach B, we found that $\Delta_q(title) \leq 0.15$ was the most selective conjunct, using which we performed the similarity join. Table 5 shows the results. In particular, the total time for approach B was 1743 seconds, and the size of the final result set was also 619.

Selected Condition	Time (sec)	Number of Record Pairs
Clause 2, $\Delta_q(title) \leq 0.15$	1743	619

Table 5: Result of approach B for disjunctive clause.

For this example, approach B was better than approach A. In general, algorithm 1 produces a solution that tries to minimize the time of each individual similarity join, which tends to produce a small candidate set. Algorithm 2 produces a solution that needs to perform a similarity join only once for all the clauses. It may need more time for the single similarity join, since the threshold could be large. To fix this problem of Algorithm 2, we can set an upper bound on the threshold of each attribute, and we will never consider those conjuncts on this attribute whose threshold is above this bound. One observation is that if the thresholds for the attributes vary a lot, it could be better to take Algorithm 1. Otherwise, Algorithm 2 could be preferable.

7 Conclusion

In this paper we developed a novel approach to the record-linkage problem: given two lists of records, we want to find similar record pairs, where the overall similarity between two records is defined based on domain-specific similarities over individual attributes. For each attribute of records, we first map records to a multidimensional Euclidean space that preserves domain-specific similarity. Given the merging rule that defines when two records are similar, a set of attributes are chosen along which the merge will proceed. A multidimensional similarity join over the chosen attributes is used to determine similar pairs of records. Our extensive experiments using real data sets showed that our solution has very good efficiency and accuracy. In addition, our approach is very extendable, since many existing techniques can be used. It can also be generalized to other similarity functions between strings.

Acknowledgments

We thank Michael Ortega-Binderberger for providing his implementation of the algorithm in [16] and many fruitful discussions. We thank the authors of [14] for providing the details of their implementation. We thank Heikki Mannila for letting us know the work in [29].

References

[1] K. Alsabti, S. Ranka, and V. Singh. An efficient parallel algorithm for high dimensional similarity join. In *IPPS: 11th International Parallel Processing Symposium*. IEEE Computer Society Press, 1998.

[2] P. M. Aoki. Algorithms for index-assisted selectivity estimation. In *ICDE*, page 258, 1999.

[3] R. E. Bellman. *Adaptive Control Processes: A Guided Tour.* Princeton University Press, Princeton, New Jersey, U.S.A., 1961.

[4] M. Bilenko and R. J. Mooney. Learning to combine trained distance metrics for duplicate detection in databases. Technical report, Technical report, Computer Science Dept., University of Texas, Austin, 2002.

[5] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.

[6] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.

[7] W. W. Cohen, H. A. Kautz, and D. A. McAllester. Hardening soft information sources. In *Knowledge Discovery and Data Mining*, pages 255–259, 2000.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Presss, 1990.

[9] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.

[10] V. C. (Editor) and L. D. B. (Editor). *Image Databases: Search and Retrieval of Digital Imagery.* John Wiley and Sons, 2001.

[11] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid. Tailor: A record linkage toolbox. In *ICDE*, 2002.

[12] C. Faloutsos and K.-I. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In M. J. Carey and D. A. Schneider, editors, *SIGMOD*, pages 163–174, 1995.

[13] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.

[14] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[15] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In M. J. Carey and D. A. Schneider, editors, *SIGMOD*, pages 127–138, 1995.

[16] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In L. M. Haas and A. Tiwary, editors, *SIGMOD*, pages 237–248, 1998.

[17] G. R. Hjaltason and H. Samet. Contractive embedding methods for similarity searching in metric spaces. Technical report, University of Maryland Computer Science, 2000.

[18] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical Report 99-50, Rutgers Univ., 8 1999.

[19] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. A cost model for estimating the performance of spatial joins using r-trees. In *Statistical and Scientific Database Management*, pages 30–38, 1997.

[20] P. Indyk. Sublinear time algorithms for metric space problems. In *STOC*, pages 428–434, 1999.

[21] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

[22] N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE*, pages 466–475, 1998.

[23] J. B. Kruskal and M. Wish. *Multidimensional Scaling.* Sage Piblications, Beverly Hills, CA, 1978.

[24] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[25] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[26] D. Loshin. Value added data: merge ahead. *Intelligent Enterprise*, 3(3), 2000.

[27] W. L. Low, M. L. Lee, and T. W. Ling. A knowledge-based framework for duplicates elimination. *Information Systems: Special Issue on Data Extraction, Cleaning and Reconciliation*, 26(8), 2001.

[28] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *SIGMOD*, pages 1–12, 1999.

[29] H. Mannila and J. K. Seppnen. Recognizing similar situations from event sequences. In *First SIAM Conference on Data Mining*, 2001.

[30] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.

[31] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[32] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *ICDE*, pages 301–311, 1997.

[33] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD*, pages 343–354, 2000.

[34] J. T.-L. Wang, X. Wang, K.-I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *Knowledge Discovery and Data Mining*, pages 307–311, 1999.

[35] W. Winkler. Advanced methods for record linkage, 1994.

[36] P. N. Yianilos and K. G. Kanzelberger. The LIKEIT intelligent string comparison facility. Technical report, NEC Research Institute, 1997.

[37] F. W. Young and R. M. Hamer. *Multidimensional Scaling: History.* Theory and Applications Erlbaum, New York, 1987.