The VLDB Journal (2008) 17:1213–1229
DOI 10.1007/s00778-007-0061-2

REGULAR PAPER

SEPIA: estimating selectivities of approximate string predicates in large Databases

Liang Jin · Chen Li · Rares Vernica

Received: 8 August 2006 / Revised: 8 May 2007 / Accepted: 14 May 2007 / Published online: 30 May 2008

Abstract Many database applications have the emerging need to support approximate queries that ask for strings that are similar to a given string, such as "name similar to `smith`" and "telephone number similar to `412-0964`". Query optimization needs the selectivity of such an approximate predicate, i.e., the fraction of records in the database that satisfy the condition. In this paper, we study the problem of estimating selectivities of approximate string predicates. We develop a novel technique, called SEPIA, to solve the problem. Given a bag of strings, our technique groups the strings into clusters, builds a histogram structure for each cluster, and constructs a global histogram. It is based on the following intuition: given a query string q, a preselected string p in a cluster, and a string s in the cluster, based on the proximity between q and p, and the proximity between p and s, we can obtain a probability distribution from a global histogram about the similarity between q and s. We give a full specification of the technique using the edit distance metric. We study challenges in adopting this technique, including how to construct the histogram structures, how to use them to do selectivity estimation, and how to alleviate the effect of non-uniform errors in the estimation. We discuss how to extend the techniques to other similarity functions. Our extensive experiments on real data sets show that this technique can accurately estimate selectivities of approximate string predicates.

Keywords SEPIA · Approximate · String · Selectivity · Estimation

A short version of this article appeared as [21] in the proceedings of the 31st International Conference on Very Large Data Bases (VLDB), August 30 – September 2, 2005, Trondheim, Norway. The source code of our algorithms is available at http://flamingo.ics.uci.edu/.

L. Jin · C. Li · R. Vernica (✉)
University of California, Irvine, USA
e-mail: rvernica@ics.uci.edu

L. Jin
e-mail: liangj@ics.uci.edu

C. Li
e-mail: chenli@ics.uci.edu

1 Introduction

Optimizers in database systems need various types of information to improve the performance of query executions. One important type of information is selectivities of query predicates. Consider a table in a database with a large number of employee records, including information such as names, telephone numbers, ages, and salaries. A query can ask for records that satisfy two predicates "age $\leq$ 40 and salary $\geq$ 55". To decide an efficient execution plan, it is critical for the database optimizer to estimate the *selectivity* of each predicate, i.e., the fraction of records in the table that satisfy the predicate. Such information helps the optimizer choose an efficient plan to answer the query.

Textual information is prevalent in databases. Recent applications see an emerging need to support queries with *fuzzy* (approximate) predicates on string attributes, such as "name similar_to `Smith`" and "telephone similar_to `472-0964`", where "similar_to" uses a predefined, domain-specific function to specify the similarity between two strings. Such functions include edit distance or Levenshtein distance [28], cosine similarity [12], Jaccard coefficient distance [11], and variants thereof [9,35]. They could classify `Smith` and `Smyth` to be similar strings. There are many reasons to support queries with fuzzy string predicates. To name a few: (1) the user might not remember exactly the name or the telephone number when issuing the query. (2) There could be typos in the conditions of a query. (3) There could be errors or inconsistencies even in the database, especially in applications such as data cleaning [2,13,15,22,27].

There are recent studies on how to process such a fuzzy predicate efficiently in large databases [e.g., 2,4,13,20]. In order to utilize these techniques to decide an efficient execution plan for a query with fuzzy string predicates (and possibly other predicates), it is important for the query optimizer to know the selectivity of a fuzzy predicate. For instance, consider a query with two predicates "name similar_to `Smith`" and "salary ≥ 85". If there are many records that satisfy the first predicate and only few satisfy the second, processing the second predicate first might be a good choice. On the other hand, if we replace the name `Smith` with a less popular name such as `Schwarzenegger`, then processing the first predicate on the name attribute may produce a good plan (assuming there is no index on the salary attribute).

In this paper, we study how to estimate selectivities of fuzzy string predicates in large databases. Specifically, we are given a string similarity function f, which returns $f(s_1, s_2)$ as the similarity value between two strings s_1 and s_2. Given a bag of strings, a query string q, and a threshold value δ, we want to estimate how many strings s in the bag satisfy the condition $f(q, s) \leq \delta$. The bag of strings can be the values of an attribute in a table in a relational database. In this paper, we use bag of strings and data set interchangeably.

Assume we adopt edit distance for the function f. Our goal thus becomes estimating how many strings in a bag of strings have an edit distance to a given query string within a given threshold. We develop a novel technique, called SEPIA, for solving this problem.[1] Its main idea is to group strings into clusters, and build a histogram for the strings in each cluster. In particular, all the strings within the cluster that have the same proximity from the pivot string are summarized in the histogram. Given a query string q, we look at the proximity v_1 from the string q to the pivot string p in each cluster. We also look at the proximity v_2 from the pivot to each of the strings in the cluster. We obtain a distribution of the similarities between the query string and the strings in the group G, based on this proximity pair (v_1, v_2). We obtain this distribution by analyzing the given collection of strings, and storing the information in a global histogram. This distribution helps us estimate how many strings in the group G satisfy the condition in the query predicate.

In this work, we make the following contributions:

– We propose and fully specify SEPIA as a solution to the problem of estimating selectivities of fuzzy string predicates. To the best of our knowledge, our work is the first attempt to solve this important problem.
– We study challenges in adopting SEPIA, including how to construct effective histogram structures, how to use the structures to do estimation, and how to dynamically maintain the structures in the presence of data changes.
– We study how to extend the technique developed using edit distance to other string similarity functions, using Jaccard coefficient distance as an example.
– We conduct a thorough experimental evaluation of our technique. The results show that our technique can provide accurate selectivity estimations.

The rest of the paper is organized as follows. Section 2 formulates the selectivity estimation problem. Section 3 describes the histograms used in SEPIA. Section 4 studies how to construct and maintain the histogram structures. Section 5 discusses how to improve the estimation accuracy using an error-correction step. Section 6 discusses how to extend the technique to other similarity functions. Section 7 reports the results of our experiments. We conclude the work in Sect. 8.

1.1 Related work

Many techniques have been developed to estimate selectivities of range conditions on single or multiple numeric attributes, [e.g., 18,25,31,32,34]. Most of them are based on summary structures in the form of histograms. They partition the domain of attribute(s) using certain measurement. Based on different partitioning rules, we can have different kinds of histograms, such as equi-width histograms and equi-height histograms [34]. One could extend these histograms and use their partitioning rules for strings based on some order such as their lexicographic order. However, two similar strings might not appear close to each other in such an order. For example, the edit distance between two telephone numbers `412-0964` and `472-0964` is only 1, but they can appear arbitrarily far from each other in a lexicographic order. Thus a histogram based on such an ordering does not provide accurate selectivity estimation for a fuzzy string predicate.

There are studies on estimating selectivities of string predicates with a substring and wildcards such as `name LIKE '%Smith%'`. [17,26] proposed techniques that use summary structures such as pruned suffix trees or Markov tables to store the frequencies of carefully selected substrings. To estimate the selectivity of a wildcard predicate, these techniques divide the query string into disjoint or overlapping substrings, and estimate the selectivity of each substring using the summary structure. They combine these selectivities to compute the selectivity of the query string based on different assumptions. Chaudhuri et al. [5] develop an estimation technique based on a hypothesis called "shortest identifying substring". Informally, it states that the selectivity of a string is close to the selectivity of one of its substrings. Their

[1] "SEPIA" stands for "Selectivity Estimation of approximate PredIcAtes."

approach guesses a set of shortest identifying substrings, and combines the selectivities of those substrings using a regression tree model. [1,29] study how to estimate selectivities of XML path expressions. These techniques cannot be directly adopted to solve our selectivity problem for fuzzy string predicates. In particular, we cannot evaluate a predicate `name similar_to 'Smith'` using the SQL `LIKE` operator because the latter only supports substring matching.

Some string similarity functions, such as edit distance and Jaccard coefficient distance, are metrics. Traina et al. [23] show that many diverse metric data sets follow a "power law" distribution. That is, for a metric-space data set, the average number of neighbors within a given distance r is proportional to r^D, where D is a constant. (A similar intuition was used in [39] for multi-dimensional data sets.) They propose a technique to estimate the D value by building an optimal M-tree for the data set. This technique cannot be applied to solve our problem because of two reasons. First, their technique estimates the *average* number of neighbors in a data set given a distance, while the actual number of neighbors for each individual string could be very different for different strings. Second, our experiments on real string data sets show that this power law property does not hold under similarity functions such as edit distance due to the large number of pairs of words within the same distance [23].

There have been studies on efficiently answering queries with fuzzy string predicates, especially in the context of data cleansing [27,36]. Gravano et al. [13] present a technique to do similar string joins inside a relational database system. Jin et al. [22] develop an efficient approach to approximate string joins using mapping techniques. Chaudhuri et al. [4] propose an indexing structure to support fuzzy queries efficiently. Jin et al. [20] develop a novel indexing structure called "MAT-tree" to support fuzzy predicates with mixed types. The solution in this paper compensates these studies since it can help the query optimizer decide a good execution plan using one of these techniques.

2 Problem formulation

In this section, we introduce basic definitions and formulate the problem of estimating selectivities for approximate (or "fuzzy") string predicates. We focus on selectivity estimation using edit distance. Section 6 discusses how to extend our technique to other similarity functions.

The *edit distance* (a.k.a. Levenshtein distance) between two strings s_1 and s_2, is the minimum number of edit operations of single characters that are needed to transform s_1 to s_2. Edit operations include insertion, deletion, and substitution. We denote the edit distance between two strings s_1 and s_2 as $ed(s_1, s_2)$. For example, $ed($`Michael Jordan`, `Michal Jordon`$)$=2. In particular, to convert the first string to the second, the minimum number of edit operations are to delete the first `e` (in the first string) and substitute the last `a` with an `o`. The edit distance between two strings s_1 and s_2 can be computed using a dynamic programming algorithm, with a time complexity $O(|s_1| \times |s_2|)$, where $|s_1|$ and $|s_2|$ are the length of s_1 and s_2, respectively [37]. If a distance threshold is given, a linear time algorithm (assuming the threshold is constant) has been proposed in [40] (also known as "the early termination algorithm"). There has been a large amount of work on improving the basic algorithm. See [33] for an excellent survey.

Let $\mathcal{B}$ be a bag with $|\mathcal{B}|$ strings. These strings can be the values of an attribute in a relational table, such as names in an employee table. An approximate string predicate P is a triplet $\langle ed, q, \delta \rangle$, where ed is the edit distance function, q is a query string, and δ is a distance threshold. A string s in the bag $\mathcal{B}$ satisfies this predicate if $ed(q, s) \leq \delta$. The *frequency* $f(P)$ of the predicate is the number of strings in $\mathcal{B}$ that satisfy the predicate. The *selectivity* of the predicate is $f(P)/|\mathcal{B}|$. We assume the size $|\mathcal{B}|$ is known. Thus our problem becomes the following.

Problem Statement: Given an approximate string predicate $P = \langle ed, q, \delta \rangle$ on a bag of strings, estimate how many strings s in the bag satisfy the predicate, i.e., $ed(q, s) \leq \delta$.

3 Selectivity estimation Using SEPIA

This section presents our novel approach to the problem above. Its main idea is to group the given bag of strings into disjoint clusters, and construct histogram structures to support estimation. Figure 1 illustrates the intuition behind our technique. Given an approximate predicate $P = \langle ed, q, \delta \rangle$, consider a *pivot* string p in a cluster of strings. Let v_1 be a measure of the proximity between q and p. (We will discuss how to choose the pivot string and measure the proximity shortly.) For each string s in the cluster, let v_2 be a measure of the proximity between p and s. If we have a probability

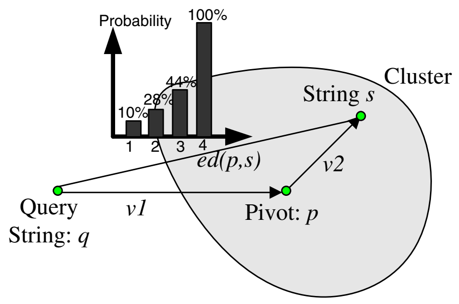


Fig. 1 Intuition of SEPIA

distribution of the edit distance between q and s, then we can utilize this distribution to compute the expected number of strings in the cluster with this proximity v_2 that satisfy the query predicate. This distribution depends on the proximity v_1 and the proximity v_2. We can analyze the strings in the data set to obtain such a distribution.

3.1 Measuring string proximity

A simple, natural way to represent the proximity between two strings is to use their edit distance. However, this number is imprecise in terms of differentiating strings with the same edit distance to a pivot string. For instance, Fig. 2a shows a cluster with a pivot string $p =$ `lucia`. Two strings in the cluster, `lucas` and `luciano`, have the same edit distance 2 to p. Consider an approximate query predicate $P = \langle ed, \texttt{lukas}, 3\rangle$. String `lucas` is closer to the query string (with an edit distance 1) than `luciano` (with an edit distance 4), but we cannot differentiate these two strings based on their edit distance to the pivot string p. In other words, we need a more discriminative representation of the edit distance in order to differentiate strings in a cluster based on their proximities to the pivot.

To more precisely describe the proximity between two strings, we introduce a new representation, called *edit vector*, to keep track of the edit operations during the computation of the edit distance between the strings.

Definition 1 Edit Vector Let s_1 and s_2 be two strings. An *edit vector from* s_1 *to* s_2 is a three-number vector in the form $\langle I, D, S\rangle$, in which I, D, and S are the number of insertions, deletions, and substitutions, respectively, in a sequence of edit operations of single characters that transforms s_1 to s_2 with the minimum number of edit operations. Let v be such an edit vector. Clearly the edit distance between the two strings is $|v| = I + D + S$.

For instance, an edit vector from string `lucia` to `luciano` is $\langle 2, 0, 0\rangle$, since two character insertions are needed to transform the former to the latter. An edit vector from string `lucia` to `lucas` is $\langle 1, 1, 0\rangle$, since we need an insertion and a deletion of single characters to transform the former to the latter. Figure 2b shows the edit vectors for

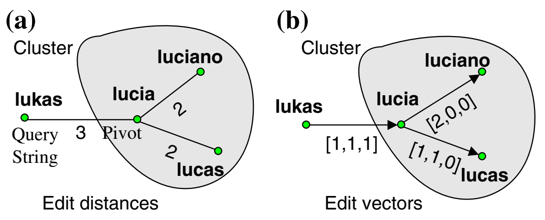


Fig. 2 Using edit vectors to better describe string proximity than edit distances

some of the string pairs. The advantage of using edit vectors over edit distances is that edit vectors are more discriminative for string pairs with the same edit distance, while it can still maintain the edit distance information between the strings.

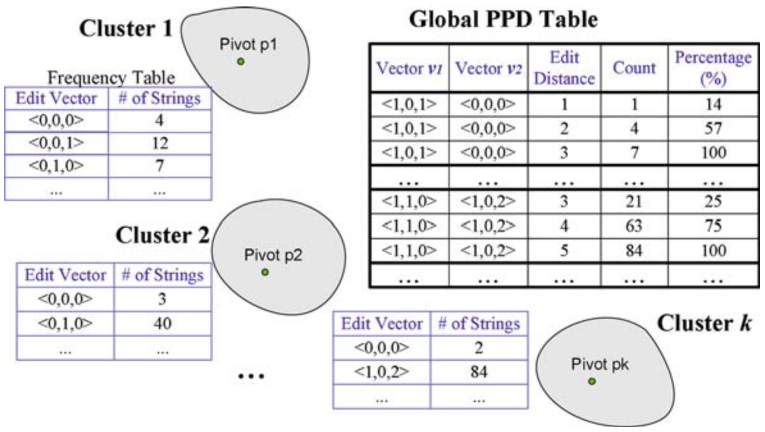
There can be different edit vectors from a string s_1 and a string s_2, since there can be different sequences with the minimum number of edit operations. We can choose any of them as a representation of their proximity. Our experiments with real data strings showed that there tends to be a unique edit vector from a string to a similar string. In our experiments, more than 91% of string pairs with an edit distance within three have a unique edit vector. We can compute an edit vector from s_1 to s_2 by slightly modifying the dynamic programming algorithm that computes their edit distance. Notice that an edit vector from s_1 to s_2 might not be the same as an edit vector from s_2 to s_1, i.e., edit vectors are not symmetric.

3.2 Histogram structures

Figure 3 illustrates the histogram structures used in our approach. We group the strings into clusters. Let $C_1, \ldots, C_k$ be the clusters. For each of them C_i, we choose one of its strings as the *pivot* for the cluster. This pivot, denoted as p_i, is a representative of the strings in C_i. In Sect. 4.1, we describe the details of this step. For the purpose of easy dynamic maintenance, the decided pivot, even though corresponds to a string s in the cluster, is maintained separately from the cluster. That is, this pivot string is never deleted even if the original string s is deleted. (The idea of selecting a pivot for a cluster is also adopted in [3,6,10,16,19,41].) This string is selected in such a way that it is close to the strings in C_i. We also keep the radius r_i of this cluster, which is the maximum edit distance between p_i and any string in the cluster.

Frequency tables: for each cluster, we compute an edit vector from its pivot to each string in the cluster. We group these strings based on their edit vector from the pivot. We summarize these strings by storing the number of strings for each group. These numbers are stored in a structure, called "frequency table", for this cluster. Each entry in the table has an edit vector and the number of strings that have this edit vector from the pivot. For instance, the frequency table for cluster 1 shows that 4 strings in this cluster have an edit vector $\langle 0, 0, 0\rangle$ from the pivot string (i.e., there are 4 identical strings), 12 strings with the edit vector $\langle 0, 0, 1\rangle$ from the pivot, and 7 strings with $\langle 0, 1, 0\rangle$. Intuitively, this table summarizes the distribution of the strings in the cluster in terms of their proximity from the pivot.

PPD table: We also construct a histogram, called "proximity-pair-distribution table" ("PPD table" for short), to store global statistical information about the strings in the data set. As illustrated in Fig. 1, the goal of having this histogram is to store information about the edit-distance distribution given a



Edit Vector	# of Strings
<0,0,0>	4
<0,0,1>	12
<0,1,0>	7
...	...

Edit Vector	# of Strings
<0,0,0>	3
<0,1,0>	40
...	...

Vector *v1*	Vector *v2*	Edit Distance	Count	Percentage (%)
<1,0,1>	<0,0,0>	1	1	14
<1,0,1>	<0,0,0>	2	4	57
<1,0,1>	<0,0,0>	3	7	100
...	...	...	...	...
<1,1,0>	<1,0,2>	3	21	25
<1,1,0>	<1,0,2>	4	63	75
<1,1,0>	<1,0,2>	5	84	100
...	...	...	...	...

Edit Vector	# of Strings
<0,0,0>	2
<1,0,2>	84
...	...

Fig. 3 Histograms used in SEPIA

pair (v_1, v_2) of edit vectors. Each entry in the table is in the format of:

(Edit Vector, Edit Vector, EditDist, Count, Percentage).

Each entry (v_1, v_2, e, c, f) means that, for a query string that has an edit vector v_1 to the pivot of a cluster, among all the strings in the cluster that have an edit vector v_2 from the pivot, statistically, on the average f (percentage) of these strings have an edit distance within e to the query string. The count c is the number of generated triplets (v_1, v_2, e') (where $e' \le e$) in the construction of this table (discussed in Sect. 4.2). We keep this count in order to support incremental maintenance of this table.

For instance, consider the PPD table in Fig. 3. The first three entries mean the following. If a query string has an edit vector $\langle 1, 0, 1\rangle$ to the pivot of a cluster, for all the strings in the cluster that have an edit vector $\langle 0, 0, 0\rangle$ from the pivot, on the average (over all the clusters) 14, 57, and 100% of these strings have an edit distance within 1, 2, and 3, respectively, to the query string. In addition, during the construction of this table, there are 1, 4, and 7 triplets (v_1, v_2, e') for all the clusters, where e' is within 1, 2, and 3, respectively. To support efficient lookups, the PPD table can be implemented as a hash table using the first three values of each entry as the hash key.

3.3 Frequency estimation

In this subsection we present our frequency estimation algorithm. The intuition is the following. Given an approximate string predicate that contains the similarity function, a query string, and a distance threshold, we go over each of the clusters. For each cluster, using its frequency table we select the candidate edit vectors. For each edit vector, we do a look-up in the PPD table and retrieve the corresponding percentage value. Using the frequencies from the frequency tables and the percentages from the PPD table, we can compute the estimated selectivity of the predicate.

```
Data Structures:
  • String clusters $C_1, \ldots, C_k$, each $C_i$ has a pivot
    string $p_i$, a radius $r_i$, and a frequency table $FT_i$.
  • Global proximity-pair-distribution table $PPD$.
Estimation Algorithm
Input: Approximate string predicate $\langle ed, q, \delta\rangle$, where
  • $q$ is a query string, and
  • $\delta$ is an edit-distance threshold.
Output: Estimated # of strings satisfying the predicate.
Method:
 $est \leftarrow 0$;
 for (i = 1 to k) {
     Compute an edit vector $v_1$ from $q$ to $p_i$;
     if ($|v_1| > r_i + \delta$) continue; // ignore this cluster
     for (each entry $(v_2, n)$ in $FT_i$) {
         if ($|v_1| + |v_2| \le \delta$)) { $est \leftarrow est + n$; continue;}
         if ($||v_1| - |v_2|| > \delta$)) continue;
         Use $(v_1, v_2, \delta)$ to find a percentage $f$ in $PPD$;
         if ($f$ is not found) continue; // no such entry
         $est \leftarrow est + f \times n$;
     }
 }
 return ($est$);
```

Fig. 4 Estimation algorithm

Figure 4 shows the pseudo code of our estimation algorithm. To estimate the frequency of an approximate string predicate $P = \langle ed, q, \delta\rangle$, we scan through the pivots of the clusters. For the pivot p_i of cluster C_i with a radius r_i, we compute an edit vector v_1 from q to p_i and their edit distance $ed(q, p_i)$. If $|v_1| > r_i + \delta$, based on the triangular inequality, we can ignore this cluster since no string in this cluster can satisfy the predicate.

For each remaining cluster with a pivot p_i, we go through the entries in its frequency table. Recall that each entry (v_2, n) in the frequency table means that there are n strings in this cluster with an edit vector v_2 from the pivot p_i. If $|v_1| + |v_2| \le \delta$, by the triangular inequality, all these n strings satisfy the predicate, so we add n to the total estimation. If $||v_1| - |v_2|| > \delta$, then we can ignore this entry based on the triangular inequality. Otherwise, we use the triplet (v_1, v_2, δ) to look up the PPD table, and find the corresponding percentage f. The product of f and n gives us an estimation about how many in these n strings have an edit distance within δ to the query string q. We take the sum of these products for different v_2 vectors in this cluster, and for all the clusters.

Complexity analysis: The outer loop is executed k times, where k is the number of clusters. For each cluster C_i, the inner loop is executed $s(C_i)$ times, where $s(C_i)$ is the number of entries in the frequency table of C_i. The worst case of the complexity is when no clusters can be pruned, and no entries in the frequency tables can be pruned. In our experiments we will analyze the performance of this estimation algorithm.

4 Construction and maintenance

In this section, we study how to construct and maintain the histograms in Sepia.

4.1 Clustering strings

Clustering has been studied in the literature due to its importance in many applications. Clustering algorithms developed for Euclidean spaces (e.g., the k-means algorithm [14]) are not directly applicable in our case, since edit distance does not form a Euclidean space.

We need to consider two factors when generating clusters of strings. The first one is the quality of each cluster. We want to group similar strings into one cluster. The pivot we choose for a cluster is a representative of the strings in the cluster. The more similar the strings are to this pivot, when we use a pair of edit vectors to look up the global PPD table during an estimation, the more accurate the distribution information we can get from the PPD table. We can measure the quality of a cluster as the average edit distance from the strings in the cluster to its pivot. The smaller the average edit distance is, the better the cluster is.

The second factor is the number of clusters. As the number of clusters increases, we will have a better chance to improve the quality of each cluster. On the other hand, increasing this number can also increase other costs: (1) the number of frequency tables will increase. This increase of clusters might decrease the size of each table. Our experiments showed that the overall size of the index structure tends to increase. (2) When estimating the frequency of a fuzzy predicate, the estimation time can also increase since we need to scan through the pivots. In particular, we need to compute an edit vector from the query string to each pivot. Our experiments showed that the total estimation time is mainly dominated by this computation. Thus, in order to reduce the estimation time and histogram space, we need to restrict the number of clusters in our histogram structures. We present two algorithms for clustering strings, which are experimentally evaluated.

Clustering based on lexicographic order: one naive clustering method is to group strings based on their lexicographic order. We could adopt the idea in equi-height histograms [34] by partitioning the range into k segments (clusters) with the same number of strings in each segment. Within each cluster, we can choose a string in the "middle" as the pivot. This approach is based on the assumption that strings close in their lexicographic order tend to have a small edit distance, such as `university` and `universal`.

Clustering using k-Medoids methods: We can cluster strings using the k-Medoids algorithm [24] based on the idea of "Partitioning Around Medoids", or "PAM". The medoid concept used in these algorithms is the same as our *pivot* concept. Using the k-medoids algorithm, we proceed in two steps. In the first step (called "BUILD"), we select an arbitrary collection of k strings from the data set as initial pivots, and assign each remaining object to the closest pivot according to their edit distances. We define an objective function as the total distance between each string in a cluster and its pivot. In the second step (called "SWAP"), we try to reduce the value of the objective function by swapping a selected pivot with an unselected string. We pick the pair that can best improve the objective function, swap the pair, and re-distribute the remaining strings to the new pivots. We repeat this step till the value of the objective function can no longer be decreased.

We can further improve the efficiency of the basic algorithm by adopting the idea in [24]. We sample the data set several times (e.g., five times). For each sample, we do the BUILD step described above, and calculate the value of the objective function. The pivots of the sample with the minimum objective function value are chosen as the final pivots of the entire data set. Each remaining object is assigned to the closest pivot to form clusters. We call this method *k-Medoids* [24].

Another way to improve the efficiency of the basic algorithm is to choose a subset of the objects as a sample, and get the best set of pivots for the sample. In order to get the best set of pivots, we select a random set of pivots and we apply the SWAP step described above until the clusters cannot be improved. We call this method *k-Medoids**.

4.2 Constructing histogram structures

Frequency tables: For each cluster, we calculate the edit vector from the pivot string to every string inside the cluster. We construct the frequency table by counting how many strings exist in the cluster for each unique edit vector.

PPD Table: To populate the PPD table, we need to gather enough samples of string triplets (q, p, s), where q is a string in a fuzzy predicate, p is the pivot in a cluster, and s is a string in the cluster. Once we have enough such string triplets, we calculate an edit vector v_1 (from q to p) and an edit vector v_2 from p to s. We also compute the edit distance $e = ed(q, s)$. After generating enough such triplets T, for each unique triplet (v_1, v_2, e) in T, we insert a record $(v_1, v_2, e, c, c/A)$ into the PPD table, where c is the total number of occurrences of triplets (v_1, v_2, e') in T, where $e' \leq e$, and A is the total number of occurrences of the pair (v_1, v_2) in T.

There are different ways to generate samples of string triplets (q, p, s). Recall in Fig. 4, when we look up the PPD table during an estimation, if a pair (v_1, v_2) does not appear in the table, we assume no string with an edit vector v_2 from the pivot string can satisfy the fuzzy predicate. Thus we want to generate sample triplets to cover as many (v_1, v_2) pairs as possible to avoid possible miss hits during an estimation. On the other hand, we also need to consider the running time when generating sample string triplets, due to the cost of computing edit vectors. We present the following methods for generating sample string triplets, which are experimentally evaluated (Sect. 7).

– `ALL_RAND`: The method randomly samples a small number of strings in the data set as query strings in fuzzy predicates. It generates a collection of string triplets (q, p, s) by considering each of the query strings, the pivot of each cluster, and each string in the cluster.
– `CLOSE_RAND`: The method is similar to `ALL_RAND` except that, for each query string, it only considers, say, 10 closest pivots to the query string based on edit distances.
– `CLOSE_LEX`: This method is different from `CLOSE_RAND` in the way they generate query strings. `CLOSE_LEX` sorts the strings in the data lexicographically, and uniformly selects sample strings in the order.
– `CLOSE_UNIQUE`: This method is different from `CLOSE_RAND` in the way they generate query strings. In the process of generating random query strings, `CLOSE_UNIQUE` keeps a sample string only if the string can generate at least a certain number, say 10, of new v_1 edit vectors. The objective of `CLOSE_UNIQUE` is to generate as many unique edit-vector pairs as possible.

4.3 Dynamic maintenance

The frequency tables for the clusters can be easily maintained in the presence of data updates. If a new string s_{new} is inserted into the data set, we add it to its closest cluster C. We compute an edit vector v_2 from the pivot of C to s_{new}, and increment the count of this vector v_2 in the frequency table of C. We can modify the radius of this cluster if needed. The case of deleting a string can be dealt with in a similar manner.

In order to incrementally maintain the global PPD table when the bag of strings is updated, we need to consider the effect of insertions and deletions on the table. In Sect. 4.2 we discussed how to populate the PPD table by generating samples of string triplets (q, p, s), in which q is from a *small* number of strings in a workload of fuzzy predicates. Let S denote the set of these query strings. To support incremental maintenance of the PPD table, we keep these query strings in S. For each pivot string p_i, we also store the precomputed edit vectors from these strings to the pivot p_i.

Consider a new string s_{new} inserted into the data set. Let C be the cluster whose pivot p is closest to s_{new} among all the pivots, and we modify the frequency table of this cluster (as described above). We compute an edit vector v_2 from p to the new string. For each string q in S, we compute the edit distance $ed(q, s_{\text{new}})$. We use the (precomputed) edit vector v_1 from q to p to form a new proximity pair (v_1, v_2). We use this pair to look up the PPD table and locate entries with this pair. For each entry (v_1, v_2, e, c, f), we increase the count c by one if $e \geq ed(q, s_{\text{new}})$. Accordingly we modify the f percentage values for these entries. Since the number of strings in S is small, the cost of this incremental maintenance is small, as shown in our experiments. The PPD table can be maintained in a similar manner when existing strings are deleted. Notice that the pivots are *not* part of the data set. As many other histogram structures proposed in the literature, if there are enough insertions/deletions in the bag of strings, we may need to reconstruct the histogram structures in order to support accurate estimations.

5 Improving estimation accuracy

Estimated frequencies using the histograms in SEPIA could be different from the real frequencies mainly due to two reasons. The first one is that the percentage entries in the PPD table may not be accurate. The second one is due to miss hits of the PPD table, i.e., a proximity pair during an estimation does not exist in the PPD table. In this section we address these problems, and develop an approach for further improving the accuracy.

5.1 Choosing string-proximity metric

One way to improve the representativeness of the entries in the PPD table is to choose a good string-proximity metric, based on which the histograms are constructed. The intuition behind using a histogram is to show the number of strings in the data set that share the same value over a chosen metric.

Let us revisit the edit vector proximity metric presented in Sect. 3.1. We started from the edit distance metric, and showed why this metric is too general to be used for some strings. That is, different strings could have the same edit distance to a given string. Then, we introduced the edit vector metric. The main difference is that an edit distance is a single value, while an edit vector is a 3-D vector. Figure 2 showed why edit vector could be more appropriate than edit distance to differentiate different strings based on their similarity to a given string.

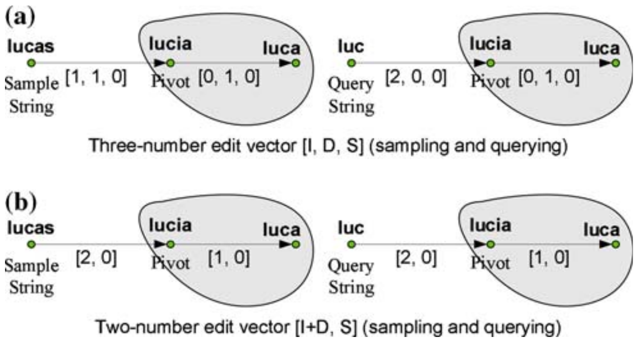


Fig. 5 Difference between using a three-number edit vector $\langle I, D, S\rangle$, and a two-number edit vector $\langle I + D, S\rangle$

Each entry in the PPD table can be viewed as a point in a 7-D space, since it has three values for the first edit vector, three values for the second, and one value for the edit distance. The edit vector metric might lead to a very sparse 7-D space. To address this concern, instead of having a three-number edit vector (insertion, deletion, and substitution), we can build a two-number edit vector, by grouping two of the three numbers into one. We can group insertion and deletion, and the form of the edit vector will be $\langle I + D, S\rangle$. Similarly we can also group insertion and substitution, or deletion and substitution. By doing this, we decrease the dimensionality of the edit vector from 3 to 2. Correspondingly, each entry in the PPD table can be viewed as a point in a 5-D space.

We use an example (Fig. 5) to show a case where a two-value proximity metric can be better than the three-value edit vector metric. For illustration purposes, let us assume we have a single cluster with a pivot `lucia`, and a string `luca`. A sample string used in constructing the PPD table is `lucas`, and a query string (for selectivity estimation) is `luc` with an edit distance threshold 2. Figure 5a shows the case where we use the 3-value edit vector as the proximity metric. The only entry in the PPD table has two edit vectors, $\langle 1, 1, 0\rangle$ and $\langle 0, 1, 0\rangle$. During the selectivity estimation for the query string, we use the combination of $\langle 2, 0, 0\rangle$ and $\langle 0, 1, 0\rangle$ to lookup the PPD table, which does not have such an entry. Thus our estimated selectivity is 0, even though the real selectivity is 1, since $ed(\texttt{luc}, \texttt{luca}) = 1$, which is less than 2. Figure 5b shows the case where we use $\langle I + D, S\rangle$ as the proximity metric. The only entry in the PPD table will have two edit vectors, $\langle 2, 0\rangle$ and $\langle 1, 0\rangle$. When estimating the selectivity for the query predicate, we use the combination of $\langle 2, 0\rangle$ and $\langle 1, 0\rangle$ to lookup the PPD table, which does have such an entry. Our estimated selectivity is 1, which is equal to the real selectivity. What proximity metric is the best for selectivity estimation depends on the data set, such as its number of entries and their distribution. In the experiments we have compared different proximity metrics (Sect. 7).

5.2 Increasing lookup hit rate

During the frequency estimation described in Sect. 3.3, if a pair of edit vectors does not exist in the PPD table, we assume the selectivity is 0 for this pair of edit vectors. This can cause an underestimation of the selectivity.

One way to increase the PPD lookup hit rate during estimation is to increase the number of samples in the construction of the PPD table. As the number of samples increases, there are more entries in the PPD table. As a result, the number of successful edit vector lookups in the table during an estimation also increases. This can alleviate the underestimation problem. On the other hand, as the number of samples increases, the percentages in the PPD table will also change, which will affect the estimation accuracy. Our experiments show that these percentages tend to give us an overestimation. So the overall effect on the estimation accuracy is the combined result of this underestimation due to the missing entries in the PPD table and the overestimation of the percentages. In Sect. 7 we will show the effect of the number of samples on the estimation accuracy.

Another way to increase the lookup hit rate is the following. During the construction of the PPD table, for each query string, we can increase the number of pivots that we consider for generating the sample triplets. (Notice that `CLOSE_RAND` considers only the 10 closest pivots.) In this way, we will have a better chance to produce more triplets in the PPD table.

5.3 Reducing size of PPD table

In order to do selectivity estimation efficiently, it is important to keep the histograms in memory. Since the PPD table takes the largest amount of space to store, it is critical to reduce the size of this table. We present the following methods to reduce this size. In all of the following methods we store the PPD table in binary format, as opposed to the original text format.

The first method relies on *pruning*. As we described in Sect. 3.2, an entry in the PPD table is in the following format:

(Edit Vector, Edit Vector, EditDist, Count, Percentage),

When the PPD table is used to estimate the selectivity of a predicate, a lookup in the PPD table is done for entries with corresponding "Edit Vectors" and "EditDist" within the threshold specified in the query. If the threshold tends to be small (e.g., within 5), we can delete from the PPD table those entries that have an "EditDist" greater than 5. The reason is that we will never look up the PPD table for entries with an "EditDist" greater than 5.

The second method is to store the "Edit Vectors" and the "EditDist" values efficiently. Since these values tend to be small, we can represent each of them using a short integer or even a byte. Similarly, we can store the floating point "Percentage" value, as a fixed-size integer. In order to do this, we truncate the value and consider only its first four digits after 0. We also apply the pruning step from the first method.

Using the third method, we could even remove the "Percentage" field completely and compute it on the fly whenever it is needed. That is, we compute this value in the selectivity estimation step. In order to compute it, we have to keep in the PPD table the entry that contains the maximum count value for each of the "Edit Vector" pairs. Because of this, we can only partially apply the first method. We apply the second method in its full extent.

5.4 Improving estimation accuracy by error correction

In addition to considering the previous factors that can affect the estimation accuracy, we also propose the following approach to further improve the accuracy of the initial estimation using the PPD table. The main idea is to use a small number of query strings, and use their estimated frequencies and real selectivities to build a model. Given the initial estimation for a query string, we apply this model to the initial estimation to get a new, more accurate estimation. (A similar idea is used in [30].)

We select a small number of strings to generate a workload of fuzzy string predicates. The threshold of a predicate is chosen randomly within an edit distance range. We estimate the frequencies of these predicates using the PPD table. We also compute their real frequencies by computing how many strings satisfy each query predicate. We analyze the relative error for each estimation, defined as $(f_{\text{est}} - f_{\text{real}})/f_{\text{real}}$, where f_{est} is the estimated frequency, and f_{real} is the real frequency. Figure 6 shows a simple example of such errors for the frequency estimations of four predicates $P_1, \ldots, P_4$. For instance, the estimated frequency for predicate P_1 is 750, while the real frequency is 500. The relative error for this estimation is +50%, which is an overestimation.

Based on the relative errors of the four queries, we obtain an error distribution model, in which probabilistically 25% of the predicates have an estimation with a relative error −40%; 50% of the predicates have an estimation with a relative error +50%; and 25% of the predicates will produce an accurate estimation. The average relative error is +15%. Given an initial estimation for a query predicate, we can use this average error to adjust the initial estimation, so that probabilistically we can bring it closer to the real frequency.

Predicates	Real	Estimate	Relative Error
P1(tommy,2)	500	750	+50%
P2(james,3)	400	600	+50%
P3(jordan, 2)	600	600	0%
P4(david, 2)	500	300	-40%

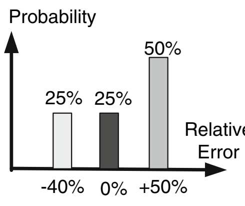


Fig. 6 Relative errors of predicates

The relative error for each estimation is related to factors such as the initial estimation, the length of the query string, and the threshold in the predicate. Our experiments show the following. The initial estimation tends to be an overestimation for longer query strings, and large initial estimations are usually an underestimation. One reason is the following. With the same edit distance threshold, the longer the query string is, the more likely it has a small frequency. The percentage values in the PPD table are average values. Therefore, longer strings tend to suffer from overestimations, and shorter strings can suffer from underestimations. These observations suggest that we cannot build a universal error distribution model for all predicates.

We use a decision tree to compute the expected relative error for the initial estimation of a fuzzy predicate based on its string length, threshold, and initial estimation. We use a workload of fuzzy predicates to produce this decision tree. Figure 7 shows such a decision tree. In each intermediate node, we use the three factors as the conditions on the branches. Each leaf node has an average relative error for those predicates that satisfy the conditions on the path from the root to this leaf node. Take the leftmost leaf node as an example. It means that for all fuzzy predicates with a threshold $\delta = 1$, a query string with length between 1 and 5, whose initial estimated frequency is within 40, the average relative error is −15%.

Given a fuzzy string predicate $P(q, \delta)$, we traverse the tree and identify a leaf node based on its δ value, the length L of q, and the initial estimate IE using the PPD table. We use the average relative error r at the leaf node to compute a new estimated frequency. That is, we return $\frac{IE}{r+1}$ as the final estimated frequency based on the definition $(f_{\text{est}} - f_{\text{real}})/f_{\text{real}}$ of relative error. In our experiments we will show that this step can effectively improve the accuracy of estimations.

6 Extensions to other functions

Our SEPIA approach provides a general framework for frequency estimation of fuzzy string predicates. It can be extended to other similarity functions. Let $\mathcal{F}$ be such a function. When we group strings to clusters, we should use $\mathcal{F}$ as a distance function to measure the similarity between two strings. One important issue in the extension is how to measure the proximity between two strings. In the edit distance case, we use the concept of edit vector to represent such a proximity. For a new function $\mathcal{F}$, we need to develop such a measure as a good representation for the string proximity. We need to consider the tradeoff between the specificity

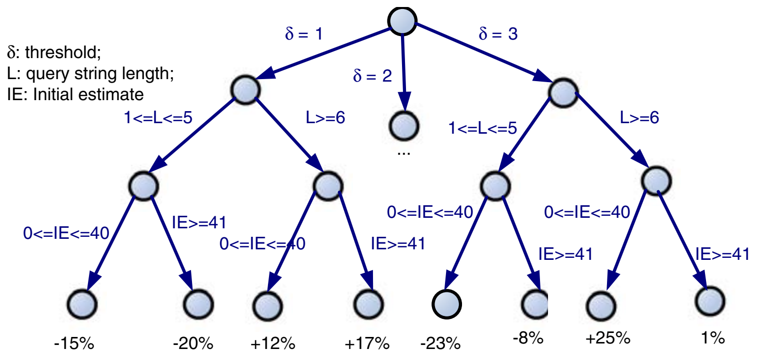


Fig. 7 Decision tree to compute average relative error of initial estimation

and generality of such a measure. On one hand, this measure should be specific enough so that it can differentiate strings within a cluster, based on their proximity from the pivot of the cluster (See, for example, Fig. 2). On the other hand, the measure cannot be too specific either, since otherwise we cannot have enough samples in each proximity pair in the PPD table to obtain a meaningful probability distribution. The reason is that it becomes more likely that different strings in a cluster have different proximities from the pivot string.

We use the *Jaccard coefficient distance* function as an example to show how to extend SEPIA to a new similarity function. Let us first revisit its definition. Let n be an integer. Given a string s, the set of *n-grams of* s, denoted $G(s, n)$, is obtained by sliding a window of length n over the characters of string s. For instance, if $n = 3$:

G(“Michael Jordon”)={’Mic’, ’ich’, ’cha’, ’hae’, ’ael’, ’el ’, ’l J’, ’ Jo’, ’Jor’, ’ord’, ’rdo’, ’don’}.
G(“Michal Jordan”)={’Mic’, ’ich’, ’cha’, ’hal’, ’al ’, ’l J’, ’ Jo ’, ’Jor’, ’ord’, ’rda’, ’dan’}.

The *Jaccard Coefficient Distance* [9] between two strings s_1 and s_2 for an integer n, denoted $jcd(s_1, s_2, n)$, is defined as:

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

For example, jcd(“Michael Jordon”, “Michal Jordan”, 3) $= 1 - \frac{7}{16} \approx 0.56$. The smaller the Jaccard coefficient distance between two strings is, the more similar they are.

To adopt SEPIA to estimate frequencies of fuzzy predicates using Jaccard coefficient distance, instead of using edit vector, we need a new measure that is discriminative enough, and retains the semantics of proximity between strings as well. Following this principle, we use the following vector between two strings, s_1 and s_2, as a proximity representation:

$$\langle |G(s_1, n) \cap G(s_2, n)|, |G(s_1, n) \cup G(s_2, n)|, ed(s_1, s_2)\rangle.$$

Instead of using edit vector, we use this new proximity vector to construct the frequency tables of different clusters, and the global PPD table. Our experiments show that this proximity measure works comparably well for Jaccard coefficient distance.

7 Experiments

This section presents the results of our extensive experiments.

7.1 Experimental setting

We ran our experiments on two data sets of 100,000 collected from the Internet Movie Database (IMDB)[2]. The first data set consisted of actor last names. The length of each name varied from 2 to 51, and the average length was around 7. The data set had around 55,000 unique names. The number of duplicates was between 1 and 279. The second data set consisted of movie titles. The length of each title varied between 1 and 162, and the average length was around 19. The data set had around 30,000 unique names.

We evaluated the accuracy of SEPIA using a workload of query predicates. The predicates were randomly selected from the data set. The edit distance threshold of each predicate was a random integer between 1 and 4.

To evaluate the accuracy of an estimation for a fuzzy predicate, we used its *relative error*, defined as $(f_{\text{est}} - f_{\text{real}})/f_{\text{real}}$, where f_{est} is the estimated frequency, f_{real} is the real frequency of this predicate. Correspondingly, we define its *absolute relative error* as $|f_{\text{est}} - f_{\text{real}}|/f_{\text{real}}$. Compared to related studies [5,7,29] that use measures that favor large selectivities, the results using this measure show that SEPIA can provide accurate estimates for both small and large selectivities.

We implemented SEPIA using C++ and the STL library [38]. All the experiments were run on a PC, with Dual AMD Opteron 1.4 GHz (64 bit) processors and 1,024MB memory. The operating system was Redhat Linux 8.0.

7.2 Clustering algorithms

We implemented the lexicographic-based clustering algorithm, the k-Medoids clustering algorithm, and the k-Medoids* clustering algorithm, as discussed in Sect. 4.1. We fixed the number of clusters to 1,000.

Measuring quality of clusters: One way to measure the overall quality of the clusters is to use a histogram that shows

[2] `http://www.imdb.com/interfaces#plain`.

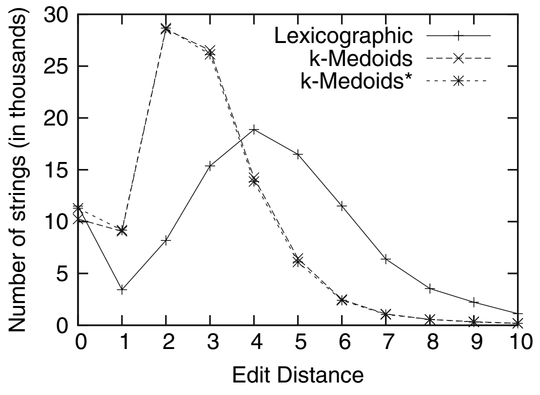


Fig. 8 Histograms of the edit distances between a pivot and the strings in the same cluster

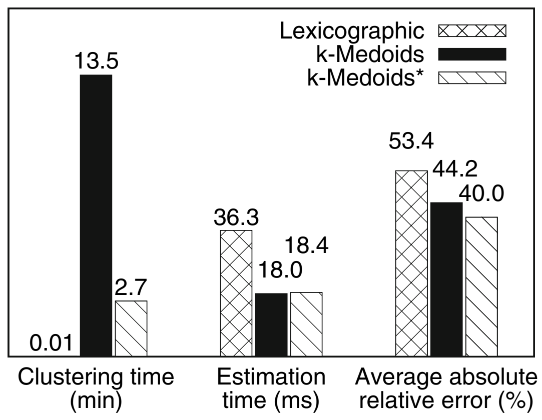


Fig. 10 Performances of three clustering algorithms

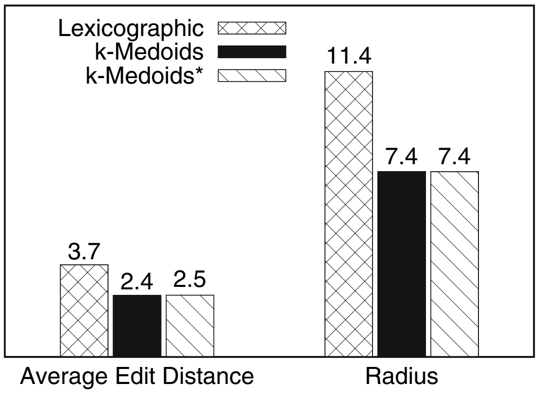


Fig. 9 Quality of clusters generated by three clustering algorithms

the number of strings within an edit distance from the pivot of a cluster, for all the clusters. Intuitively, a set of clusters has a good quality if most of its strings are within a small edit distance from the corresponding pivot. Figure 8 shows the histograms of the edit distances for the clusters generated by the three clustering algorithms. Compared to the lexicographic-based algorithm, the two k-Medoids variants have more strings that have a small edit distance to their pivot (within 3), and fewer strings with a large edit distance (greater than 3). That is, the clusters generated by the two k-Medoids variants had similar quality, which was better than the clusters created by the lexicographic-based method.

Another way to measure the overall quality of the clusters is the following. For each cluster we compute the average edit distance between its pivot and its strings. We then take the average of these distances for all the clusters. Similarly, we compute the maximum edit distance (radius) between the pivot and its strings for each cluster, and take the average of these radii for all the clusters. A set of clusters with a smaller average edit distance and radius has a good quality. Figure 9 shows the average edit distance and the average radius for the 1,000 clusters generated by the three clustering methods. Again, it shows that the k-Medoids and k-Medoids* algorithms generated better clusters than the lexicographic-based algorithm.

Estimation accuracy: After running each clustering algorithm, we applied the `CLOSE_RAND` heuristic to populate the PPD table. We ran our estimation algorithm using the testing query load to calculate the average absolute relative error. In these experiments we did not apply the error-correction step discussed in Sect. 5.4.

Figure 10 shows the results of the three clustering algorithms. The k-Medoids algorithm took the most time (13.5 min) to finish the clustering step and the lexicographic-based algorithm was the fastest (0.01 min). The k-Medoids* algorithm had a much shorter time (2.7 min) that the k-Medoids algorithm. The estimation times for the k-Medoids (18.0 ms) and k-Medoids* (18.4 ms) were similar. The estimation time for the lexicographic-based algorithm took twice the amount of time needed for the k-Medoids variants. For the average absolute relative error, k-Medoids* (40.0%) was the best, followed by k-Medoids (44.2%), and the lexicographic-based algorithm (53.4%). Thus we chose the k-Medoids* algorithm to cluster strings in all the remaining experiments.

7.3 Populating PPD table

We experimentally evaluated the four different heuristics for constructing the PPD table, namely, `ALL_RAND`, `CLOSE_RAND`, `CLOSE_LEX`, and `CLOSE_UNIQUE`, as discussed in Sect. 4.2. We first generated 1,000 clusters using k-Medoids*. For each heuristic, we sampled 20% of the strings as strings in a workload of fuzzy predicates. We collected the average absolute relative error for each heuristic. Figure 11 shows the details of the sampling time, number of entries, and error comparisons.

The running time for the `CLOSE_RAND` was the largest (20.9 min), because its closeness condition is based on edit distance which is expensive to compute. `CLOSE_UNIQUE` was the fastest (5.4 min), since the number of unique v_1 edit

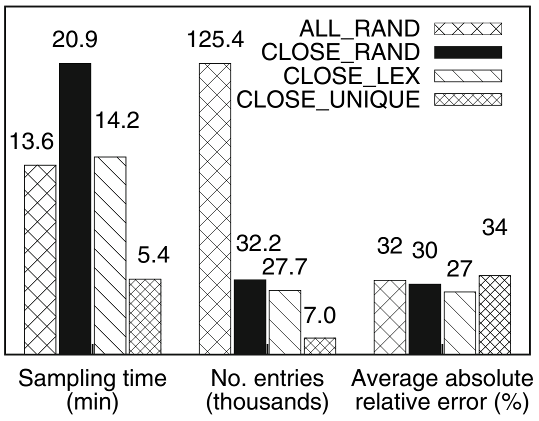


Fig. 11 Different heuristics for populating the PPD table

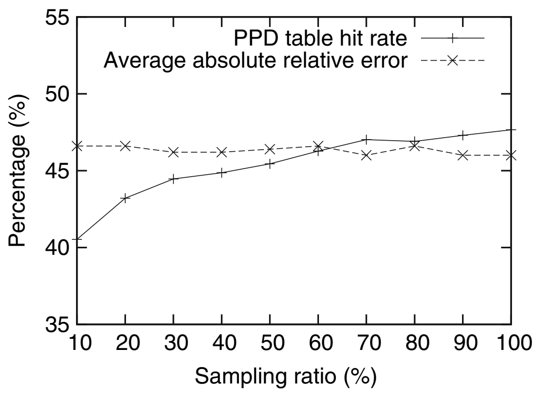


Fig. 13 Effect of sampling ratio on the hit rate of the PPD table

vectors was small. The number of entries for ALL_RAND was the most (125,400), since it constructed string triplets for *all* the clusters for each query string. CLOSE_RAND and CLOSE_LEX had a similar number of entries (about 30,000), since they only constructed samples for the 10 closest clusters for each query string. There was no big difference in the estimation errors for the heuristics. In the remaining experiments we used CLOSE_RAND.

Number of workload predicates: when populating the PPD table, we generated a workload of predicates, the strings of which were sampled from the data set. This number affects the quality of the PPD table. A very small number of samples will not be able to generate an accurate PPD table. However, the cost of sampling more string triplets also becomes larger. We generated 1,000 clusters using k-Medoids* algorithm. We used CLOSE_RAND to sample 0.01 to 40% of the strings as the strings in a workload of predicates. The results are in Fig. 12.

Figure 12a shows the sampling time for different sampling ratios. For instance, it took 14 min to sample 20% of the data set as query strings to generate triplets to populate the PPD table. For a 40% sample, the time was about 28 min. The average absolute relative error did not change much.

We also evaluated how the sampling ratio affected the hit rate during a lookup in the PPD table. The results are shown in Fig. 13. As the sampling ratio increased, there were more entries in the PPD table, and the hit rate increased, alleviating the underestimation problem. On the other hand, the overall estimation accuracy did not increase significantly. The main reason is that the percentages in the PPD table tend to give us overestimations, which were compensated by the underestimations. As we had more samples, these overestimations tend to be more accurate. The combined effect of the overestimation and the underestimation did not change much as the sampling ratio increased.

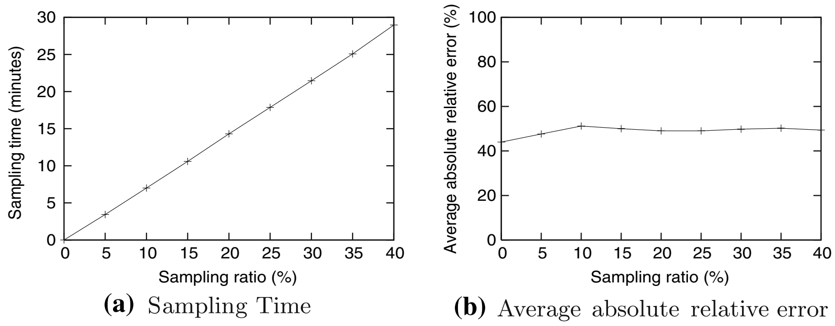
Number of closest clusters: we evaluated the number of closest clusters for each sample query string used by CLOSE_RAND. Figure 14 shows how this number affected the overall estimation accuracy. Increasing this number did help reduce the estimation error. The reduction was more significant for the data set of movie titles.

7.4 Effect of number of clusters

We evaluated the effect of the number of clusters. The more clusters we have, the closer the strings inside a cluster are to its pivot. As a result, the pivots can better represent the strings and the histograms are more accurate. On the other hand, more clusters require more time for online estimation.

We used the k-Medoids* algorithm to generate clusters, and CLOSE_RAND to populate the corresponding PPD table. We applied the error-correction step. We let the number of clusters vary from 500 to 2,000. Figure 15 shows the

Fig. 12 Effect of number of predicates in populating PPD table



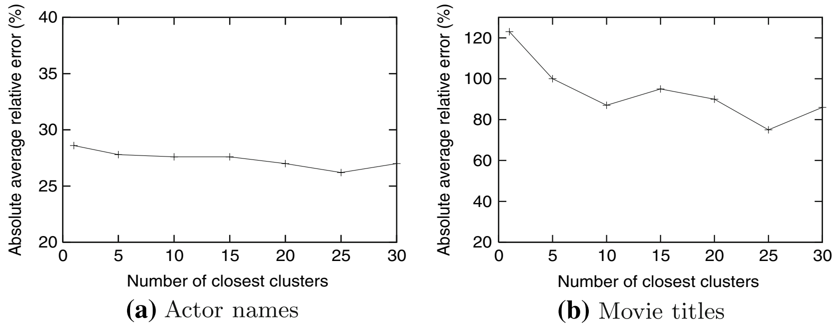


Fig. 14 Effect of the number of closest clusters in CLOSE_RAND sampling

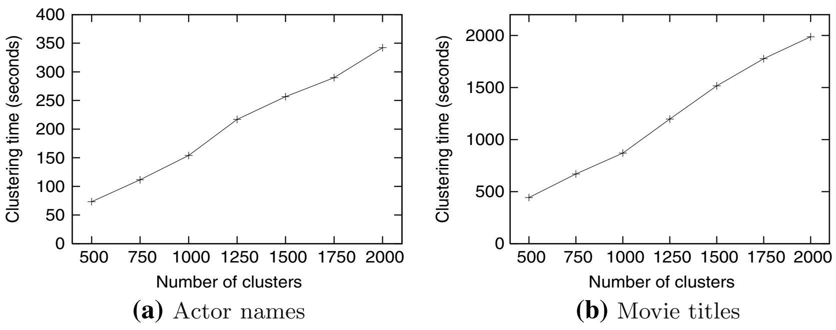


Fig. 15 Clustering time

clustering time for the two data sets. The clustering time increased linearly with the number of clusters. The clustering time for the data set of movie titles was larger due to its larger average string length. Figure 16 shows the average size (in number of entries) of the frequency tables. As the number of clusters increased, the size of the frequency tables decreased.

Figure 17 shows how the number of clusters affected the percentage of clusters pruned using the triangle inequality during an estimation. As we can see, as the number of clusters increased, more clusters can be pruned. For the data set of actor names, when there were 1,000 clusters, about 35% of the clusters were pruned. This result shows that if we want to further reduce the estimation time of scanning the clusters, we could adopt some indexing structure such as M-tree [8].

Figure 18 shows how the number of clusters affected the total amount of estimation time. It also shows how much estimation time was spent on scanning the entries in the frequency tables. The difference between the two time values is the time spent on scanning the clusters. These results show that about half of the estimation time was spent on scanning the clusters. Therefore, adopting indexing structures such as M-tree to reduce the number of scanned clusters may achieve a slight improvement on the overall estimation time.

Figure 19 shows the average absolute relative error for different numbers of clusters on both data sets. As expected, this error decreased as the number of clusters increased.

7.5 Size of data set

We created data sets of different sizes by randomly generating subsets of the records in the original data set. We used the same number of clusters 1,000, for these subsets. Figure 20a shows the average absolute relative errors for different data sizes, for different edit distance thresholds. The relative error stayed stable for those subsets with different sizes. For instance, when the threshold was 4, the average absolute relative error of a subset with 10,000 records was 21%, while it was 22% for a subset with 100,000 records.

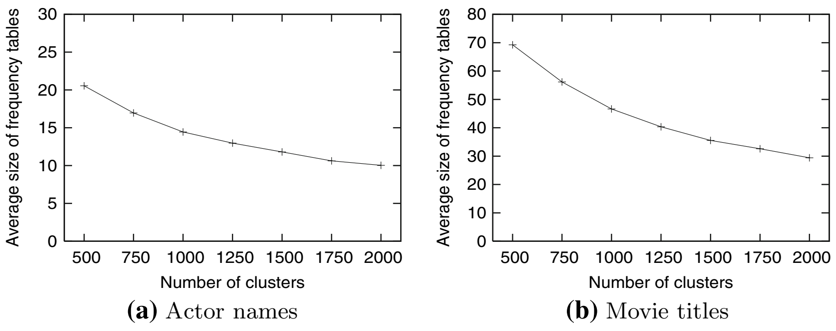


Fig. 16 Average number of entries in frequency tables

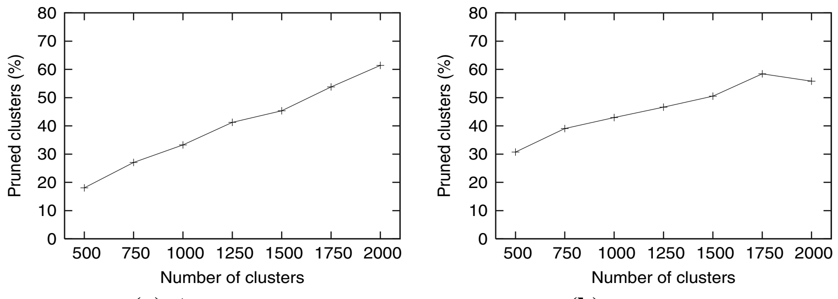


Fig. 17 Pruned clusters during estimation
(a) Actor names **(b)** Movie titles

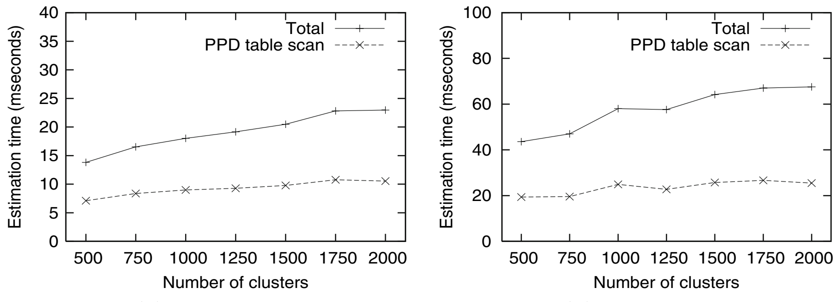


Fig. 18 Estimation time
(a) Actor names **(b)** Movie titles

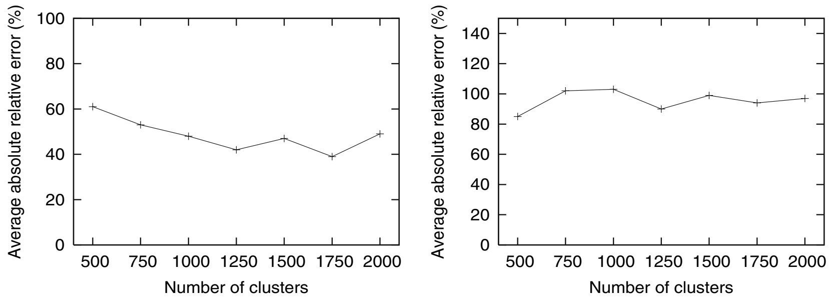


Fig. 19 Average absolute relative error
(a) Actor names **(b)** Movie titles

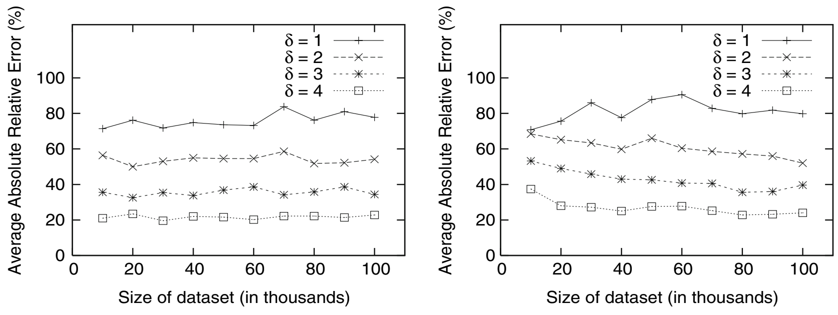


Fig. 20 Relative error
(a) Relative error with 1,000 clusters **(b)** Relative error versus data set size (number of clusters was 1% of data set size)

If we want to reduce the average estimation error as the data size increases, we need to use more clusters. We did a new set of experiments, in which for different subsets of the records, the number of clusters was 1% of the number of strings in each subset. Figure 20b shows the results for different subsets. As the data set became larger, the error decreased from 53% to 20,000 records ($\delta = 3$) to 39% for 100,000 records.

7.6 PPD table size

We used the *k*-Medoids* clustering algorithm to generate 1,000 clusters, sampled 20% of the data set and used the `CLOSE_RAND` heuristic to populate the PPD table. Figure 21 shows the size of the PPD table (in KB) for the three methods discussed in Sect. 5.3. Method 1 reduced the space by 63.6%. Method 2 reduced the space by 69.7%. Method 3 reduced the

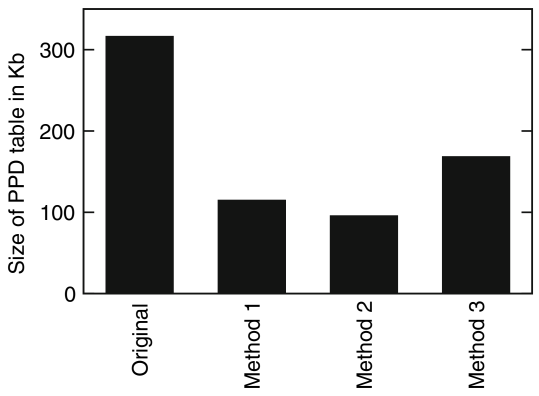


Fig. 21 PPD table size

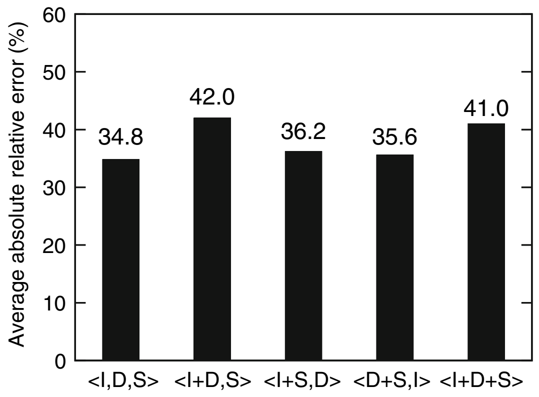


Fig. 22 Comparison of different proximity metrics

space by 46.7%. For Method 3, the additional time during the online computation of the percentage was less than 0.01ms.

7.7 Different proximity metrics

We evaluated the accuracy of different proximity variants of edit vector. We used k-Medoids* to generate 1,000 clusters, the `CLOSE_RAND` heuristic for populating the PPD table, sampled 20% of the data set, and applied the error correction step. Figure 22 shows the average absolute error for different proximity metrics. The best accuracy is obtained by the $\langle I, D, S\rangle$ metric, followed by the $\langle I + S, D\rangle$ and $\langle D + S, I\rangle$ metrics.

7.8 Effectiveness of error correction

After an initial estimation using the PPD table, we applied the error-correction step to further improve the estimation accuracy. We used the k-Medoids* algorithm to generate 1,000 clusters. We use the `CLOSE_RAND` heuristic to sample 20% of the data set to populate its PPD table. We learned a decision tree as described in Sect. 5.4. We ran the estimation for the testing query load with and without the error correction. The difference of their running times was negligible (less than 1 ms). We mainly evaluated the effect of the error-correction step on the relative estimation error.

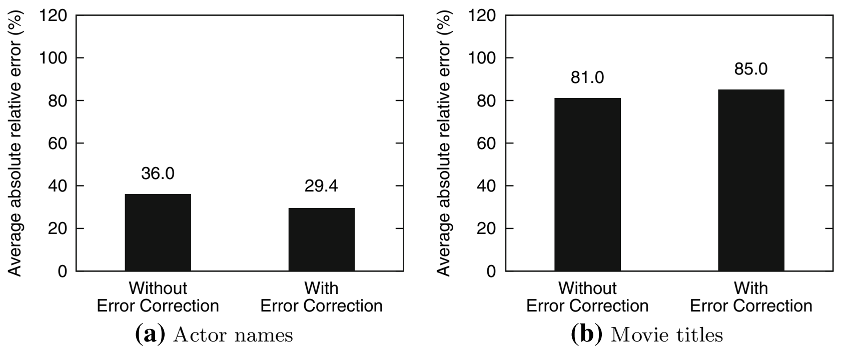
Figure 23 shows that for the data set of actor names, the error decreased after the error-reduction step. For example, the average absolute relative error was 36.0% without the error correction, and it reduced to 29.4% after the error-correction step. These results show that the error-correction step is effective in improving the quality of our selectivity estimation. However, for the data set of movie titles, the error did not improve. This result shows that whether the step should be used depends on the data set.

To see the details of estimation errors, we also calculated the quartile distribution of the relative errors. The quartile distribution bucketizes the relative errors into the following buckets: $[-100\%, -75\%)$, $[-75\%, -50\%)$, $[-50\%, -25\%)$, $[-25\%, 0\%)$, $[0\%, 25\%)$, $[25\%, 50\%)$, $[50\%, 75\%)$, $[75\%, 100\%)$, $[100\%, \infty)$. Estimates that are negative indicate underestimation, and positives mean overestimation. Figure 24 shows the quartile distributions of the relative errors for our data sets. The results show that estimation using our SEPIA technique was very accurate for relatively short strings, while the error increased for longer strings.

7.9 Jaccard coefficient distance

We also evaluated the applicability of our approach to other distance metrics by using Jaccard coefficient distance as an example. First, we repeated the same set of experiments and

Fig. 23 Effectiveness of error correction models **a** Actor names. **b** Movie titles



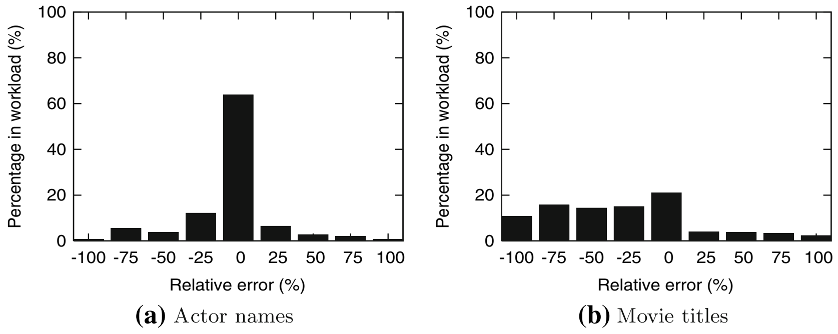


Fig. 24 Quartile distribution of relative errors

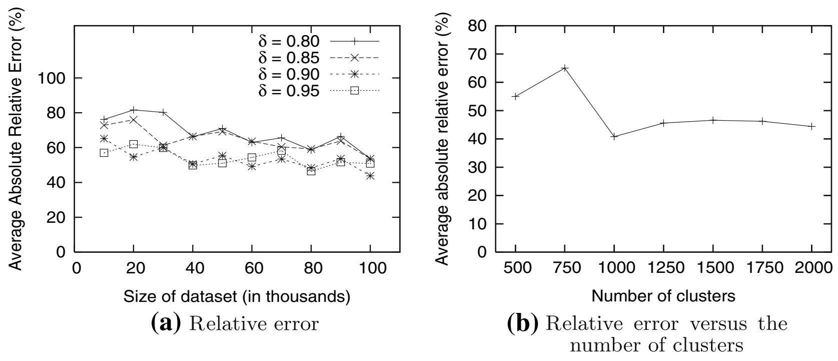


Fig. 25 Jaccard coefficient distance

replaced the edit distance function with the Jaccard coefficient function. Figure 25a shows that the estimation accuracy using SEPIA was also very high for this new similarity function.

We also repeated the experiments in Sect. 7.4 using the new function to see how the average absolute relative error changes with different numbers of clusters. Figure 25b shows a similar trend as Fig. 19c.

Acknowledgements Supported by NSF CAREER Award No. IIS-0238586.

8 Conclusions

We proposed a novel technique, called SEPIA, to support accurate selectivity estimation of fuzzy string predicates. It groups strings into clusters, and builds a histogram for the strings in each cluster. It also constructs a global histogram to keep distributions of similarity values based on string proximities. The histograms can be efficiently constructed and maintained. Our extensive experiments showed that SEPIA can support accurate estimation efficiently.

References

1. Aboulnaga, A., Alameldeen, A.R., Naughton, J.F.: Estimating the selectivity of XML path expressions for internet scale applications. VLDB (2001)
2. Anathakrishna, R., Chaudhuri, S., Ganti, V.: Eliminating fuzzy duplicates in data warehouses. VLDB (2002)
3. Bustos, B., Navarro, G., Ch'avez, E.: Pivot selection techniques for proximity searching in metric spaces. In: Proceedings of the XXI Conference of the Chilean Computer Science Society (SCCC'01) (2001)
4. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: SIGMOD (2003)
5. Chaudhuri, S., Ganti, V., Gravano, L.: Selectivity estimation for string predicates: overcoming the underestimation problem. In: International Conference on Data Engineering (2004)
6. Chávez, E., Navarro, G., Baeza-Yates, R.A., Marroquín, J.L: Searching in metric spaces. ACM Comput. Surv. **33**(3), 273–321 (2001)
7. Chen, Z., Korn, F., Koudas, N., Muthukrishnan, S.: Selectivity estimation for boolean queries. In: Symposium on Principles of Database Systems, pp. 216–225 (2000)
8. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. VLDB (1997)
9. Cohen, W., Ravikumar, P., Fienberg, S.: A Comparison of String Metrics for Matching Names and Records. Data Cleaning Workshop in Conjunction with KDD (2003)
10. Filho, R.F.S., Traina, A.J.M., Jr., C.T., Faloutsos, C.: Similarity search without tears: The OMNI family of all-purpose access methods. ICDE 623–630 (2001)
11. Gower, J.C., Legendre, P.: Metric and euclidean properties of dissimilarity coefficients. J. Class. **3**(1), 5–48 (1986)
12. Gravano, L., Ipeirotis, P., Koudas, N., Srivastava, D.: Approximate text joins and their integration into an RDBMS.In: Proceedings of WWW (2002)
13. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. VLDB 491–500 (2001)
14. Hartigan, J.A., Wong, M.A.: A k-means clustering algorithm. Appl. Stat. 100–108 (1979)

15. Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: Carey, M.J., Schneider D.A. (eds.) SIGMOD, pp. 127–138 (1995)
16. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Trans. Database Syst. **28**(4), 517–580 (2003)
17. Jagadish, H.V., Kapitskaia, O., Ng, R.T., Srivastava, D.: Multi-dimensional substring selectivity estimation. VLDB (1999)
18. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K.C., Suel, T.: Optimal histograms with quality guarantees. VLDB 275–286 (1998)
19. Jin, L., Koudas, N., Li, C.: NNH: improving performance of nearest-neighbor searches using histograms. In: EDBT (2004)
20. Jin, L., Koudas, N., Li, C., Tung, A.K.: Indexing mixed types for approximate retrieval. In: Proceedings of VLDB (2005)
21. Jin, L., Li, C.: Selectivity estimation for fuzzy string predicates in large data sets. In: VLDB 397–408 (2005)
22. Jin, L., Li, C., Mehrotra, S.: Efficient record linkage in large data sets. In: Eighth International Conference on Database Systems for Advanced Applications (2003)
23. Traina, C. Jr., Traina, A.J., Faloutsos, C.: Distance exponent: A new concept for selectivity estimation in metric trees. In: ICDE (2000)
24. Kaufman, L., Rousseeuw, P.J.: Finding Groups in Data: An Introduction to Cluster Analysis. Wiley, New York (1990)
25. Kooi, R.P.: The optimization of queries in relational databases. Ph.D thesis, Case Western Reserve University (1980)
26. Krishnan, P., Vitter, J.S., Iyer, B.R.: Estimating alphanumeric selectivity in the presence of wildcards. In: SIGMOD, pp. 282–293. ACM Press, New York (1996)
27. Lee, M.L., Ling, T.W., Low, W.L.: Intelliclean: a knowledge-based intelligent data cleaner. In: Knowledge Discovery and Data Mining, pp. 290–294 (2000)
28. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Dokl. **10**(8), 707–710 (1966). (Original in Russian in Doklady Akademii Nauk SSSR 163, 4, 845–848, 1965)
29. Lim, L., Wang, M., Padmanabhan, S., Vitter, J., Parr, R.: Xpathlearner: an on-line selftuning markov histogram for xml path selectivity estimation. In: 28th International Conference on Very Large Data Bases (2002)
30. Liu, Z., Luo, C., Cho, J., Chu, W.: A probabilistic approach to metasearching with adaptive probing. ICDE (2004)
31. Mattias, Y., Vitter, J.S., Wang, M.: Dynamic maintenance of wavelet-based histograms. In: Proceedings of the International Conference on Very Large Databases, (VLDB), pp. 101–111. Cairo, Egypt (2000)
32. Muralikrishna, M., DeWitt, D.J.: Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: SIGMOD (1988)
33. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
34. Poosala, V., Haas, P.J., Ioannidis, Y.E., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. SIGMOD 294–305 (1996)
35. Sahinalp, S.C., Tasan, M., Macker, J., Ozsoyoglu, Z.M.: Distance based indexing for string proximity search. In: International Conference on Data Engineering (2003)
36. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: Proceedings of VLDB (2002)
37. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J. Mol. Biol. **147**, 195–197 (2001)
38. Standard template library (stl) http://www.sgi.com/tech/stl/
39. Tao, Y., Faloutsos, C., Papadias, D.: The power-method: a comprehensive estimation technique for multi-dimensional queries. In: CIKM (2003)
40. Ukkonen, E.: Algorithms for approximate string matching. Inform. Control **64**(1–3), 100–118 (2001)
41. Vleugels, J., Veltkamp, R.C.: Efficient image retrieval through vantage objects. Visual Inform. Inform. Syst. 575–584 (1999)