# Supporting Efficient Record Linkage for Large Data Sets Using Mapping Techniques*

Chen Li, Liang Jin, and Sharad Mehrotra
ICS 424B, University of California, Irvine, CA 92697, USA
chenli@ics.uci.edu, 1-949-824-9470 (telephone), 1-949-824-4056 (fax)

**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. The record-linkage problem arises naturally in the context of data cleansing that usually precedes data analysis and mining. Since the scalability issue of record linkage was addressed in [21], 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 [16] 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 over the chosen attributes is used to find similar pairs of records. Our extensive experiments using real data sets show that our solution has very good efficiency and recall.

**Keywords: Data Cleaning, Record Linkage, Similarity Search, StringMap**

*Part of this article was published in [28]. In addition to the prior materials, this article contains more analysis, a complete proof, and more experimental results that were not included in the original paper.

# 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, especially those on the Web. Databases frequently contain approximately duplicate fields and records that refer to the same real-world entity, but are not identical. As the following examples illustrate, 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 [5].

**EXAMPLE 1.1** The DBLP bibliography server [33] has the following two pages for authors: "Sang K. Cha"[1] and "Sang Kyun Cha"[2]. They are referring to exactly the same person in reality. However, due to the abbreviation of the middle name, the publications of the same person are split into two pages. This problem exists in many web sources and databases. □

**EXAMPLE 1.2** A hospital at a medical school has a database with thousands of patient records. 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, or whether a patient's telephone number has changed. 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 record linkage (a.k.a data linkage) in a variety of data-analysis applications, developing effective and efficient techniques for record linkage has emerged as an important problem [38]. It is further evidenced by the emergence of numerous organizations (e.g., Trillium, FirstLogic, Ascential Software, 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 [10, 48]. 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 the 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 and the type of the data is string. However, as the examples above illustrated, domain-specific similarity measures (e.g., different functions 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 [5, 10, 48].

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 [42, 15, 17, 40] 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

[1] `http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/c/Cha:Sang_K=.html`

[2] `http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/c/Cha:Sang_Kyun.html`

pair of records. This approach is computationally prohibitive as the two data sizes become large. The scalability issue of record linkage has previously been studied in [21]. Since the original work in [21], 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 [31, 50, 16, 6, 25, 46, 47] have been developed. Furthermore, many efficient multidimensional similarity joins have been studied [1, 7, 13, 30, 43, 32, 39, 45, 44]. In this paper, we develop an efficient strategy by exploiting these advances to solve the record-linkage problem. In particular, 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. In general, many mapping techniques can be used. As an example, in this paper we focus on the FastMap algorithm [16] 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 chosen closely related to the old threshold of string distances and the mapping step. See Section 4 for details. Again, many similarity-join algorithms can be used in this step. In this paper we use the algorithm proposed by Hjaltason and Samet [23] 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 query threshold, and find those similar-record pairs.

We next study mechanisms for record linkage when multiple 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 from, for example, a classifier such as a decision tree after being trained with certain number of records using a labeled training set [42]. Given a set of merging rules, 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 (e.g., running time).

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 records. (3) Our extensive experiments using real large data sets show that this approach has very good efficiency and recall (greater than $99\%$).

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.

### 1.1 Related Work

Record linkage was studied in [21, 22] (called the "Merge/Purge" problem). Their proposed approach first merges two given lists of records, then sorts the records based on lexicographic ordering for each attribute (or a "key" as defined in that paper). A fixed-size sliding window is applied, and the records within a window are checked to determine if they are duplicates using a merging rule. 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 this 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 recall and performance. In addition, it might not be easy to choose good "keys" to bring similar records close to each

other. Sometimes it requires specific domain knowledge to design the "key" construction.

[20] is tackling the problem of approximate whole-string matching inside a commercial DBMS. Their proposed approach can guarantee no false dismissal. Their approach primarily focuses on the edit-distance metric. In addition, the use of a DBMS could introduce extra storage and execution-time overhead. [8] incorporates the TF/IDF [3] idea from the information retrieval community to define record similarities. They split records into tokens, and consider token frequencies when calculating the similarity/distance between two records. They also developed an error tolerant index to identify the closest fuzzy matching tuples with high probability. Lee et al. developed a data cleansing system called "IntelliClean" [34]. In [35] they present several efficient techniques to pre-process records before trying to find similar pairs. After these prepocessing steps a key problem is still how to find those similar pairs using merging rules efficiently, which is the focus of this work.

One key challenge in record linkage is to develop good merging rules to identify duplicate records. Recently researchers are trying to apply machine-learning techniques to solve the record-linkage problem [42, 5]. These techniques can be used to generate merging rules, and in this work we focus on how to evaluate given such merging rules.

# 2 Problem Formulation

In this section, we formulate the record-linkage problem.

## 2.1 Distance Metrics of Attributes

We first introduce the concept of $metric$ distances. Let set $\Psi$ be the universe of the objects in a database. A function denoting a measure of the distance between two objects

$$d : \Psi \times \Psi \rightarrow R$$

is a $metric$ if it satisfies the following properties:

1. **Positiveness:** $\forall x, y \in \Psi, d(x, y) \geq 0$;
2. **Symmetry:** $\forall x, y \in \Psi, d(x, y) = d(y, x)$;
3. **Reflexivity:** $\forall x \in \Psi, d(x, x) = 0$;
4. **Triangular Inequality:** $\forall x, y, z \in \Psi, d(x, y) \leq d(x, z) + d(z, y)$.

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 between a value of $R.A_j$ and a value of $S.A_j$.[3] 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 [37], 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,

[3] In [21], "keys" constructed from multiple fields are used to represent the similarity between records. These keys can be viewed as attributes in our setting.

$\Delta_e$("Harrison Ford", "Harison Fort") = 2.

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, $\Delta_e$("Jack Lemmon","Jack Lemon") = 1, and $\Delta_e$("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 [49].

There are many variations of the original edit-distance metric, such as allowing transposition of two characters, allowing "block move" [12] (i.e., move of multiple characters in one step), and assigning different weights to different characters and operations.

The *Jaccard coefficient distance* [11, 29] is another possible metric. Let $q$ be an integer. Given a string $s$, 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, if $q = 2$:

$G$("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 *Jaccard Coefficient Distance* between two strings $s_1$ and $s_2$, denoted $\Delta_j(s_1, s_2)$, is defined as:

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

For example, $\Delta_j$("Harrison Ford", "Harison Fort") = 1 - $\frac{10}{13} \approx 0.23$. The Jaccard coefficient distance is a metric, and the square root of Jaccard coefficient distance is Euclidean [19]. Clearly the smaller the Jaccard coefficient distance between two strings is, the more similar they are.

## 2.2 Similarity Merging Rules

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 [36]. 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 is consistent with the merging functions considered in [21].

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

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

For each conjunct $M_j(A_j) \le \delta_{i,j}$ ($i = 1, \ldots, k$, 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 $M_j(r.A_j, s.A_j) \le \delta_{i,j}$.

For instance, given three attributes about papers, *title*, *author*, and *year*, suppose we use the edit distance $\Delta_e$ as a metric for attributes *author* and *year*, and the Jaccard coefficient distance function $\Delta_j$ as a metric for

attribute $title$. Then we could have the following rule:

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

The record-linkage problem studied in this paper is the following.

> Given two relations $R$ and $S$, and the merging rules defined above, find pairs of records $(r, s)$ from relations $R$ and $S$, such that each pair satisfies the merging rule.

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 $10,000$ names, assuming the edit-distance function is used.

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 real matching pairs (but with a high recall) might be more preferable.

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

Table 1: Symbols used in this paper.

| Symbol | Meaning |
|---|---|
| $R$, $S$ | two relations for record linkage |
| $M$ | metric function in the original space |
| $\Delta_e$ | edit distance |
| $\Delta_j$ | Jaccard coefficient distance function |
| $\delta$ | threshold in the original string metric space |
| $\delta'$ | new threshold in the mapped Euclidean space |
| $d$ | dimensionality of the Euclidean space |

## 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 $\delta$, 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 $\delta$, 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 $O_1, \ldots, O_N$ in a metric space, find $N$ points $P_1, \ldots, P_N$ in a $d$-dimensional Euclidean space, such that the distances are maintained as well as possible.

One way to describe how well the distances are maintained is to use the following $stress$ function:

$$stress^2 = \frac{\sum_{i,j}(\hat{d}_{ij} - d_{ij})^2}{\sum_{i,j} d_{ij}^2} \tag{1}$$

in which $d_{ij}$ is the dissimilarity measure between object $O_i$ and object $O_j$ in the old metric space, and $\hat{d}_{ij}$ is the Euclidean distance between their image objects in the new space. This function gives the relative error of the mapping. Ideally the mapping algorithm can make $stress$ as small as possible.

Many mapping/embedding functions can be used, such as multidimensional scaling [31, 50], FastMap [16], Lipschitz [6], SparseMap [25], L1-Mapping [46], and MetricMap [47]. These algorithms have different properties in terms of their efficiency, contractiveness, and stress. (See [24] for a good survey.) In this paper we use the FastMap algorithm because of its good time efficiency and distance-preserving capability. Here we briefly review the algorithm. (See [16] for the details.) 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 distances. It also computes the new distances after the projections.

We modify the FastMap slightly and propose an algorithm called "StringMap," as shown in Fig. 1 (a). StringMap removes the recursion in FastMap, and computes a distance between objects only when it becomes needed. In particular, 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 (ideally the farthest pair). As we cannot afford a quadratic approach to find the exact farthest pair, we adopt the linear heuristic algorithm in [16]. The function $ChoosePivot()$ iterates $m$ times to find the pivots. A typical $m$ value could be $5$ (as in [16]). 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

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

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

There are few observations about the algorithm.

1. In the last line of the algorithm, the computation of $coord[i,h]$ is not symmetric with respect to values $x$ and $y$.
2. In function $GetDistance()$, it is known that $dist * dist - w * w$ can be negative [47] . In StringMap, we take the square root of the *absolute* value to compute the new distance. See [24] for other ways to deal with this case.

**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) {
    (p1, p2) = ChoosePivot(h,M); // choose pivot strings
    PA[1,h]= p1; PA[2,h] = p2; // store them
    dist = GetDistance(p1, p2, 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], p1, h, M);
        y = GetDistance(t[i], p2, 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 sa = a random string from t[1], . . . , t[N];
    for (i = 1 to m) { // a typical m value could be 5
        // use function GetDistance(.,.,h,M)
        // to compute distances
        seed sb = a farthest point from sa;
        seed sa = a farthest point from sb;
    }
    return (sa, sb);
}

// 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 × dist − w × w|);
    }
    return ( dist );
}
```

(b) Functions

Figure 1: Algorithm StringMap.

All the steps in the StringMap algorithm are linear on the number of strings $N$. In particular, consider the $h$-th step of the algorithm. A call to function GetDistance(.,.,h) takes $O(h)$ time. Here we assume that it takes $O(1)$ time to compute $M(a, b)$, and $O(1)$ time to compute the $dist$ value in each iteration. Therefore, for each $h$ value, it takes

$$O(h \times 2 \times m \times N) \tag{2}$$

time to find two pivot seeds, and $O(h \times N)$ time to compute the coordinates of all the strings in the $h$-th dimension. The complexity of the $h$-th step is:

$$O(h \times 2 \times m \times N + h \times N) = O(h \times m \times N)$$

Thus the complexity of the StringMap algorithm is:

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

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 [27] 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 [4, 14]. 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. Select a set of string pairs from data sets $R$ and $S$. (See Section 4.2 for details.) Use the nested-loop approach to find all similar-string pairs within threshold $\delta$ in the selected string pairs.
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*:

$$cost = \frac{\text{\# of object pairs within distance } w}{\text{\# of similar-string pairs}} \tag{3}$$

   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 $\delta'$ 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 $\delta'$ will be retrieved in step 2 and need to be pruned out by post checking according to the merging rules. Thus the lower the cost is, relatively the fewer object pairs need to be retrieved in step 2 whose original distance is more than $\delta$. Fig. 5 in Section 6.2 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 $\delta'$. For each candidate pair, we check the distance of their original strings to see it is within the original threshold $\delta$. In this section we study how to select the new threshold and how to do the similarity join.

### 4.1 Choosing New Threshold $\delta'$

The selection of the new threshold $\delta'$ depends on the mapping function. For instance, we can set $\delta' = \delta$ if the mapping function is *contractive* [9]. 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]

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

[4]This equation measures the *distortion* of the mapping [24].

Then we can just set $\delta' = c_2 \cdot \delta$. Properties of different mapping functions are studied in [24].

Ideally, the threshold $\delta'$ 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, we can choose the threshold $\delta'$ as follows. We select a small number of string pairs from data sets $R$ and $S$. (See Section 4.2 for details.) 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 pairs, and compute their new Euclidean distances after StringMap. We choose their maximal new distance as the new threshold $\delta'$. We may do this sampling multiple times (on different sets of selected strings), and choose $\delta'$ as the maximal new distance of those similar-string pairs. By doing this sampling process for multiple times, we increase the opportunity that the new threshold $\delta'$ can *capture* the similar-string pairs as many as possible.

### 4.2 Selecting String Pairs via Sampling

In order to decide the dimensionality $d$ and the new threshold $\delta'$, we need to obtain a sample set of string pairs from the two data sets $R$ and $S$. The sample set should have enough string pairs whose original distances are within $\delta$. Now we consider several ways to choose the sample set.

- *Double Random Sampling*: We sample the strings $r$ and $s$ *randomly* from $R$ and $S$ respectively, and find similar string pairs from the sampled strings.
- *Single Random Sampling*: We sample a certain number of strings randomly from one data set, say, $R$. For each of them $r$, we find similar strings $s$ in $S$ and produce pairs of $(r, s)$.
- *Sorted Sampling*: We sort the strings in $S$ lexicographically. We sample a certain number strings randomly from $R$. For each of them $r$, we locate it in the sorted list of $S$, and find its nearby, e.g., 100, strings $s$. We produce pairs of $(r, s)$.

We will show in Section 6 that Sorted Sampling produces the best new threshold among these methods. That is, its estimated new threshold is the closest to the real new threshold $\delta'$. The reason is that the lexicographical order of $S$ strings improves the chance for catching similar string pairs from.

### 4.3 Finding Object Pairs within $\delta'$

We want to find all those object pairs whose new distance is within this new threshold $\delta'$. Similarity joins over multidimensional spaces have been studied in [1, 7, 13, 30, 43, 32, 39, 45, 44]. Many algorithms can be used in this step. In this paper we use a simplified version of the algorithm in [23] as an example. We could instead have chosen any of those algorithms for our purpose. We chose the algorithm in [23] 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.

Here we will briefly explain the main idea of the algorithm. (See [23] 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

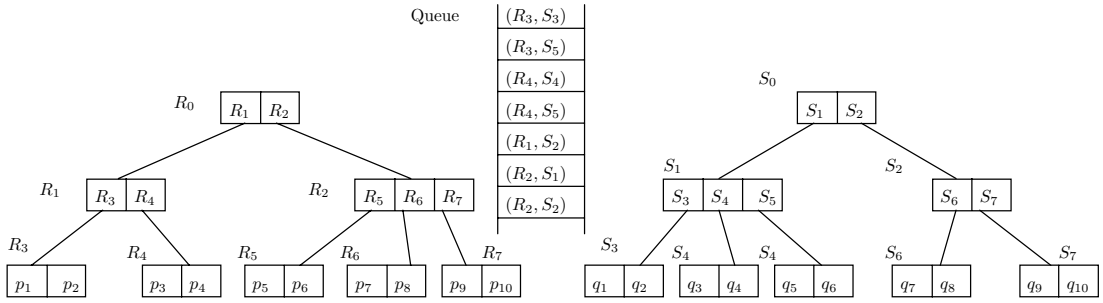



Figure 2: Finding similar-object pairs using R-trees.

nodes to find those pairs of objects within distance $\delta'$. 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. The lower bound of the distance between the node pairs must be within the new threshold $\delta'$. Those node pairs that cannot produce results are pruned in the traversal, i.e., they are never inserted back into the queue.

Take Fig. 2 as an example. Initially, a pair of the root nodes $(R_0, S_0)$ is inserted into the 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 [41] 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 $\delta'$, 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 $\delta'$. On the other hand, if the distance of two nodes is within $\delta'$, 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 $\delta'$, 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 $\delta'$, we will not consider this pair, since they cannot generate object pairs that have a distance within $\delta$. In other words, all the pairs of their descendants are greater than $\delta'$.

Suppose only the following pairs have a MINDIST distance within $\delta'$:

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

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 $\delta$. If so, we compute the metric distance of their original strings. We output this pair of strings if their metric distance is within the original threshold $\delta$.

[5] Since we just want to find those object pairs whose distance is within $\delta'$, we do *not* need a *priority* queue. A priority queue based on object-pair distances is necessary in [23], since they want to find the "all pair top-k" object pairs with the smallest distances.

### 4.4 Traversal Strategies

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. These observations are consistent with those in [23].

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.

## 5 Combining Multiple Attributes

So far we have studied the record-linkage problem for the single attribute case. 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.

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

We could first do a similarity search to find all the record pairs that satisfy the first condition, $\Delta_j(title) \leq 0.15$. 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, 26]), 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 similarly to [21], we could also search along multiple attributes of the conjunction to improve the recall. Since the mapping in Step 1 does not guarantee that all the relevant string pairs will be found, using multiple attributes may improve recall. 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 recall (over 99%), after doing a join based on the condition of one attribute, we can postprocess the candidate record pairs by checking the remaining conditions. Thus the high-recall property can help to reduce the running time of a conjunctive clause with multiple attributes.

### 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 $\Delta_j(title) \leq 0.10$. 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 $\Delta_j(title) \leq 0.15$ in the second conjunctive clause. These pairs also include all the pairs satisfying the first conjunctive clause, since $\Delta_j(title) \leq 0.10$ implies $\Delta_j(title) \leq 0.15$. Among all these pairs, find those satisfying the merging rule.

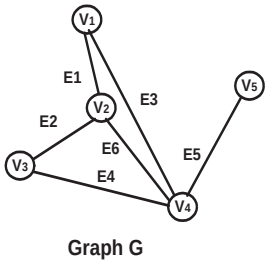
Approach A needs to do two similarity searches, while approach B requires only one. However, both similarity searches in approach A could be more selective than the single similarity search in approach B, thus require less combined running time than approach B. Which approach is better depends on the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

The example shows that to answer a disjunct (in the form of a conjunctive clause), we can choose at most one conjunct in it to do a similarity join. After we find the candidate pairs using similarity join over that conjunct, we assume that we can get the answer for this disjunct for free because the step of testing other attributes takes much much less time than the similarity join, as will be shown in experimental section. 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.

**Theorem 5.1** *Assuming the time of doing a similarity search for each conjunct is given, the problem of finding an optimal solution (i.e., a plan with a minimum total running time for the similarity joins) to answer the query is $\mathcal{NP}$-hard.* □

**Proof:** We reduce the Vertex Cover problem [18] to our problem. Since the Vertex Cover problem is $\mathcal{NP}$-complete, our problem is $\mathcal{NP}$-hard.

Given a graph $G$ with $n$ vertices $V_1, \ldots, V_n$, we construct a query on $n$ attributes, with $n$ conjuncts $M_1 \leq \delta_1, \ldots, M_n \leq \delta_n$, where each $\delta_i$ is a nonzero constant. For all $1 \leq i \leq j \leq n$, if $V_i$ and $V_j$ are connected by an edge $E$ in $G$, we construct a conjunctive clause $C_E$ in the form of $M_i \leq \delta_i \ \wedge \ M_j \leq \delta_j$. If the graph $G$ has $m$ edges, we get a merging rule with $m$ conjunctive clauses, each of which has two conjuncts. Fig. 3(a) shows a sample graph $G$ with 5 vertices and 6 edges, and Fig. 3(b) is the merging rule constructed from $G$.




(a) Sample graph $G$.

$$
\begin{array}{llllll}
 & M_1 \leq \delta_1 \wedge & M_2 \leq \delta_2 & & & \\
\vee & & M_2 \leq \delta_2 \wedge & M_3 \leq \delta_3 & & \\
\vee & M_1 \leq \delta_1 \wedge & & & M_4 \leq \delta_4 & \\
\vee & & & M_3 \leq \delta_3 \wedge & M_4 \leq \delta_4 & \\
\vee & & & & M_4 \leq \delta_4 \wedge & M_5 \leq \delta_5 \\
\vee & & M_2 \leq \delta_2 \wedge & & M_4 \leq \delta_4 &
\end{array}
$$

(b) Merging rule converted from the sample graph $G$.

Figure 3: Sample graph and the corresponding merging rule.

The above construction of the merging rule takes time that is polynomial in the size of $G$. Now, we show that $G$ has a vertex cover of size $k$ if and only if we can pick $k$ different conjuncts in the merging rule to answer

the query, with the total cost of $k \times C$, where $C$ is the cost of doing one similarity search on a single conjunct. Here all the conjuncts have the same cost to evaluate.

For the "only if" part, suppose $G$ has a vertex cover $S$ of size $k$. For each vertex $V_i$ in $S$, we evaluate one of the $M_i \leq \delta_i$ conjuncts. Since all the edges are covered by the vertices in $S$, all the conjunctive clauses in the merging rule can be answered either by being evaluated, or by doing post-processing. The cost of this evaluation plan is $k \times C$.

For the "if" part, suppose there is a plan to answer the query, which picks $k$ conjuncts to evaluate and has a total cost of $k \times C$. Without loss of generality, let them be $M_1 \leq \delta_1, \ldots, M_k \leq \delta_k$. We pick the corresponding $V_1, \ldots, V_k$ to form a cover set $S$. For each $M_i \leq \delta_i$, we can do post-processing, and get the answers for all the conjunctive clauses that contain $M_i \leq \delta_i$. Since the plan can compute the answers for all the conjunctive clauses, the corresponding vertices must cover all the edges in the graph $G$. So $S$ is a vertex cover of graph $G$ with $k$ vertices. ∎

Notice the proof also shows that the problem of finding optimal solution to answer the query is $\mathcal{NP}$-hard in terms of the number of *distinct* conjuncts.

When the number of distinct conjuncts is small, we can exhaustively search among all the combinations of the conjuncts to find an optimal plan. In the case where this number is large, we propose three heuristic-based greedy algorithms for finding a good solution to evaluate a merging rule.

- 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.

- Algorithm 3: For each conjunct, we associate it with the cost as the running time required to evaluate the conjunct, and the benefit as the number of conjunctive clauses it can cover. A conjunct $M_j(A_j) \leq \delta_{i,j}$ covers those conjuncts on the same attribute with a threshold within $\delta_{i,j}$ (including itself), and those conjuncts in their conjunctive clauses. We greedily choose a conjunct with the largest "benefit/cost" ratio, and add this conjunct to the plan. We remove the conjunct and its covered conjuncts, and repeat the process until all the conjuncts are covered. (An alternative to define the benefit of a conjunct is to use the total running time of the covered conjuncts.)

For instance, consider the query $Q_1$ in Section 2.2. Suppose Algorithm 1 chooses $\Delta_j(title) \leq 0.10$ 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 $\Delta_j(title) \leq 0.15$ 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 $12$.

2. **Source 2** is from the Die Vorfahren Database, a database of mostly Pomeranian surnames and geographic locations.[7] The database as of 2001 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.

3. **Source 3** is from the publications in DBLP.[8] We randomly selected $20,000$ papers in the proceedings. We use this data source to show how to do data linkage in multiple-attribute cases. The lengths of titles are less than $300$, and their mean length is around $70$. The lengths of author names are less than $50$, and their mean length is around $15$. The published years range from $1970$ to $2002$, with an average of $1995$.

For each dataset, we introduced about $10\%$ duplicate records by slightly modifying the values of randomly selected records. The errors introduced to string values consisted of character insertions, deletions, and substitutions. The errors for the numeric attributes were additions and subtractions to their numeric values.

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 compiler is gnu C++ running in cygwin. We used $8,192$ as the page size to build R-trees. Most of our experimental results are similar for three sources.

### 6.1 Nested-Loop and Our Approach

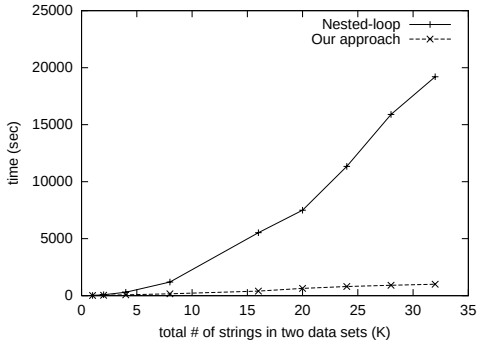



Figure 4: Comparing nested-loop with our approach.

Fig. 4 shows the performance difference of the nested-loop approach and our approach. We selected subsets of strings from Source 1, and let both sets have the same number of strings. In our approach we used the edit distance as the distance function, and chose threshold $\Delta_e = 2$, dimensionality $d = 20$, and new threshold $\delta' = 6.3$. The $x$-axis is the *total* number of strings from both sets. The $y$-axis is the running time in seconds. The figure shows that our approach can substantially reduce the time of finding similar-string pairs. For instance, when each data set had $16,000$ strings, it took the nested-loop approach about $19,200$ seconds ($5$ hours $20$ minutes), while it took our approach only about $1,000$ seconds (less than $17$ minutes).

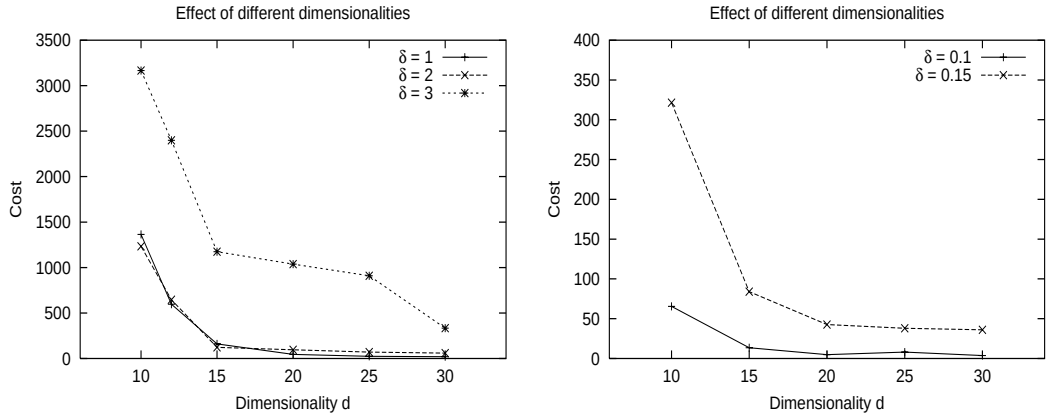
[6] http://www.imdb.com/

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

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

## 6.2 Choosing Dimensionality $d$

As discussed in Section 3.2, we need to choose a good dimensionality $d$ for the StringMap algorithm. We used Source 1 as an example. To select a $d$ value, we randomly sampled $2,000$ strings from $R$. For each of them, we located its lexicographical position in $S$, and paired it with the nearby $100$ strings. We used the sampled string pairs to measure the "cost" of different $d$ values (Section 3.2). For edit distance, we considered the case where $\delta = 1$, 2, and 3. For the Jaccard metric, we considered $\delta = 0.1$ and $0.15$.




(a) Edit Distance (b) Jaccard Distance

Figure 5: Costs of different dimensionalities.

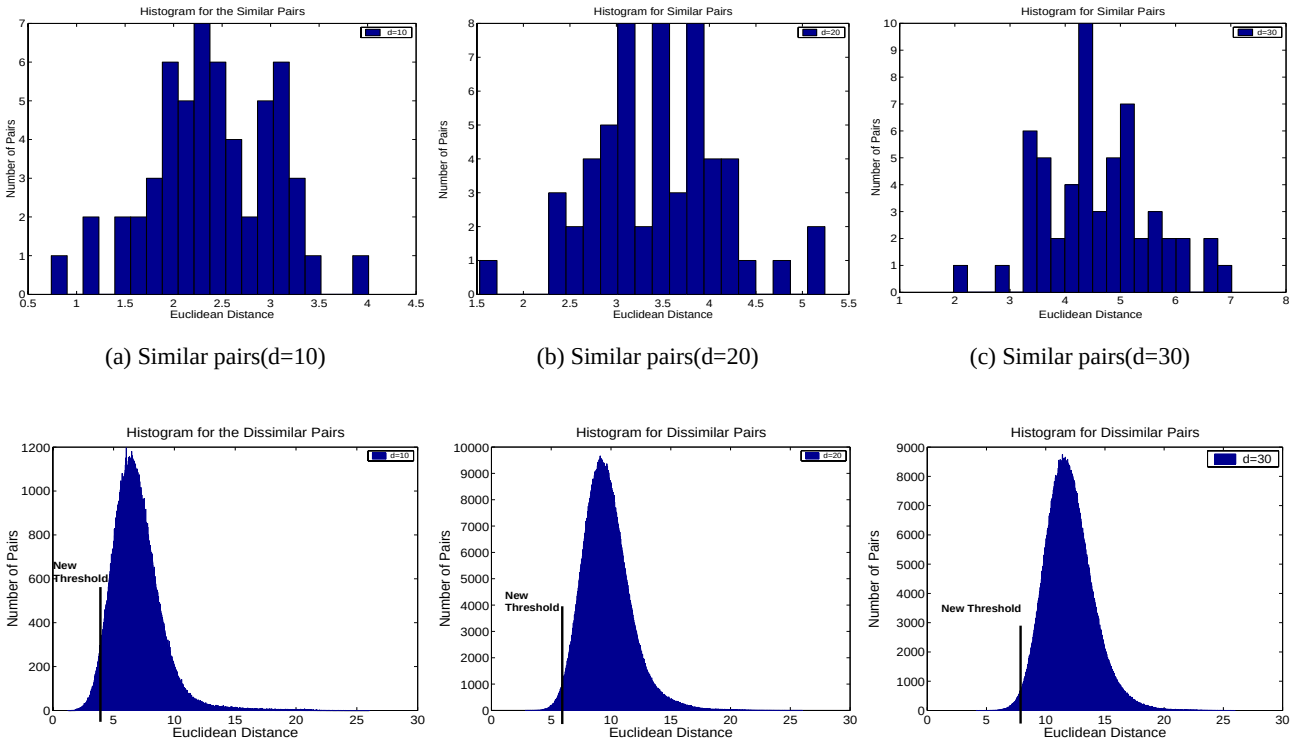
Fig. 5(a) and (b) show costs for different dimensionalities for edit distance and Jaccard distance, respectively. (See Section 3.2 for the definition of "cost.") It is clear that the cost decreased with the increase of the dimensionality. That is, the larger the dimensionality is, the fewer extra object 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 $\delta$. 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.

Fig. 6 shows the distributions of the object-pair distances after StringMap, for the edit-distance metric. We constructed the sample set from Source 1. We chose $\delta = 2$ for similar-string pairs, and set $d = 10, 20$, and $30$, respectively. 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 $5.5$ in the case when $d = 20$. Figures for other two dimensionalities exhibit the similar existence of such a new threshold.

Fig. 7(a) and (b) show the distributions of the object-pair distances after StringMap, for the Jaccard 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 $0.2$.

## 6.3 Choosing Threshold $\delta'$

As discussed in Section 4.1, we selected the new threshold $\delta'$ for the second step as follows. We constructed the sample set as described in Section 4.2. We ran StringMap with $d = 20$, and traced the new Euclidean distances of these sampled similar-string pairs. We did this sampling $10$ times, and chose $\delta'$ as the largest new object-pair distance of the sampled similar-string pairs. We evaluated the three sampling approaches, and compared




(a) Similar pairs(d=10) (b) Similar pairs(d=20) (c) Similar pairs(d=30)

(d) Dissimilar pairs(d=10) (e) Dissimilar pairs(d=20) (f) Dissimilar pairs(d=30)

Figure 6: Histograms of new Euclidean distances for different $d$'s: edit distance, $\delta = 2$.

them against the real $\delta'$ obtained by the nested-loop approach, which is the largest Euclidean distance of the similar string pairs. Table 2 shows the results. Among the three approaches, Sorted Sampling produced the new threshold that is closest to the real $\delta'$. Thus we used Sorted Sampling in other experiments.

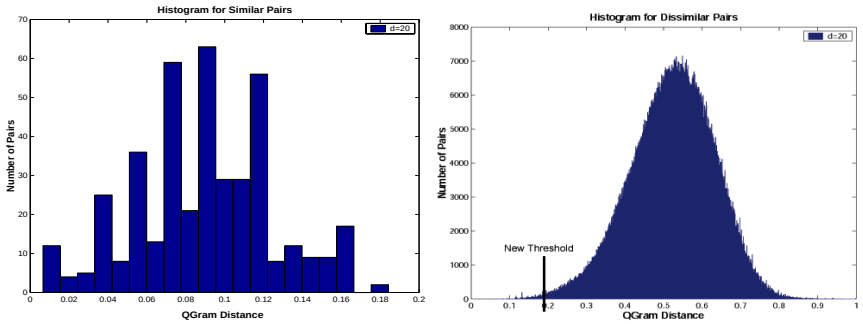
Table 2: Comparison of different sampling methods

| Data Source | Original Threshold | DoubleRandom | SingleRandom | SortedRandom | Real New Threshold |
|---|---|---|---|---|---|
| Source 1 | $\delta = 2$ | 5.8 | 5.9 | 6.3 | 6.6 |
| | $\delta = 3$ | 6.3 | 6.7 | 7.3 | 7.7 |
| Source 2 | $\delta = 2$ | 6.8 | 6.9 | 7.2 | 7.4 |
| | $\delta = 3$ | 8.6 | 9.1 | 9.3 | 10.0 |

Table 3 shows the $\delta'$ values used in step 2 for two different metrics. Notice that when we are using the edit-distance metric, since the two sources have different strings with different length distributions, it is not surprising that their $\delta'$ values from the sampling step are different. For the Jaccard metric, we set $\delta' = 0.2$ for both data sources 1 and 2.

## 6.4 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 for Source 1 vary from $2,000$ to $54,000$. We measured the corresponding running time for $\delta = 1, 2,$




(a) Similar pairs (b) Dissimilar pairs

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

Table 3: Threshold $\delta'$ used in step 2 ($d = 20$).

| | edit distance $\delta = 1$ | edit distance $\delta = 2$ | edit distance $\delta = 3$ | Jaccard distance $\delta = 0.2$ |
|---|---|---|---|---|
| $\delta'$ (Source 1) | 4.5 | 6.3 | 7.5 | 0.2 |
| $\delta'$ (Source 2) | 5.36 | 7.3 | 9.6 | 0.2 |

and 3 for the edit distance metric and $\delta = 0.2$ for the Jaccard metric.

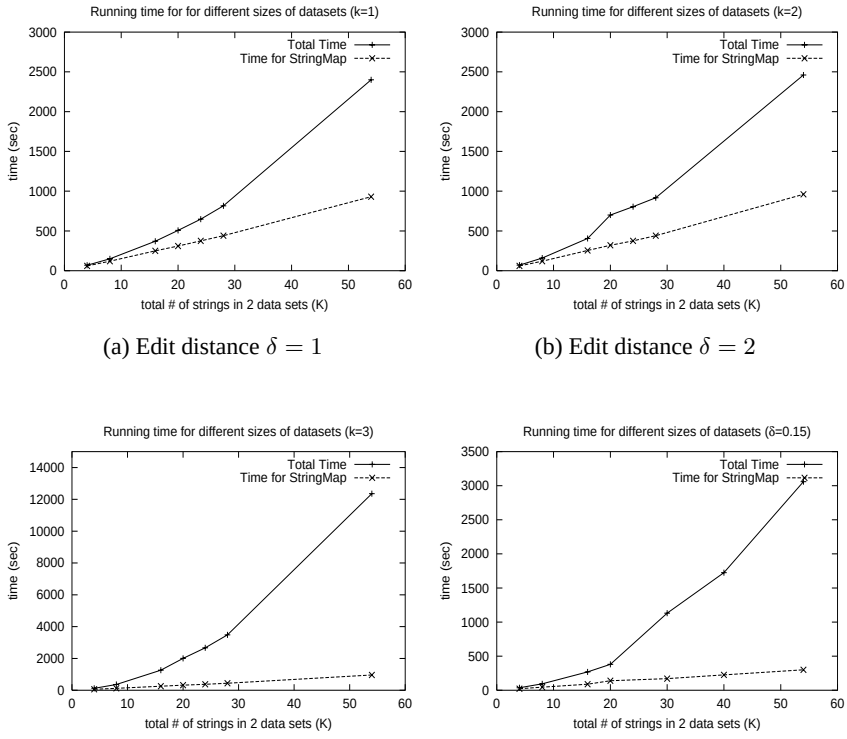
Fig. 8(a)-(c) show the time of the complete two-step algorithm, and the time of the StringMap step, assuming we use the edit-distance metric. Their gap is the time of the second step that did the R-tree similarity join. The figures show that as the data sizes increased, both the StringMap time and the total time grew. Our approach is shown to be very efficient and scalable. For instance, when the total number of strings is $54,000$, it took the approach only $41$ minutes to find the similar-string pairs, while 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. Fig. 8 (d) shows the times if we used the Jaccard distance. The times are similar to those of the edit distance.

### 6.5 Time versus Threshold $\delta'$

Fig. 9(a) and (b) illustrate how the execution time grew as the threshold $\delta'$ increased. We used the full string sets from Source 1 and Source 2, set $d = 20$, $\delta = 2$, and let $\delta'$ vary. The figures show that the time did not increase too rapidly when we increased the threshold. Therefore, to make sure we achieve a very high recall, it is desirable to choose a slightly larger threshold if possible.

### 6.6 Comparison with the Approach in [20]

In the case where the edit-distance metric is used, the approach in [20] can be used to find all the string pairs whose edit distance is within a given threshold. Its main idea is to convert the strings to q-grams stored in a relational DBMS, then run a sophisticated SQL query to find all similar-string pairs. We implemented this approach using Oracle 8.1.7 on the same PC, and let the database use indexes to run the SQL 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 $\delta' = 6.3$. Fig. 10 shows the performance difference between these two approaches. The figure shows that our approach can substantially reduce the time of finding




(a) Edit distance $\delta = 1$

(b) Edit distance $\delta = 2$

(c) Edit distance $\delta = 3$

(d) Jaccard distance $\delta = 0.2$

Figure 8: Running time (d=20).

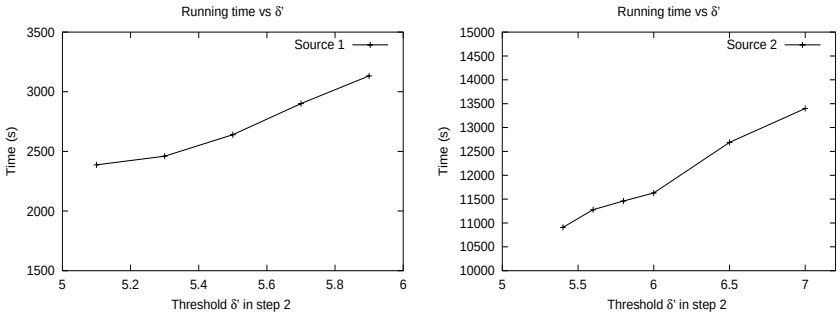
similar-string pairs. Notice that even though our approach cannot guarantee to find all such pairs, which can be achieved by the approach in [20], as we will see shortly, our approach has a very high recall.

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 [20] 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% recall) but result in significant time savings might be more preferable. In addition, the implementation of the q-gram-based approach inside a database might be less efficient than a direct implementation using other languages (e.g., the C language).

## 6.7 Recall

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 recall, i.e., ratio of similar-string pairs found among all similar-string pairs.

Fig. 11(a) shows the recall of our approach on data source 1, with different threshold $\delta$ values in the second step, using the edit-distance metric. In order to measure the recall, we first used the nested-loop approach to get all the matching record pairs. We then ran our approaches, and compared the result with all the matching pairs. As the $\delta'$ value increased, the recall also increased, and it quickly got very close to $100\%$. For instance, in the case where $\delta = 2$, the recall reached $99\%$ when $\delta' = 5.6$. When we further increased the threshold, the recall




(a) Source 1 (b) Source 2

Figure 9: Time versus threshold $\delta'$.

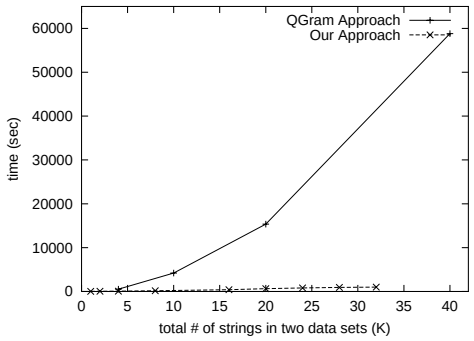


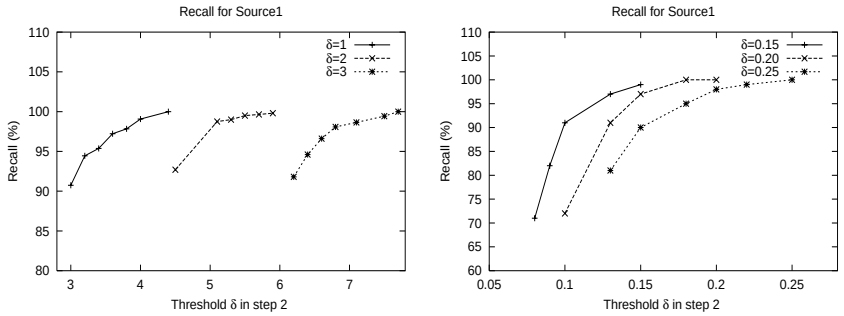

Figure 10: Our approach (approximate search) versus the q-gram approach in [20] (exact search).

continued to grow close to $100\%$. Fig. 11(b) shows similar recall results when we used the Jaccard distance on data source 1. Fig. 12 shows the similar trend for data source 2 when the edit-distance metric was used.

We also implemented the sliding-window approach in [22]. 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 measure the time and recall for each of them.

Fig. 13 shows the recall and the time for these two approaches. It shows that our approach can achieve a very high recall 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 [22] needs to consider a very large window size (and hence cost) to obtain competitive degree of recall, or choose different keys to run the sliding-window algorithm multiple times.

We also examined the effectiveness of our two-step approach on eliminating dissimilar string pairs. Fig. 14 shows the results on data source 1 using edit distance metric. It shows the cross-product size, the number of pairs after the similarity join, and the number of similar pairs, for $\delta = 1$, 2, and 3. (Our results on Jaccard coefficient metric were even better.) Notice that the $y$-axis has a log scale (in thousands). The results showed that our two-step approach can effectively eliminate dissimilar pairs. For instance, when $\delta = 2$, the approach only returned less than $0.2\%$ of the cross-product size. This ratio remained low for different edit distances. The postprocessing step took about $80$ seconds, while the total running time was about $2,500$ seconds. Thus the postprocessing time was relatively small compared to the total running time.




(a) Edit Distance (b) Jaccard distance

Figure 11: Recall versus threshold $\delta'$ for Source1, ($d = 20$).

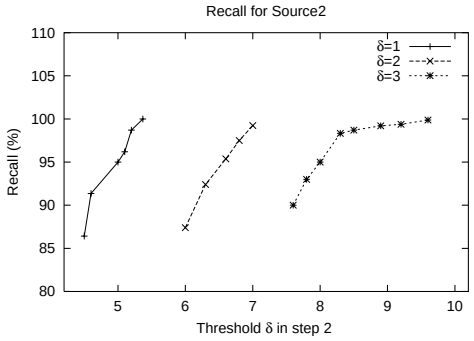



Figure 12: Recall versus threshold $\delta'$ for Source2, (edit distance, $d = 20$).

## 6.8 Results on Multiple Attributes

Now we report our experimental results for the multiple-attribute case. We report our results 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 very selective in a similarity join, and many candidate pairs were generated. Thus we mainly reported the results of similarity joins using attributes *title* and *name*.

Table 4: Results of similarity join using different attributes.

| Similarity-join Attribute | Time (sec) | Final Result Size (# of similar-record pairs) |
|---|---|---|
| $\Delta_j(title) \leq 0.15$ | $1,543$ | 702 |
| $\Delta_e(name) \leq 3$ | $1,140$ | 700 |

Table 4 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 recall (more than 99%).

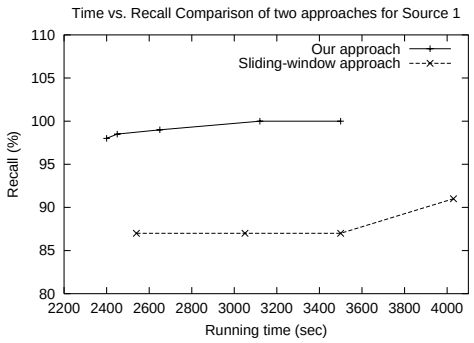



Figure 13: Our approach versus the sliding-window approach in [21].

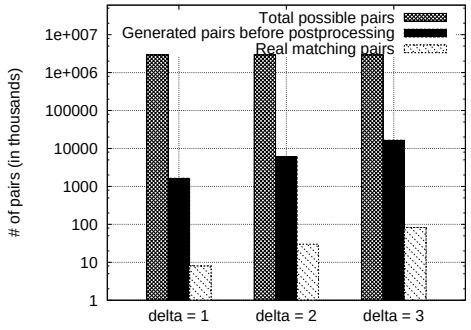



Figure 14: Candidate set size after the two steps.

## Disjunctive Normal Form

We used the query $Q_1$ in Section 2.2 as an example disjunctive normal form. We implemented the four algorithms (including exhaustive search) described in Section 5.2. We measured the running time for each conjunct in the merging rule. Our experiments showed that many record pairs were returned if we first processed the condition $\Delta_e(year) \leq 1$ ($4,587,937$ pairs) or condition $\Delta_e(year) \leq 2$ ($13,371,074$ pairs), and the post-processing time was substantially larger than those of the similarity joins on the other two attributes. In addition, the post-processing time for the conjuncts of the other two attributes was very short compared to the similarity-join times. Thus an optimal plan did not use a *year* condition to do a join first, neither did the three algorithms. Here we mainly report the results on the other two attributes. Table 5 shows the numbers.

Table 5: Running times for conjuncts in merging rule $Q_1$.

| | |
|---|---|
| $\Delta_j(title) \leq 0.10$: $1,040$ secs | $\Delta_e(name) \leq 4$: $1,780$ secs |
| $\Delta_j(title) \leq 0.15$: $1,543$ secs | $\Delta_e(name) \leq 2$: $710$ secs |

Table 6 shows the results for different algorithms. Among all the possible plans, the exhaustive search algorithm found an optimal plan, which chose $\Delta_j(title) \leq 0.15$ from Clause 2 to perform the join. This plan required $1,743$ seconds and produced $619$ pairs.

For Algorithm 1 that produces approach A, we chose the conjunct $\Delta_j(title) \leq 0.10$ 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. The total time for approach A was $1,820$ seconds, and the size of the final result set was $619$ pairs. Notice that the results of the two clauses had overlapped record pairs, explaining why $619$ is not the summation of $406$ and $518$. The recall of this approach

is more than 99%.

Table 6: Different algorithms for a disjunctive normal form.

| Algorithms | Selected Condition(s) | Time (sec) | Number of Pairs | Total Time (sec) | Total Pairs |
|---|---|---|---|---|---|
| Exhaustive Search | Clause 2, $\Delta_j(title) \leq 0.15$ | $1,743$ | $619$ | $1,743$ | $619$ |
| Algorithm 1 | Clause 1, $\Delta_j(title) \leq 0.10$ | $1,060$ | $406$ | $1,820$ | $619$ |
| | Clause 2, $\Delta_e(name) \leq 2$ | $760$ | $518$ | | |
| Algorithm 2 | Clause 2, $\Delta_j(title) \leq 0.15$ | $1,743$ | $619$ | $1,743$ | $619$ |
| Algorithm 3 | Clause 2, $\Delta_e(name) \leq 2$ | $760$ | $518$ | $1,820$ | $619$ |
| | Clause 1, $\Delta_j(title) \leq 0.10$ | $1,060$ | $406$ | | |

For Algorithm 2 that produces approach B, we found that $\Delta_j(title) \leq 0.15$ was the most selective conjunct, using which we performed the similarity join. The total time for approach B was $1,743$ seconds, and the size of the final result set was also $619$ pairs. This plan happened to be the optimal plan.

For Algorithm 3, we first chose the conjunct $\Delta_e(name) \leq 2$ from Clause 2 since it has the highest benefit/cost ratio. (Its benefit is 3, since it can cover three conjuncts (including itself). Its cost is $710$ seconds.) Then we selected $\Delta_j(title) \leq 0.10$ from Clause 1 to cover the remaining conjuncts. This plan happened to be the same as that of Algorithm 1. Notice that if we use the total running time of the covered conjuncts as the benefit of a conjunct, then this algorithm will produce the optimal plan.

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. The remaining time is spent on postprocessing the candidate record pairs. Algorithm 3 does a more sophisticated search by greedily choosing efficient conjuncts that can cover many other conjuncts.

# 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 the 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 process will proceed. A multidimensional similarity join over the chosen attributes is performed to determine similar pairs of records. Our extensive experiments using real data sets showed that our solution has very good efficiency and recall. In addition, our approach is very extendable, since many existing mapping and join techniques can be used, many similarity functions between attributes can be supported.

# 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. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

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

[5] 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.

[6] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.
[7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.
[8] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.
[9] E. Chvez, G. Navarro, R. Baeza-Yates, and J. L. Marroquln. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.
[10] W. W. Cohen, H. A. Kautz, and D. A. McAllester. Hardening soft information sources. In *Knowledge Discovery and Data Mining*, pages 255–259, 2000.
[11] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Workshop on Information Integration on the Web*, 2003.
[12] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. Technical report, Rutgers Univ., 2001.
[13] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.
[14] V. C. (Editor) and L. D. B. (Editor). *Image Databases: Search and Retrieval of Digital Imagery*. John Wiley and Sons, 2001.
[15] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid. Tailor: A record linkage toolbox. In *ICDE*, 2002.
[16] 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.
[17] 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.
[18] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.
[19] J. C. Gower and P. Legendre. Metric and euclidean properties of dissimilarity coefficients. *Journal of Classification*, 3:5–48, 1986.
[20] 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.
[21] 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.
[22] M. A. Hernández and S. J. Stolfo. An incremental merge/purge procedure. Technical report, University of Illinois, 2000.
[23] 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.
[24] G. R. Hjaltason and H. Samet. Contractive embedding methods for similarity searching in metric spaces. Technical report, University of Maryland Computer Science, 2000.
[25] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical Report 99-50, Rutgers Univ., 8 1999.
[26] 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.
[27] P. Indyk. Sublinear time algorithms for metric space problems. In *STOC*, pages 428–434, 1999.
[28] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03), Kyoto, Japan*, 2003.
[29] J. Kamps. Exploiting keyword structure for domain-specific retrieval. In *Cross Language Evaluation Forum*, 2002.
[30] N. Koudas and K. C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.
[31] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Piblications, Beverly Hills, CA, 1978.
[32] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.
[33] M. Lay. Dblp bibliography.
[34] M.-L. Lee, T. W. Ling, and W. L. Low. Intelliclean: a knowledge-based intelligent data cleaner. In *Knowledge Discovery and Data Mining*, pages 290–294, 2000.
[35] M.-L. Lee, T. W. Ling, H. Lu, and Y. T. Ko. Cleansing data for mining and warehousing. In *Database and Expert Systems Applications*, pages 751–760, 1999.
[36] M.-L. Lee, T. W. Ling, H. Lu, and Y. T. Ko. Cleansing data for mining and warehousing. In *DEXA*, pages 751–760, 1999.
[37] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[38] D. Loshin. Value added data: merge ahead. *Intelligent Enterprise*, 3(3), 2000.
[39] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *SIGMOD*, pages 1–12, 1999.
[40] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *The VLDB Journal*, pages 381–390, 2001.
[41] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.
[42] S. Sarawagi, A. Bhamidipaty, A. Kirpal, and C. Mouli. Alias: An active learning led interactive deduplication system. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB) (Demonstration session)*, Hongkong, August 2002.
[43] K. C. Sevcik and N. Koudas. High dimensional similarity joins: Algorithms and performance evaluation. *TKDE*, 12(1):3–18, 2000.
[44] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *ICDE*, pages 301–311, 1997.
[45] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD*, pages 343–354, 2000.
[46] T. Shinohara, J. An, and H. Ishizaka. Approximate retrieval of high-dimensional data with L1 metric by spatial indexing. *Journal of New Generation Computer*, 18(1):39–47, 2000.
[47] 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.
[48] W. Winkler. Advanced methods for record linkage. Technical report, Statistical Research Division, Washington, DC: U.S. Bureau of the Census., 1994.
[49] P. N. Yianilos and K. G. Kanzelberger. The LIKEIT intelligent string comparison facility. Technical report, NEC Research Institute, 1997.
[50] F. W. Young and R. M. Hamer. *Multidimensional Scaling: History*. Theory and Applications Erlbaum, New York, 1987.