

## VLDB 2002 Paper Submission Cover Page

Submission number: **447**  
Title: **Efficient Similarity String Joins in Large Data Sets**  
Authors: Liang Jin, Chen Li, Sharad Mehrotra  
Category: Research  
Topic area: Infrastructure for Information Systems  
Relevant topics: Data Cleansing, Data Integration, Data Quality, Data Transformation, Interoperability, Heterogeneous and Federated Databases  
Contact author: Liang Jin  
Contact email: liangj@ics.uci.edu  
Postal mail: 127B Computer Science Trailer  
Department of Information and Computer Science  
University of California  
Irvine, CA 92697-3430  
USA  
Telephone: 01-949-824-9683

# Efficient Similarity String Joins in Large Data Sets

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

*Many emerging database applications require very efficient mechanisms to perform similarity joins. While similarity joins have been extensively explored in the literature, the problem of similarity string joins has received comparatively less attention. The string-join problem can be specified as follows: Given two large sets of strings, a distance metric (e.g., edit distance) between strings, and a predefined threshold  $k$ , how to find all the pairs from the two sets such that their distance is no greater than  $k$ ? Such a problem arises naturally in a variety of applications such as data cleansing and data integration. We propose a novel two-step process for string join. We first map strings as points in a multidimensional space such that the mapped space preserves the distances over strings. In general, many techniques can be used for this purpose. We focus on the FastMap algorithm since it provides a good mapping in linear time. In the second step, we do a multi-dimensional similarity join, and we use the algorithm proposed by Hjaltason and Samet [HS98]. Our extensive experiments using real data sets show that our solution has very good efficiency and accuracy. Also, unlike some of the recent string-join proposals that work only when similarity between strings is computed using the edit distance function, our approach works for any arbitrary similarity function between two strings (or objects).*

---

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 28th VLDB Conference,  
Hong Kong, China, 2002**

## 1 Introduction

Many emerging database applications require very efficient mechanisms to perform similarity joins of objects. Examples include multimedia databases [Jag91], time-series databases [GD01], and string correction [Kuk92]. While many efficient approaches to similarity joins have been explored in the literature, the problem of similar-string joins has received comparatively less attention. The problem can be specified as follows: Given two large sets of strings and a predefined threshold  $k$ , how to find the pairs from the two sets such that their edit distance is no greater than  $k$ ?

The similar-string-join problem arises naturally in a variety of applications. For instance, in data cleansing [LLLK99] and data integration [Li01], we need to collect data from various autonomous and heterogeneous databases to conduct analysis to make decisions. Often, the data from these different sources has inconsistent representations and even errors. The goal of data cleansing is to match records that refer to the same real-world entity while not being syntactically equivalent, as illustrated by the following example.

**EXAMPLE 1.1** A hospital of a medical school has a database with thousands of patient records.<sup>1</sup> Every year it received 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 recorded as “Frans” by mistakes. The main task here

---

<sup>1</sup>For confidentiality reasons, we omit the real name of the hospital.

is to link records from different databases in the presence of mismatched information.  $\square$

The first issue in data merging/cleansing is that of determining an appropriate similarity/distance metric between strings. Note that as the above example illustrates, commonly used edit distance might not be the best for this purpose. Recent studies [Los00] suggest that distance/similarity functions depend largely upon the specific domain (e.g., different functions may be suitable for merging names of people, addresses, and/or gene sequences in a bioinformatics application). Many industrial organizations [Val, Dat, Tri, Fir] have explored such domain-specific distance functions (and merging rules) in the context of variety of applications, such as CRM, enterprise data integration, postal address integration, etc. The approach to similar-string joins discussed in this paper is independent of the specific function used. In contrast, most research literature on string join has primarily dealt with string matching based on edit distance. For conformity with existing literature, we will develop our ideas using the edit distance, although our approach is independent of the specific distance function used to match strings.

A simple solution to the similar-string-join problem is to use a nested-loop approach to generate the Cartesian product of strings, and then use the distance metric to compute the distance between each pair of strings. This approach is not desirable, since as two data sizes are large, the approach becomes computationally prohibitive (See Section 2). For instance, if there are 60K strings in each data set, the nested-loop approach needs to consider 3.6 billion pairs of strings to compute their distances.

Many techniques have been proposed on finding similar pairs of objects in high-dimensional spaces (e.g., [ARS98, BKS93, CMTV00, KS97, KS98, Kuk92, SML00, SSA97]). Most of these techniques assume objects exist in a multi-dimensional space. However, since the strings in our problem do not fit into such a Euclidean space given their edit distances, these techniques cannot be used directly to solve our problem.

In this paper we propose a two-step solution to the similar-string-join problem. In the first step, we combine the two sets of strings, and map them into a high-dimensional Euclidean space. In general, many multidimensional-scaling (MDS) techniques [KW78] can be used to do the mapping. We focus on the FastMap algorithm [FL95] due to its simplicity and efficiency. We develop a linear algorithm called “StringMap” to do the mapping.

In the second step, we find similar pairs of the objects in the Euclidean space, whose distance is no greater than a new threshold  $\delta$ . This new threshold is closely related to the old threshold  $k$  of string distances, and sampling techniques can be used to decide the new threshold. For each pair of objects in the

result of the second step, we check their corresponding strings to see if their edit distance is within  $k$ . Again, many similarity-join algorithms can be used in this step. In this paper we use the algorithm proposed by Hjaltason and Samet [HS95]. Its main idea is to build two R-trees [Gut84] for the objects of two data sets, and traverse the two trees to find pairs of objects whose distance is within certain threshold.

While our approach does not guarantee to find all the string pairs whose edit distance is within  $k$ , as we will see in Section 5, our experimental results show that we can achieve a very high recall (more than 99%). Our approach has the following advantages.

1. It is very efficient, and requires little extra storage space.
2. 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.
3. It does not depend on a specific similarity function of strings. In this paper we focus on the edit distance function. Our solution can be easily generalized to any similarity function between strings. For example, if there is a function that takes abbreviations (“Jack Lemmon” versus “J. Lemmon”) into consideration, we can easily use this function in the first step of our solution.
4. Our extensive experiments using real large data sets show that our solution has very good efficiency and recall.

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 of the solution, which conducts a similarity join in the high-dimensional space. In Section 5 we give the results of our extensive experiments. Section 6 compares our solution with other approaches in the literature. In Section 7 we conclude and discuss future directions.

## 2 The Similar-String-Join Problem

In this section we formalize the similarity-string-join problem.

**Definition 2.1 (String Edit Distance)** Given two strings  $r$  and  $s$ , their *edit distance*, denoted by  $ed(r, s)$ , is the *minimum* number of edit operations (insertions, deletions, and substitutions) of single characters that are needed to transform  $r$  to  $s$ .  $\square$

For instance,

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

Symbol	Meaning
$R = \{r_1, \dots, r_{n_1}\}$	relation $R$ with $n_1$ strings
$S = \{s_1, \dots, s_{n_2}\}$	relation $S$ with $n_2$ strings
$N = n_1 + n_2$	total number of strings
$k$	edit-distance threshold
$d$	dimensionality of the Euclidean space
$R' = \{r'_1, \dots, r'_{n_1}\}$	mapping objects of $n_1$ strings in $R$
$S' = \{s'_1, \dots, s'_{n_2}\}$	mapping objects of $n_2$ strings in $S$
$\delta$	threshold in the new space

Table 1: Symbols.

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,  $\text{ed}(\text{“Jack Lemmon”}, \text{“Jack Lemon”}) = 1$ , and  $\text{ed}(\text{“Anderson”}, \text{“Zandersson”}) = 2$ . It is known that the complexity of computing the edit distance between strings  $r$  and  $s$  is  $O(\text{len}(r) \times \text{len}(s))$ , where  $\text{len}(r)$  and  $\text{len}(s)$  are the lengths of  $r$  and  $s$ , respectively [YK97].

**Definition 2.2 (Similar-String-Join Problem)** Given two sets  $R$  and  $S$  of strings and a predefined value  $k$ , we want to find pairs of strings  $(r, s)$  from  $R$  and  $S$  respectively, such that  $\text{ed}(r, s) \leq k$ .  $\square$

For simplicity, a pair of strings is called a “similar-string pair” if their edit distance is no greater than  $k$ ; otherwise, it is called a “dissimilar-string pair.”

A straightforward approach is to use the nested-loop algorithm. That is, we scan relations  $R$  and  $S$ . For each string pair  $(r, s)$  from the two relations, we check if  $\text{ed}(r, s) \leq k$ . The complexity of this naive approach is  $O(|R| \times |S|)$ , which is computationally prohibitive when the two sets are large. Figure 1 shows the time of using this nested-loop algorithm on different sizes of data sets. In the experiments both data sets have the same number of strings. The  $x$ -axis is the *total* number of strings in two sets. (See Section 5 for our experimental environment.) The figure indicates that as the data-set sizes increase, the running time grows quadratically. For instance, it takes more than 5 hours to find similar-string pairs of two data sets, each of which has 16K strings. Thus the approach is too inefficient to be practical.

Our proposed solution has two steps. In the first step, we map strings to objects in a multidimensional Euclidean space, such that the mapped space preserves the edit distances over strings. In the second step, we conduct a multi-dimensional similarity join in the Euclidean space. Table 1 summarizes some symbols used in this paper. We will describe the two steps in details in the next two sections.

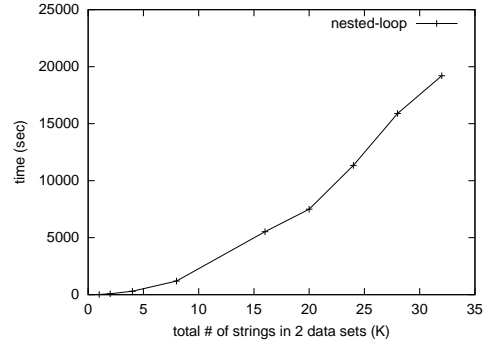


Figure 1: Time of nested-loop approach.

### 3 Step 1: Mapping Strings to Euclidean Space

In this section we discuss the first step of our approach. We review the FastMap algorithm, then present our StringMap algorithm that maps strings into a high-dimensional space.

#### 3.1 FastMap

The purpose of the FastMap algorithm [FL95] is to map objects into points in a  $d$ -dimensional space, such that the distances of the objects are preserved. The problem can be described formally as follows.

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

It is assumed that the distance function of the old objects is nonnegative, symmetric, and satisfies the triangular inequality. (Clearly the edit distance function has these properties.) One way to describe how the distances are maintained is to use the following *stress* function:

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

in which  $d_{ij}$  is the dissimilarity measure between object  $O_i$  and object  $O_j$  in the old 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. The goal of the FastMap is to make *stress* as small as possible.

See [FL95] for the details of the FastMap algorithm. Its main idea is to find  $d$  mutually-orthogonal directions. 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. In addition, it computes the remaining distance matrix after the projections.

### 3.1.1 Applying FastMap

In the first step of our approach, we combine the two sets into a set with  $N = n_1 + n_2$  strings. While trying to use FastMap to map strings to objects in a multi-dimensional space, we are faced with several challenges.

1. FastMap requires a distance matrix of the original objects. However, we do not want to run a  $O(N^2)$  procedure to compute all the edit distances.
2. We do not know what dimensionality  $d$  to use.

In the next two subsections, we first present our algorithm, called “StringMap,” which maps strings into a high-dimensional space, assuming we are given the  $d$  value. Then we discuss how to choose a good  $d$  value.

### 3.2 StringMap: Mapping Strings to Objects

Our StringMap algorithm modifies FastMap so that the computation of the distance matrix is not necessary. The matrix is needed in FastMap to compute the pivot objects on each dimension.

Figure 2 presents the algorithm. It shares the main idea of FastMap; that is, it iterates to find pivot strings to form  $d$  orthogonal directions, and compute the coordinates of the  $N$  strings on the  $d$  axes.<sup>2</sup> The function *ChoosePivot(int h)* 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. As in [FL95], a typical  $m$  value could be 5.

One important function is

$$GetDistance(int a, int b, int h)$$

which computes the distances between strings (indexed by  $a$  and  $b$ ) after they are mapped to the first  $h - 1$  axes. As shown in Figure 3, it iterates over the  $h - 1$  dimensions, and does the computation only using the two strings and their already-computed coordinates on the  $h - 1$  dimensions.

<sup>2</sup>Note that the StringMap algorithm removes the recursion in FastMap.

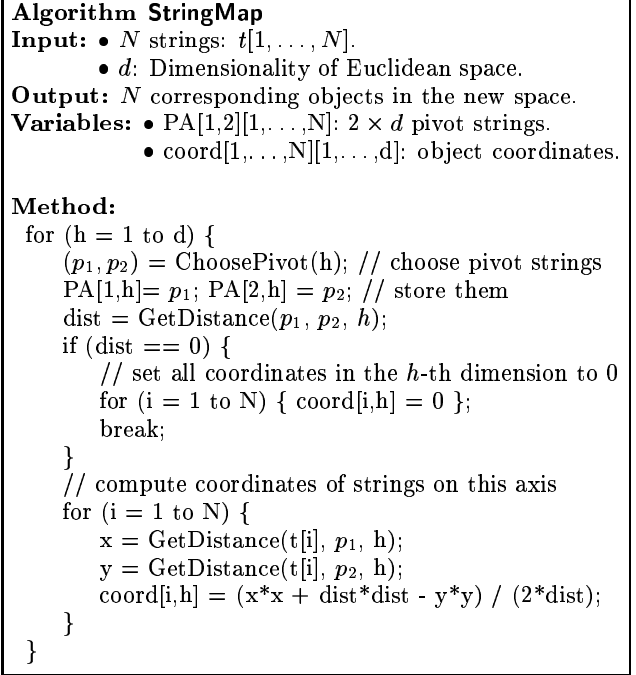


Figure 2: Map strings to objects in a Euclidean space

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 the edit distance of two string, 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 we can easily reduce the complexity “ $m \times N$ ” in Equation 2 as follows. At each step in the function *ChoosePivot()*, 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  $s_1$ . 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.

### 3.3 Choosing Dimensionality $d$

Algorithm StringMap assumes that a dimensionality  $d$  is given. A good  $d$  value should have the property that

```

// choose two pivot strings on the  $h$ -th dimension
Function ChoosePivot(int h)
{
  seed  $s_a$  = a random string from  $t[1], \dots, t[N]$ ;
  for (i = 1 to m) { // a typical m value could be 5
    // use function GetDistance(.,.,h)
    // 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)
{
  A =  $t[a]$ ; B =  $t[b]$ ; // get strings
  dist = ed(A, B);
  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 );
}

```

Figure 3: Functions used in StringMap

after the mapping, similar strings can be differentiated from those dissimilar ones. On the one hand,  $d$  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, the dimensionality  $d$  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 similarity joins in the second step, we want to avoid the curse of dimensionality [Bel61, EE01]. In particular, as  $d$  increases, it becomes more time-consuming to find object pairs whose distance is within a new threshold.

We can now describe our approach to choosing a dimensionality  $d$  for StringMap.

1. Randomly select a small number (say, 1K) of strings from the two original data sets  $R$  and  $S$ . Use the nested-loop approach to find all similar-string pairs within threshold  $k$  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, say,  $w$ .
3. Find a dimensionality  $d$  with a low *cost*.

$$cost = \frac{\# \text{ of obj. pairs within distance } w}{\# \text{ of similar-string pairs}} \quad (3)$$

Intuitively, the cost is the average number of object pairs we need to retrieve in step 2 for each similar-string pair, if we take  $w$  as the threshold  $\delta$  for selecting similar-string pairs in the mapped space.

Notice that we only use the pairs of selected strings to compute the cost. This value measures how well the 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. Thus the lower the cost is, relatively the fewer object pairs we need to retrieve in step 2 whose original edit distance is more than  $k$ . Figure 7 in Section 5.3 shows that typically a good dimensionality value could be between 15 and 20.

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

The second step of our approach finds pairs of objects whose Euclidean distance is within a new threshold  $\delta$ . We first discuss how to select the threshold, and then show how to apply one existing technique in the literature to do the similarity join.

### 4.1 Choosing Threshold $\delta$

Our approach to selecting the dimensionality  $d$  also identified the threshold  $w$  that can be used to identify similar pairs in the mapped space. It is, of course, possible that the threshold  $w$  identified using a single data sample used to choose  $d$  may not work well in differentiating similar pairs over the entire data set. Since our interest is to “miss” as few similar pairs as possible, we may wish to improve the threshold now that the dimensionality  $d$  is fixed. Ideally, the threshold  $\delta$  should be set to a maximal value of the new distance between any two similar-string pairs in the mapped space. Then it will guarantee no false dismissals. However, it is infeasible to find such a threshold, since we do not know all the similar-string pairs without computing the expensive nested-loop join.

Therefore, we randomly selected a small number (say, 1K) of strings from both data sets  $R$  and  $S$ .<sup>3</sup> We find all the similar-string pairs in these selected strings, and compute their new Euclidean distances after StringMap. We can choose their maximal new distance as the new threshold  $\delta$ . We can do this sampling multiple times (on different sets of selected strings), and choose  $\delta$  as the maximal new distance of those similar-string pairs. In Section 5.4 we will give results on how to choose  $\delta$ .

<sup>3</sup>Notice these selected strings might be different from those strings used to decide the dimensionality  $d$  in Section 3.3.

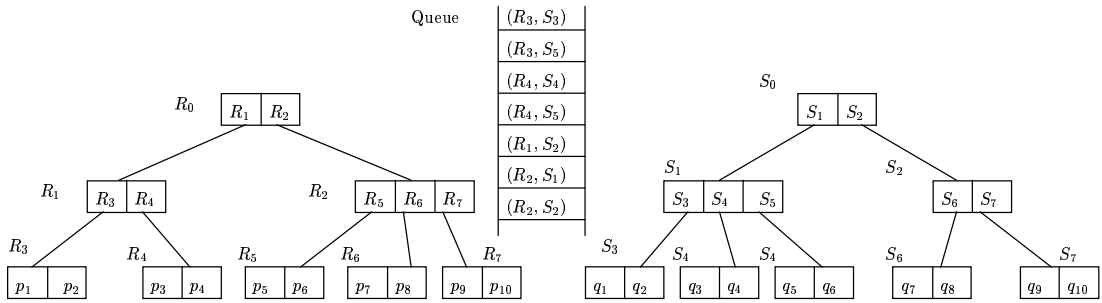


Figure 4: Finding similar-object pairs using R-trees.

## 4.2 Finding Object Pairs within $\delta$

After selecting  $\delta$ , we want to find all those object pairs whose distance is within this threshold. Many high-dimensional similarity-join techniques can be used here. In this paper we use the algorithm proposed by Hjaltason and Samet [HS98].

The main idea of the algorithm is as follows. We first build two R-trees for the objects in  $R'$  and  $S'$  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  $\delta$ . As we do the traversal, a queue is used to store pairs of nodes (internal nodes or leaf nodes) from the two trees.<sup>4</sup> 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.

We use Figure 4 to explain the algorithm in details. 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 [RKV95] 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 example, 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  $\delta$ , so we insert them into the queue. Then we remove

<sup>4</sup>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 [HS98], since they want to find the “top-k” object pairs with the smallest distances.

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. For instance, if 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 edit distance of their original strings. We output this pair of strings if the edit distance is within the original threshold,  $k$ .

## 4.3 Traversal Strategies

Different strategies can be used to traverse the two R-trees, such as “depth first” and “breadth first.” In our experiments, we have the same observations as in [HS98]. That is, using 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 has about 28K 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 we 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 some pairs of leaf nodes can be processed.

## 5 Experiments

In this section we present our experimental results, covering most aspects of our solution to the similar-string-join problem.

### 5.1 Data Sources

Our experiments use data sets with real names collected from the World-Wide Web. The following are two main sources used.

1. **Source 1** consists of 54,000 movie star names collected from The Internet Movie Database.<sup>5</sup> The length of each name varies from 5 to 20, and their average length is about 12. Figure 5(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.<sup>6</sup> 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 in Dataset2 vary from to 40, and their mean string length is around 15. Figure 5(b) shows the distribution of the string lengths.

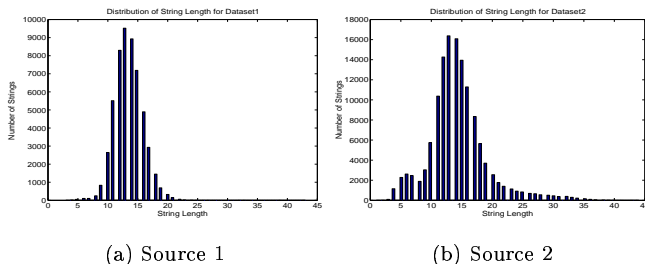


Figure 5: String-length distribution of two sources.

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. 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.

### 5.2 Nested-Loop and Our Approach

Figure 6 shows the performance difference of the nested-loop approach and our approach. We selected subsets of strings from Source 1, and we let both sets have the same number of strings. In our approach we

<sup>5</sup><http://www.imdb.com/>

<sup>6</sup><http://feefhs.org/dpl/dv/indexdv.html>

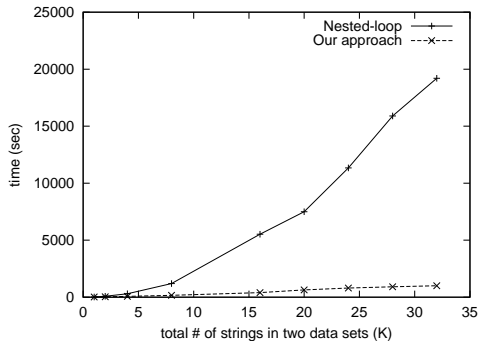


Figure 6: Comparing nested-loop with our approach.

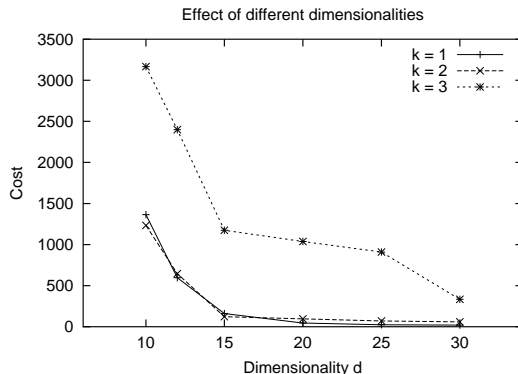


Figure 7: Costs of different dimensionalities.

chose threshold  $k = 2$ , dimensionality  $d = 20$ , and new threshold  $\delta = 5.9$ . The  $x$ -axis is the *total* number of strings. The figure shows that our approach can substantially reduce the time of finding similar-string pairs. For instance, when each data set had 16K strings, it took the nested-loop approach about 19,200 seconds (5 hours 20 minutes), while it took our approach only about 1000 (less than 17 minutes).

### 5.3 Choosing Dimensionality $d$

As discussed in Section 3.3, we need to choose a good dimensionality  $d$  for the StringMap algorithm, so that we can select a new threshold  $\delta$  to differentiate similar-string pairs from other pairs. In this experiment, we showed how to select a  $d$  value. We considered the case where  $k = 1, 2$ , and 3. We randomly selected 2K strings from Source 1 to form two data sets with the same size, and ran StringMap using different dimensionalities.

Figure 7 shows cost values for different dimensionalities. (See Section 3.3 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 an edit distance greater than  $k$ . 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  $k = 1$  and  $k = 2$ , and  $d = 30$  is good for  $k = 3$ .

Figures 8(a) to (g) show the distributions of the object-pair distances after StringMap. We randomly sampled 2K from Source 1 to form two data sets with the same size. We chose  $k = 2$  for similar-string pairs. The figures show that StringMap can keep edit distances pretty well. For instance, when  $d = 20$ , all the sampled similar-string pairs had new object-pair distances within 5.5, while a large number of dissimilar string pairs had their object-pair distances more than 5.5. As the dimensionality increased, the distances of both the similar-string pairs and the dissimilar-string pairs increased, and we had a better threshold to differentiate these pairs, as shown in Figure 7.

#### 5.4 Choosing Threshold $\delta$

As discussed in Section 4.1, we selected threshold  $\delta$  for the second step as follows. For both sources, 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  $\delta$  as the largest new object-pair distance of the sampled similar-string pairs.

	$k = 1$	$k = 2$	$k = 3$
Source 1	4.5	5.9	7.735
Source 2	5.36	7.0	10

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

Table 2 shows the  $\delta$  values used in step 2 for  $k = 1, 2$ , and 3 for the two sources. Notice that since the two sources have different strings with different length distributions, it is not surprising that we need to choose different  $\delta$  values by sampling, even for the same  $k$  and  $d$  settings.

#### 5.5 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 let the total size of strings vary from 2K, 4K, 8K, 16K, to 54k from Source 1. We also ran the experiment on the complete set of strings from Source 2. We measured the execution time. In the experiments, we considered the case where  $k = 1, 2$ , and 3. We chose dimensionality  $d = 20$ . Figures 9(a) to (c) show the time of the complete algorithm, and the time of the StringMap step. Their gap is the time of the second step that did the similarity join.

The figures show that as the data sizes increased, both the StringMap time and the total time grew. For instance, when the total number of strings is 56,000, it took our approach only 41 minutes to find the similar-string pairs. Thus our approach is very efficient in finding results, and it has a good scalability. The figures also indicate that other similarity-join techniques may be used in the second step to improve its performance.

Figures 10(a) and (b) illustrate how the execution time grew as the threshold  $\delta$  increased. They show that the time did not increase too rapidly when we increased the threshold. Therefore, to make sure we achieve a very high recall (see below), it is desirable to choose a slightly larger threshold.

#### 5.6 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, defined as follows:

$$recall = \frac{\# \text{ of similar-string pairs found}}{\# \text{ of all similar-string pairs}} \quad (4)$$

Again, we did sampling to estimate the recall. We randomly selected certain number of strings from both dataset, similar to what we did in deciding the threshold  $\delta$ . After running StringMap with  $d = 20$ , we calculated the new Euclidean distances of those similar-string pairs in the sampled strings. Using our selected  $\delta$ , we know which pairs can be captured in the second step. In other words, our step 2 can capture those similar-string pairs whose new Euclidean distance is within  $\delta$ . Thus we can compute the recall using the sampled strings. Notice that we did sampling here to evaluate how well our approach can capture similar-string pairs, and the sampling time is not included in the running time of our approach. In most cases it took only a few minutes to do the sampling.

Figures 11(a) and (b) show the recalls of both data sources, with different threshold  $\delta$  values in the second step. As the  $\delta$  increased, the recall also increased, and it quickly got very close to 100%. For instance, in the case where  $k = 2$ , the recall reached 99% when  $\delta = 5.6$  for Source 1, and  $\delta = 6.9$  for Source 2. When we further increased the threshold, the recall continued to grow close to 100% (based on the sampled data). Figures 12(a) and (b) show how the running time affects the recall.

The motivation of showing these figures is that in case it takes too much time to use a large  $\delta$  threshold to find all similar-string pairs, we may try to use a smaller  $\delta$ . In this case even though we might miss some pairs, our recall is still very high, and the running time could be much less.

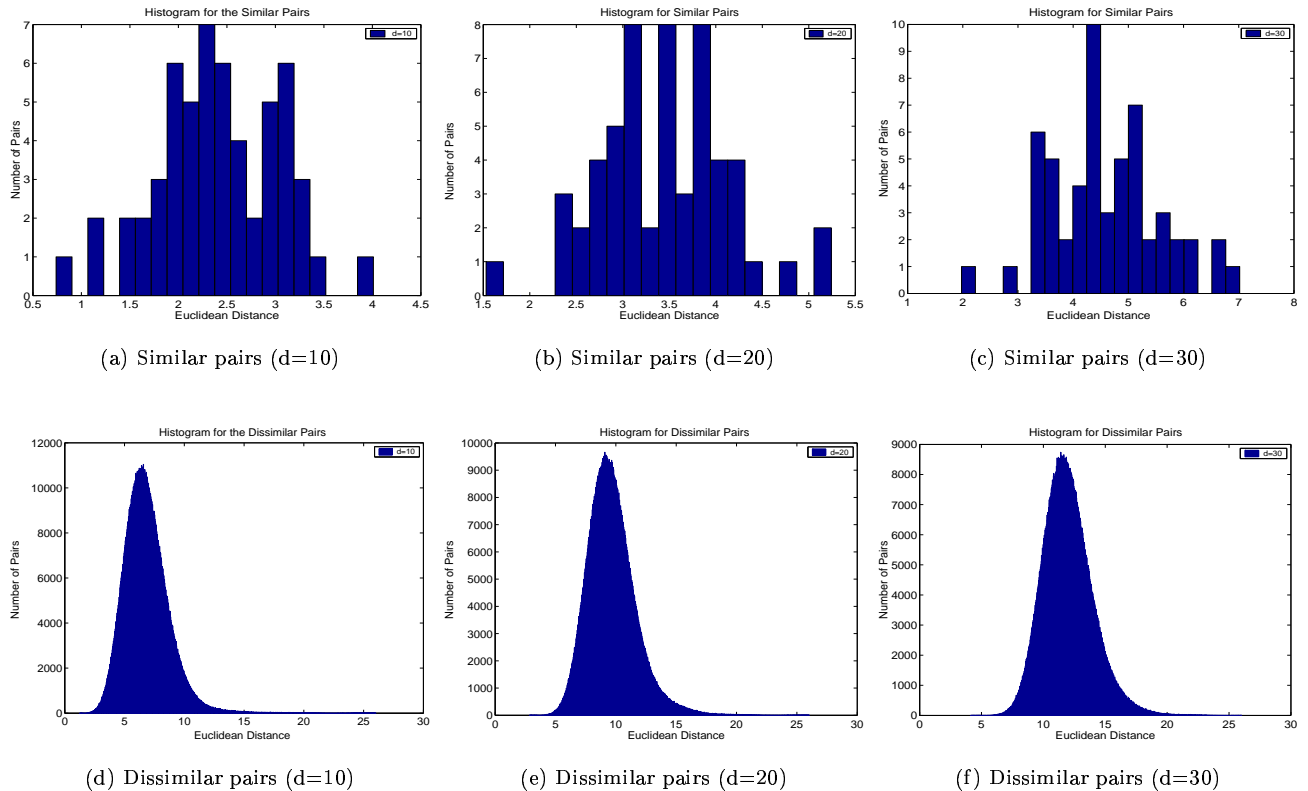


Figure 8: Histograms of Euclidean Distances ( $k=2$ ).

## 5.7 Summary of Experiments

The experimental results show that our algorithm can find almost all the similar-string pairs very efficiently. By choosing the right dimensionality (e.g.,  $d = 20$ ), the StringMap algorithm can preserve the edit distances of the strings very well. We can also easily find a new threshold that can differentiate those similar-string pairs from other pairs. As the data sizes increase, the increase of our running time is better than quadratic. In summary, our proposed algorithm has both good efficiency and effectiveness in finding similar-string pairs.

## 6 Comparison with Other Approaches

We compare our work to two previous approaches on string joins in the literature that have been motivated by the problem of data cleansing and integration.

The paper by Hernández and Stolfo studies string join in the context of the “Merge/Purge Problem” [HS95], i.e., how to merge records from different databases. The main idea of their algorithm is to sort the strings lexicographically, then use a window to scan all the strings. For the strings in the window, the algorithm finds the string pairs whose similarity value satisfies certain criterion. Notice that if we use

edit distance to define the similarity, two similar names might not be able to be captured in the window if its size is small. For example, the edit distance of “Anderson” and “Zanderson” is 1, but their names can be very far from each other in the sorted list. The algorithm assumes that two tables could have several different attributes. For two records that should be merged, it is expected that they could have one attribute in which the two records can be captured by the window. Even so, the approach is susceptible to deterministic data entry errors, e.g., the first character of the string is always erroneous.

We believe that our approach of mapping strings to multidimensional space and then using a similarity-join algorithm for merging will result in better quality of the merge results. One particular reason is that our approach does not depend on any lexicographical order of strings.

More recently, efficient string-join algorithms have been studied in [GIJ<sup>+</sup>01], where 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 size of the string and the number  $q$ . An additional column is added to the relation scheme to

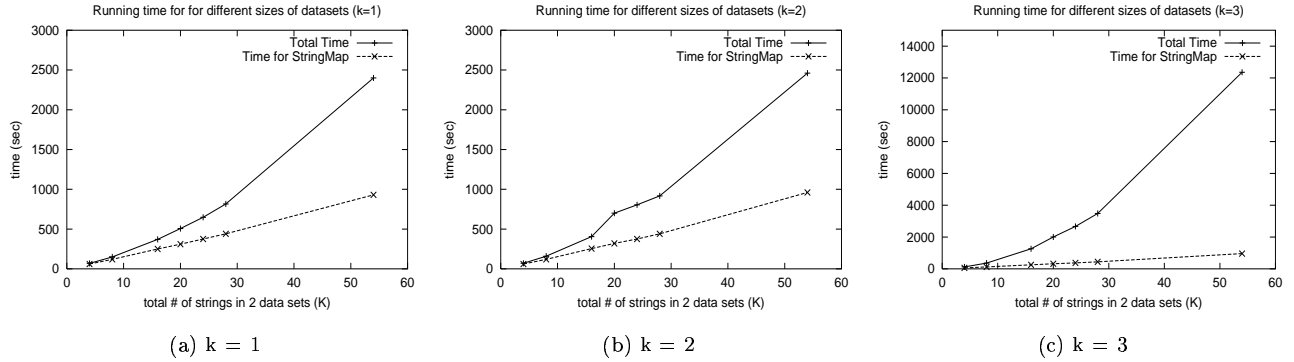


Figure 9: Running time ( $d=20$ ).

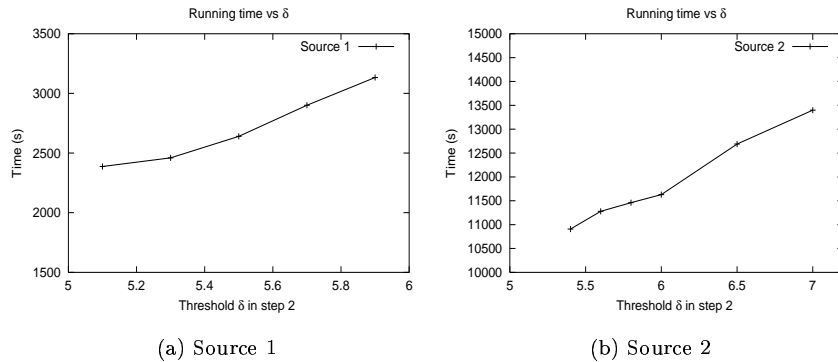


Figure 10: Time versus threshold  $\delta$ .

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 authors show that for the data sets considered, 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 nested loop.

While their approach is attractive, it does not directly compare to our approach, since the technique based on  $q$ -grams can only be used when the distance metric between two strings used is edit distance (or its

variations). Frequently, especially in data-cleansing applications, as discussed in the introduction, the distance between strings is not computed using edit distance, but a domain-specific metric. In contrast, our approach is general, and any distance measure between strings can be supported.

In addition, in the context of data-cleansing applications, as discussed in [HS95], catalogs to be merged usually do not have only a single string, but rather may consist of multiple string attributes. Using the  $q$ -gram approach will become progressively worse, since the data size increase will grow multiplicatively with the number of string attributes in the relation. In contrast, the approach for string join proposed in this paper can be easily combined with an appropriate multi-attribute merge algorithm without much overhead.

While the above are the advantages of our approach, its limitation compared to the  $q$ -gram approach is that in our case we do not guarantee to find all results.

If we restrict ourselves to string distances and have a relation with only one string attribute, it will be interesting to see a performance comparison between our approach and the  $q$ -gram approach. We leave

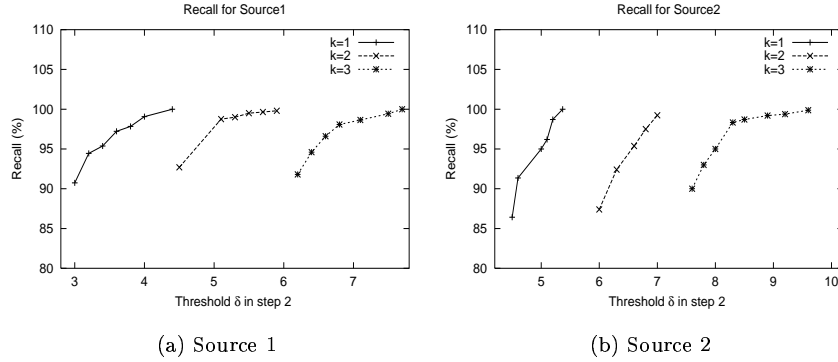


Figure 11: Recall versus threshold  $\delta$  ( $d=20$ ).

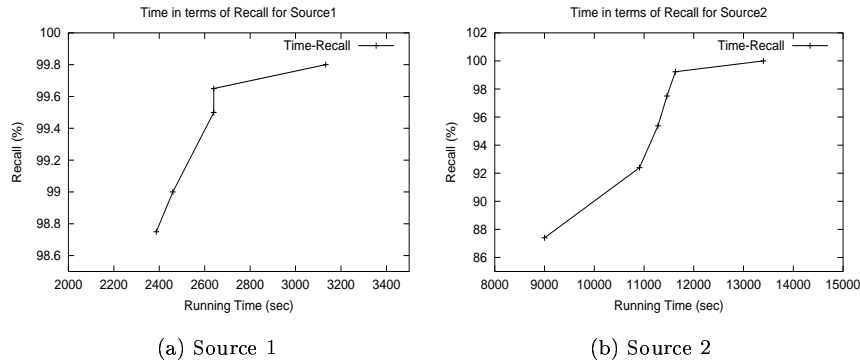


Figure 12: Recall versus time ( $d = 20, k = 2$ ).

such a comparison to future work. We note that the primary advantage of the  $q$ -gram approach is that it can be done using a hash-join (since it is an equi-join), whereas we still need to do a spatial join (even though we can exploit a data structure such as R-tree for this purpose). Some recent spatial-join algorithms (e.g., [KS98]) discuss how similarity joins can be implemented very efficiently in a manner similar to hash joins using a space-filling curve. We believe that such similarity-join mechanisms will afford us all the advantages of the  $q$ -gram approach.

## 7 Conclusion

In this paper we developed a novel technique to solve the similar-string-join problem: how to find pairs of strings from two large data sets whose edit distance is within a predefined threshold? This problem arises naturally in a variety of applications, such as data cleansing and data integration. Our approach has two steps. We first map strings to objects in a multidimensional space, such that the mapped space preserves the edit distances over strings. In general, many tech-

niques can be used for this purpose. We focused on the FastMap algorithm, and proposed a linear algorithm called StringMap to do the mapping. In the second step, we do a multi-dimensional similarity join, and we use the algorithm proposed by Hjaltason and Samet [HS98]. Our extensive experiments using real data sets showed that our solution has very good efficiency and recall. In addition, it has several advantages. (1) It does not require much extra storage space. (2) It is very extendable, since many existing techniques can be used in the two steps. (3) It can be easily generalized to other similarity functions between two strings, and even between any two objects in general.

Currently we are working on finding a bound on the Euclidean distance after the StringMap algorithm. Once we find such a bound, we can guarantee to find all the pairs in the answer. We are also investigating other techniques that can be used in the two steps.

## Acknowledgments

We thank Michael Ortega-Binderberger for providing his implementation of the algorithm in [HS98] and

many fruitful discussions.

## References

- [ARS98] Alsabti, Ranka, and Singh. An efficient parallel algorithm for high dimensional similarity join. In *IPPS: 11th International Parallel Processing Symposium*. IEEE Computer Society Press, 1998.
- [Bel61] Richard E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, New Jersey, U.S.A., 1961.
- [BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.
- [CMTV00] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, pages 189–200, 2000.
- [Dat] Dataflux. <http://www.dataflux.com/>.
- [EE01] Vittorio Castelli (Editor) and Lawrence D. Bergman (Editor). *Image Databases: Search and Retrieval of Digital Imagery*. John Wiley and Sons, 2001.
- [Fir] Firstlogic. <http://www.firstlogic.com/>.
- [FL95] Christos Faloutsos and King-Ip Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD*, pages 163–174, 1995.
- [GD01] Dimitrios Gunopulos and Gautam Das. Time series similarity measures and time series. In *SIGMOD*, 2001.
- [GIJ<sup>+</sup>01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.
- [Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
- [HS95] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.
- [HS98] Gisli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.
- [Jag91] H. V. Jagadish. A retrieval technique for similar shapes. In *SIGMOD*, pages 208–217, 1991.
- [KS97] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.
- [KS98] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE*, pages 466–475, 1998.
- [Kuk92] Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.
- [KW78] Joseph B. Kruskal and Myron Wish. *Multidimensional Scaling*. Sage Publications, Beverly Hills, CA, 1978.
- [Li01] Chen Li. Query processing and optimization in information-integration systems. *Ph.D. Thesis, Computer Science Dept., Stanford Univ.*, 2001.
- [LLLK99] Mong-Li Lee, Tok Wang Ling, Hongjun Lu, and Yee Teng Ko. Cleansing data for mining and warehousing. In *DEXA*, pages 751–760, 1999.
- [Los00] David Loshin. Value added data: merge ahead. *Intelligent Enterprise*, 3(3), 2000.
- [RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.
- [SML00] Hyoseop Shin, Bongki Moon, and Sukho Lee. Adaptive multi-stage distance join processing. In *SIGMOD*, pages 343–354, 2000.
- [SSA97] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-dimensional similarity joins. In *ICDE*, pages 301–311, 1997.
- [Tri] Trillium. <http://www.trillium.com/>.
- [Val] Vality. <http://www.vality.com/>.
- [YK97] Peter N. Yianilos and Kirk G. Kanzelberger. The LIKEIT intelligent string comparison facility. Technical report, NEC Research Institute, 1997.