Selectivity Estimation for Fuzzy String Predicates in Large Data Sets*

Liang Jin
University of California, Irvine, USA
liangj@ics.uci.edu

Chen Li
University of California, Irvine, USA
chenli@ics.uci.edu

Abstract

Many database applications have the emerging need to support fuzzy 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 a fuzzy 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 fuzzy string predicates. We develop a novel technique, called Sepia, to solve the problem. It groups strings into clusters, builds a histogram structure for each cluster, and constructs a global histogram for the database. 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 function. 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 fuzzy string predicates.

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

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

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 & 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 [12], cosine similarity [10], Jaccard coefficient distance [9], and variants thereof [7, 30]. 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, 11, 14, 19, 24].

There are recent studies on how to process such a fuzzy predicate efficiently in large databases (e.g.,[2, 4, 11, 18]. 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 $\geq$ 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 predicate $f(q, s) \leq \delta$.

Assume we adopt edit distance for the function f. Our goal thus becomes estimating how many strings in a database 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 group G of 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 strings in the database, 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 database 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.

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

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, and how to extend the technique to other similarity functions. Section 6 reports the results of our experiments. We conclude the work in Section 7.

1.1 Related Work

Many techniques have been developed to estimate selectivities of range conditions on single or multiple numeric attributes (e.g., [16, 22, 27, 28, 29]). 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 [29]. 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%'`. [15, 23] 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 approach guesses a set of shortest identifying substrings, and combines the selectivities of those substrings using a regression tree model. [1, 25] study how to estimate selectivities of XML path expressions. These techniques cannot be directly adopted to solve our selectivity problem for fuzzy string predicates.

Some string similarity functions, such as edit distance and Jaccard coefficient distance, are metrics. Traina et al. [20] show that many diverse metric datasets follow a "power law" distribution. That is, for a metric-space dataset, 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 [32] for multi-dimensional data sets.) They propose a technique to estimate the D value by building an optimal M-tree for the dataset. 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 [20].

There have been studies on efficiently answering queries with fuzzy string predicates, especially in the context of data cleansing [24, 31]. Gravano et al. [11] present a technique to do similar string joins inside a relational database system. Jin et al. [19] 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. [18] 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 fuzzy string predicates. We focus on selectivity estimation using edit distance. Section 5.2 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 [34].

Let $\mathcal{B}$ be a bag with $|\mathcal{B}|$ strings. Examples include all names in an employee table, or all telephone numbers in such a table. A fuzzy string predicate P is in the format of $P(q, \delta)$, where q is a query string, and δ is a threshold, i.e., the maximal edit distance. 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 a fuzzy string predicate $P(q, \delta)$ 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 strings in the database into clusters, and construct histogram structures to support estimation. Figure 1 illustrates the intuition behind our technique. Given a fuzzy predicate $P(q, \delta)$, 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 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.

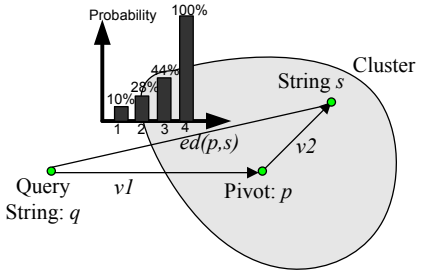


Figure 1: Intuition of SEPIA.

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, Figure 2(a) 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 $P($`lukas`$, 3)$. 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 same edit distance to the pivot string p.

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 3.1 (**Edit Vector**) Let s_1 and s_2 be two strings. An *edit vector from s_1 to s_2* is a three-number

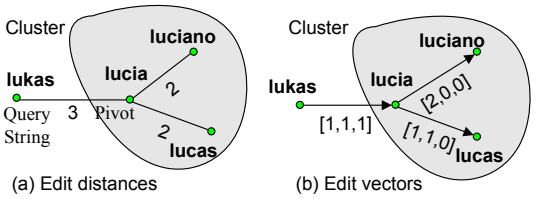


Figure 2: Using edit vectors to better describe string proximity than edit distances.

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 2(b) shows the edit vectors for 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 3 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 [34]. 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 in the database 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. We assume that the pivot is *not* part of the data set for the purpose of easy dynamic maintenance, as discussed in Section 4. (The idea of selecting a pivot for a cluster is also adopted in [3, 8, 17, 33].) 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.

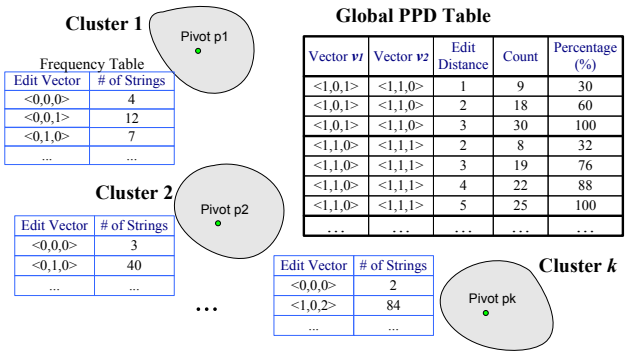


Figure 3: Histograms used in SEPIA.

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 database. As illustrated in Figure 1, the goal of having this histogram is to store information about the edit-distance distribution given a 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 $e' \leq e$) in the construction of this table (discussed in Section 4.2). We keep this count in order to support incremental maintenance of this table.

For instance, consider the PPD table in Figure 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 $\langle 1, 1, 0\rangle$ from the pivot, 30%, 60%, 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 9, 18, and

30 triplets (v_1, v_2, e'), 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

Figure 4 shows the pseudo code of our estimation algorithm. To estimate the frequency of a fuzzy string predicate $P(q, \delta)$, 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 $|v_1|+|v_2| \leq \delta$, 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.

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: A fuzzy string predicate $P(q, \delta)$, where
- q is a query string, and
- δ is an edit-distance threshold.

Output: Estimated # of strings satisfying the predicate.
Method:

```
est ← 0;
for (i = 1 to k) {
    Compute an edit vector v1 from q to pi;
    if (|v1| > ri + δ) continue; // ignore this cluster
    for (each entry (v2, n) in FTi) {
        if (|v1| + |v2| ≤ δ)) { est ← est + n; continue;}
        if (||v1| − |v2|| > δ)) continue;
        Use (v1, v2, δ) to find a percentage f in PPD;
        if (f is not found) continue; // no such entry
        est ← est + f × n;
    }
}
return (est);
```

Figure 4: 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 [13]) 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, causing their total size 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 [29] 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 [21] 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 dataset 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, swaps the pair, and re-distributes 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 [21]. We sample the database several times (e.g., 5 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.

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 Figure 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 (Section 6).

- `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 database is updated, we need to consider the effect of insertions and deletions on the table. In Section 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 database. 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 $ed(q, s_{new})$. 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 $e \geq ed(q, s_{new})$. 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 database, we may need to reconstruct the histogram structures in order to support accurate estimations.

5 Improving Estimation Accuracy and Extensions to Other Functions

In this section we discuss how to further improve the accuracy of estimations using SEPIA, and how to extend the technique to other similarity functions.

5.1 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 miss hits of the PPD table, i.e., a proximity pair during an estimation does not exist in the PPD table. The second one is that the percentage entries in the PPD table may not be accurate.

We 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 [26].)

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 $(f_{est} - f_{real})/f_{real}$, where f_{est} is the estimated frequency, and f_{real} is the real frequency. Figure 5 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.

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%

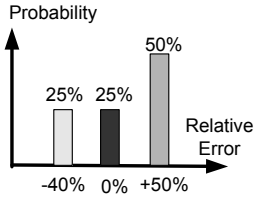


Figure 5: Relative errors of predicates.

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.

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 6 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%.

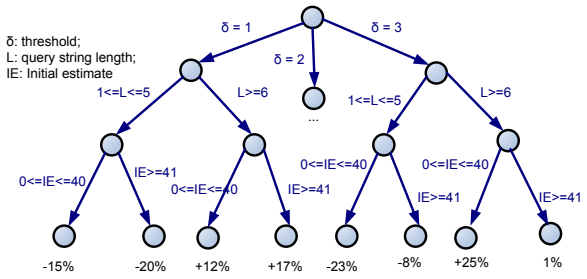


Figure 6: Decision tree to compute average relative error of initial estimation.

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 $(f_{est} - f_{real})/f_{real}$ of relative error. In our experiments we will show that this step can effectively improve the accuracy of estimations.

5.2 Extensions to Other Similarity 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 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, Figure 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* [7] 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.

6 Experiments

This section presents our extensive experimental results to evaluate the proposed SEPIA approach. We used two main data sources. (1) *Data set 1* consisted of 71,000 author names collected from the Citeseer Digital Library.[2] The length of each name varied from 2 to 20, and the average length was around 12. (2) *Data set 2* was a set of movie records available at the UCI KDD repository.[3] We extracted 11,423 unique movie titles, whose lengths varied between 3 and 80, with a mean length about 35. These two datasets had few duplicates originally. In order to evaluate SEPIA for predicates with various selectivities, we introduced duplicates into each dataset. We randomly selected 10% of the records from each dataset. For each of them, we introduced a number of duplicates that followed a uniform distribution between 1 and 20. Therefore, on the average each record in the 10% of the records was duplicated 10 times. After this step, Data set 1 contained 142,000 author names and Data set 2 had 22,846 movie titles.

We evaluated the accuracy of SEPIA using a workload of query predicates. It had two types of predicates: one was a predicate whose string was from the dataset, the other was a predicate whose string was not in the dataset. SEPIA gave similar estimation accuracies for both types of predicates. The edit distance threshold of each predicate was a random integer between 1 and 4 for Data set 1, and between 6 and 10 for Data set 2. To evaluate the accuracy of an estimation for a fuzzy predicate, we used its *relative error*, defined as $(f_{est} - f_{real})/f_{real}$, where f_{est} is the estimated frequency, f_{real} is the real frequency of this predicate. Correspondingly, we define its *absolute relative error* as $|f_{est} - f_{real}|/f_{real}$. Compared to related studies [5, 6, 25] 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 Visual C++. All the experiments were run on a PC, with a 2.4GHz Pentium-4 CPU and 1024MB memory. The operating system was Windows XP. The experimental results were very consistent between the two data sets, even though they have different sizes and query-length distributions, and we were using different thresholds. Due to space limitation, we mainly report the results on Data set 1 with the edit distance metric. Section 6.6 will report some results of Jaccard coefficient distance. The size of the SEPIA histogram structures was small. For instance, when we had 1,000 clusters for Data set

[2] http://citeseer.ist.psu.edu/
[3] http://kdd.ics.uci.edu/databases/movies/movies.html

1, the size of the PPD table was about 5 MB, and the total size of the frequency tables was about 200 kB.

6.1 Clustering Algorithms

We used the lexicographic-based clustering algorithm and the k-Medoids clustering algorithm, as discussed in Section 4.1, to cluster strings. We used Data set 1 with 142,000 strings, and fixed the number of clusters to 1,000. After running each clustering algorithm, we applied the `CLOSE_RAND` heuristic to populate the PPD table. Then 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 Section 5.1.

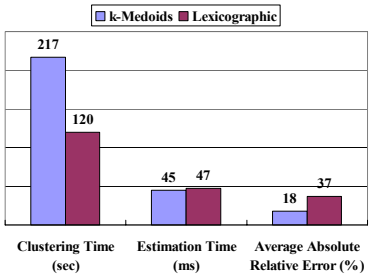


Figure 7: Comparison of two clustering algorithms.

Figure 7 shows the results of the two clustering algorithms. The k-Medoids algorithm took more time (217 seconds) to finish the clustering step than the lexicographic-based algorithm (120 seconds). Their estimation times were similar: $45ms$ for k-Medoids and $47ms$ for the other. For the average absolute relative error, k-Medoids (18%) was better than the other algorithm (37%). Thus we chose the k-Medoids algorithm to cluster strings in all other experiments.

6.2 Populating PPD Table

Heuristics: 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 Section 4.2. We first generated 1,000 clusters for Data set 1 using k-Medoids. For each heuristic, we sampled 5% of the strings as strings in a workload of fuzzy predicates. We collected the average absolute relative error for each heuristic. Figure 8 shows the details of the sampling time and error comparisons.

The running time for `ALL_RAND` was the most (54 minutes), since it constructed string triplets for *all* the clusters for each query string. `CLOSE_RAND` ran much faster (20 minutes) since it only constructed samples for the 10 closest clusters for each query string. `CLOSE_LEX` was slightly slower than `CLOSE_RAND` because `CLOSE_LEX` had an additional sorting step. On the other hand, `CLOSE_LEX` had a slightly smaller estimation error (17% versus 18%). `CLOSE_UNIQUE` took

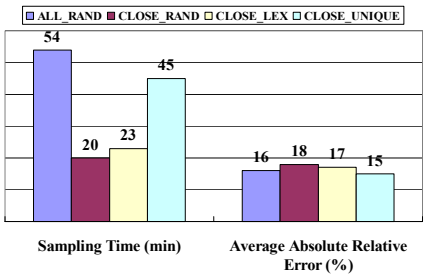
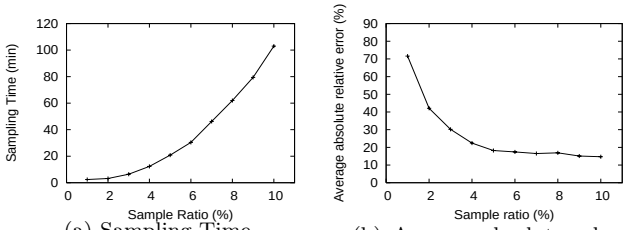


Figure 8: Different heuristics for populating the PPD table.

45 minutes, mainly because it needed to do additional condition checking. There was no big difference in the estimation errors for the heuristics. In the remaining experiments we chose `CLOSE_RAND` due to its low running time.

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. The more samples we get, the more accurate the PPD table becomes. However, the cost of sampling more string triplets also becomes larger. We used Data set 1, and generated 1,000 clusters using k-Medoids algorithm. We picked the `CLOSE_RAND` heuristic to perform the sampling, and sampled 1% to 10% of the strings as the strings in a workload of predicates. The results are in Figure 9.



(a) Sampling Time

(b) Average absolute relative error

Figure 9: Effect of number of predicates in populating PPD table.

Figure 9(a) shows the sampling time for different sampling ratios. For instance, it took 20 minutes to sample 5% of the dataset as query strings to generate triplets to populate the PPD table. For a 10% sample, the time was about 100 minutes. The average absolute relative error enjoyed a big reduction when the sampling ratio increased from 1% to 4%, then decreased slowly as we sampled more strings. Considering the sampling time and the relative error, we chose a sample ratio around $4\% - 5\%$ in other experiments.

6.3 Effectiveness of Error Correction Step

After an initial estimation using the PPD table, we applied the error-correction step to further improve the estimation accuracy. We used both Data set 1 and Data set 2. We used the k-Medoids algorithm to generate $1,000$ and 200 clusters for Data set 1 and Data set 2, respectively. We use the `CLOSE_RAND` heuristic to sample 5% of each dataset to populate its PPD table. We learned the decision tree as described in Section 5.1. We ran the estimation for the testing query load with and without the error correction. The difference of their running times was negligible. For instance, without error correction, the average estimation time for a query predicate for Data set 1 was around 45 ms. With error correction, it became 49 ms. Therefore, we mainly evaluated the effect of the error-correction step on the relative estimation error.

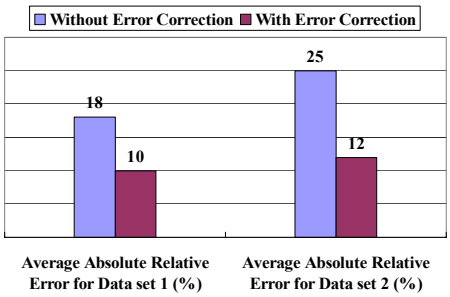
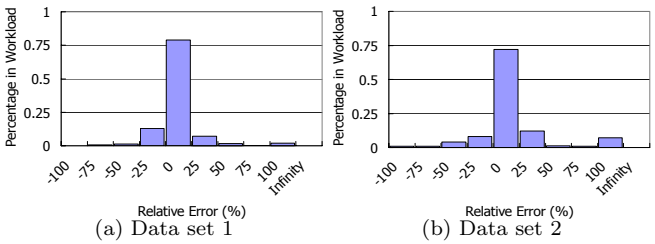


Figure 10: Effectiveness of error correction models.

Figure 10 shows a big reduction on the error after the error-reduction step. For example, the average absolute relative error for Data set 1 was 18% without the error correction, and it reduced to 10% after the error-correction step. The step achieved a 15% reduction for Data set 2 as well (from 25% to 12%). These results show that the error-correction step is very effective in improving the quality of our selectivity estimation.



(a) Data set 1 (b) Data set 2

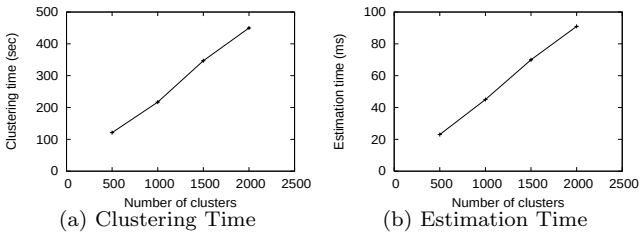
Figure 11: Quartile distribution of relative errors.

To see the details of estimation errors, we also calculated the quartile distribution of the relative errors for both datasets. 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 11 shows the quartile distributions of the relative errors for both datasets. The results show that estimation using our SEPIA technique was very accurate.

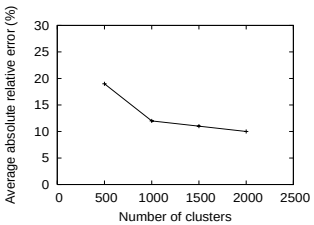
6.4 Effect of Number of Clusters

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, because the cost of the estimation is mainly due to scanning all the pivots and calculating the edit vectors.

We did experiments on Data set 1. 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 12 shows the clustering time, the estimation time, and the average absolute relative error for different numbers of clusters. The clustering time and the estimation time increased linearly with the number of clusters. The average absolute relative error decreased as the number of clusters increased, and the reduction became smaller after more than $1,000$ clusters were used. When we used $1,000$ clusters, the estimation time was only about 42 ms.



(a) Clustering Time (b) Estimation Time



(c) Average Absolute Relative Error

Figure 12: Effect of Number of Clusters.

We then chose subsets of the records in Data set 1 with different numbers of records. We used the same number of clusters, $1,000$, for these subsets. Figure 13 shows the average absolute relative errors for different data sizes. As the data set became larger, the error

increased from 4% for 20,000 records ($\delta = 4$) to 14% for 142,000 records, which is still very low.

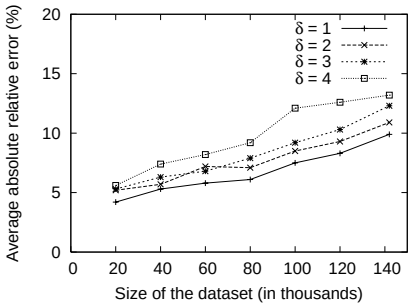


Figure 13: Relative error with $1,000$ clusters.

If we want to keep the average estimation error as the data size increases, we need to use more clusters. In the next set of experiments, for different subsets of the records, the number of clusters was 1% of the number of strings in each subset. Figure 14 shows the results for different subsets. The relative error stayed stable for those subsets with different sizes. For instance, when the threshold of 2, the average absolute relative error of a subset with $2,000$ records was 9.2%, while it was 9.9% a subset with $120,000$ records.

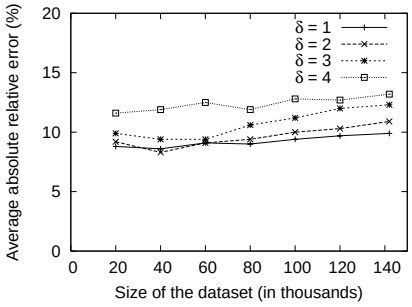
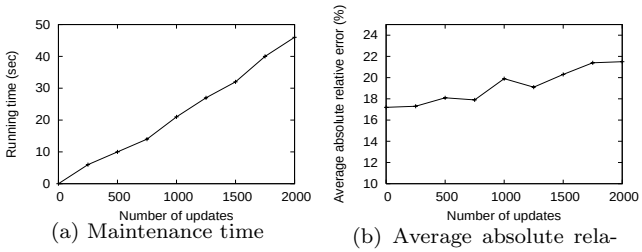


Figure 14: Relative error versus database size (Data set 1).

6.5 Dynamic Maintenance

We performed dynamic maintenance on Data set 1. We randomly generated a sequence of updates. For each update, we randomly selected a string in the dataset, and introduced a random number (between 1 and 5) of edit changes to the string. Figures 15(a) and (b) show the maintenance time and the average relative errors after different numbers of updates. On the average, an update took about 18 ms to finish, and the average absolute relative error after many updates did not deteriorate much. For instance, before any updates, the relative error was 17%. After $2,000$ updates, it remained close to 21%. These results show that the histograms in Sepia can be maintained efficiently, and the estimation accuracy does not change much after many updates.



(a) Maintenance time

(b) Average absolute relative error

Figure 15: Dynamic maintenance.

6.6 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 for Data set 1, and replaced the edit distance function with the Jaccard coefficient function. Figure 16 shows that estimation accuracy using Sepia was also very high for this new similarity function.

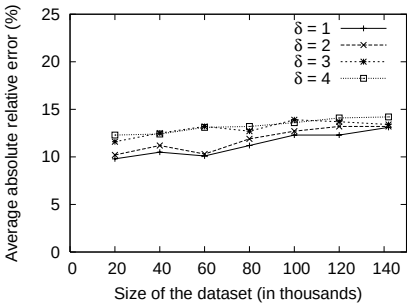


Figure 16: Relative error for Data set 1 using Jaccard coefficient distance.

We also repeated the experiments in Section 6.4 using the new function to see how the average absolute relative error changes with different numbers of clusters. Figure 17 shows a similar trend as Figure 12(c).

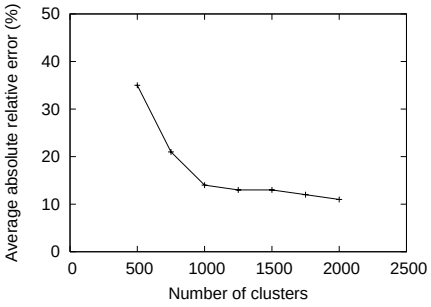


Figure 17: Relative error versus the number of clusters using Jaccard coefficient distance.

7 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 our SEPIA technique can support accurate estimation efficiently.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *The VLDB Journal*, pages 591–600, 2001.

[2] R. Anathakrishna, S. Chaudhuri, and V. Ganti. Eliminating Fuzzy Duplicates in Data Warehouses. *VLDB*, Aug. 2002.

[3] B. Bustos, G. Navarro, and E. Ch'avez. Pivot selection techniques for proximity searching in metric spaces. In *Proc. of the XXI Conference of the Chilean Computer Science Society (SCCC'01)*, 2001.

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[5] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *International Conference on Data Engineering*, 2004.

[6] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Symposium on Principles of Database Systems*, pages 216–225, 2000.

[7] W. Cohen, P. Ravikumar, and S. Fienberg. A Comparison of String Metrics for Matching Names and Records. *Data Cleaning Workshop in Conjunction with KDD*, Aug. 2003.

[8] R. F. S. Filho, A. J. M. Traina, C. T. Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *ICDE*, pages 623–630, 2001.

[9] J. C. Gower and P. Legendre. Metric and euclidean properties of dissimilarity coefficients. *Journal of Classification*, 3:5–48, 1986.

[10] L. Gravano, P. Ipeirotis, N. Koudas, and D. Srivastava. Approximate Text Joins and Their Integration into an RDBMS. *Proceedings of WWW*, 2002.

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

[12] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1998.

[13] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. In *Applied Statistics*, pages 100–108, 1979.

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

[15] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-dimensional substring selectivity estimation. In *The VLDB Journal*, pages 387–398, 1999.

[16] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. *VLDB*, pages 275–286, Aug. 1998.

[17] L. Jin, N. Koudas, and C. Li. NNH: Improving performance of nearest-neighbor searches using histograms. In *EDBT*, 2004.

[18] L. Jin, N. Koudas, C. Li, and A. K. Tung. Indexing mixed types for approximate retrieval. In *Proc. of VLDB*, 2005.

[19] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Eighth International Conference on Database Systems for Advanced Applications*, 2003.

[20] C. T. Jr., A. J. Traina, and C. Faloutsos. Distance exponent: A new concept for selectivity estimation in metric trees. In *ICDE*, 2000.

[21] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[22] R. P. Kooi. The Optimization of Queries in Relational Databases. *PhD Thesis, Case Western Reserve University*, Sept. 1980.

[23] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD*, pages 282–293. ACM Press, 1996.

[24] M.-L. Lee, T. W. Ling, and W. L. Low. Intelliclean: a knowledge-based intelligent data cleaner. In *Knowledge Discovery and Data Mining*, pages 290–294, 2000.

[25] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. Xpathlearner: an on-line selftuning markov histogram for xml path selectivity estimation. In *28th International Conference on Very Large Data Bases*, 2002.

[26] Z. Liu, C. Luo, J. Cho, and W. Chu. A probabilistic approach to metasearching with adaptive probing. In *ICDE*, 2004.

[27] Y. Mattias, J. S. Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. *Proceedings of the International Conference on Very Large Databases, (VLDB), Cairo, Egypt*, pages 101–111, Sept. 2000.

[28] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *SIGMOD*, 1988.

[29] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.

[30] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Ozsoyoglu. Distance based indexing for string proximity search. In *International Conference on Data Engineering*, 2003.

[31] S. Sarawagi and A. Bhamidipaty. Interactive Deduplication Using Active Learning. *Proceedings of VLDB*, 2002.

[32] Y. Tao, C. Faloutsos, and D. Papadias. The power-method: A comprehensive estimation technique for multi-dimensional queries. In *CIKM*, 2003.

[33] J. Vleugels and R. C. Veltkamp. Efficient image retrieval through vantage objects. In *Visual Information and Information Systems*, pages 575–584, 1999.

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