Selectivity Estimation in the Presence of Alphanumeric Correlations

Min Wang
Duke University
minw@cs.duke.edu

Jeffrey Scott Vitter
Duke University
jsv@cs.duke.edu

Bala Iyer
IBM Santa Teresa Lab
balaiyer@vnet.ibm.com

Abstract

Query optimization is an integral part of relational database management systems. One important task in query optimization is selectivity estimation, that is, given a query P, we need to estimate the fraction of records in the database that satisfy P.

Almost all previous work dealt with the estimation of numeric selectivity, i.e., the query contains only numeric variables. The general problem of estimating alphanumeric selectivity is much more difficult and has attracted attention only very recently, and the focus has been on the special case when only one column is involved.

In this paper, we consider the more general case when there are two correlated alphanumeric columns. We develop efficient algorithms to build storage structures that can fit in a database catalog. Results from our extensive experiments to test our algorithms, on the basis of error analysis and space requirements, are given to guide DBMS implementors.

1 Introduction

Query optimization, conducted by a query optimizer, is an integral part of relational database management systems. One important task in query optimization is selectivity estimation [13]. Given a predicate P specified by a query, a query optimizer first estimates its selectivity $sel(P)$ (also called *filter factor*) which is the fraction of records in the database that satisfy P. Based on such estimation, the optimizer will choose how to execute the query in an economical way [9]. The performance of a query optimizer, and thus the whole query processing, depends essentially on how accurate the selectivity estimation is [6, 9].

A lot of work has been done on selectivity estimation [6, 11]. Almost all previous work dealt with the estimation of numeric selectivity, i.e., the predicate P contains only numerical variables. The more general problem of estimating alphanumeric selectivity has attracted attention only very recently, due to the work of Krishnan, Vitter and Iyer [4, 5].

Since alphanumeric data are quite different from numeric data, many known techniques for estimating numeric selectivity are no longer suitable for the more general case. In [4, 5], Krishnan, Vitter and Iyer proposed to use suffix trees to estimate alphanumeric selectivity for the case when only one individual column is involved (we call this *one-column estimation*).

In practice, a query usually involves many columns. A survey of DB2 (relational database system from IBM) found 93 SQL SELECT queries on single table used an average of 4 predicates [12]. Without the outlier queries that have over 5 predicates, the average was 1.83 predicate per table. Therefore, it is necessary to develop techniques for estimating selectivity when more than one columns is involved (we call this *multi-column estimation*). Accurate two predicate selectivity estimation is beneficial and is addressed in this paper.

Multi-column estimation can be done very efficiently if the queries contain only numeric data. For example, the problem may be solved by the generalized Equi-Depth Histogram (EDH) method [10, 14]. In general, we can use any of the various multi-dimensional data structures [3, 8] to solve the multi-column estimation problem if only numeric data are involved. But those methods and data structures cannot be used to deal with alphanumeric data because no good "order" property can be used in the later case. That is, alphanumeric data are truly "discrete" data whereas numeric data are continuous.

Selectivity estimation usually involves two phases: an off-line phase and an on-line phase. In the off-line phase, usually referred to as *RUNSTATS*, information about a (large) table is collected into a catalog (actually a persistent data structure) of reasonable size. In the on-line phase, the catalog is consulted to estimate selectivity for any given query against the database. The practical problem is to store vast information in little space, and address all cases. Because this is impractical, only a partial solution is possible. Information loss is accepted and so are heuristic solutions.

In this paper we generalize the work of [4, 5]. We consider the problem of selectivity estimation for the case when two alphanumeric columns are involved. As in [4, 5], we

use suffix tree as our basic data structure. We introduce a *dynamic graph construction problem* to characterize the possible complex correlation between the two columns. We develop efficient algorithms to collect statistics about the two columns and their correlation and store it in a dynamically constructed graph of reasonable size that we propose be stored in the database catalog. The graph would be used for selectivity estimation according to various heuristic strategies. We have conducted experiment to validate our algorithms.

This paper is organized as follows. In Section 2, we describe the problem of selectivity estimation and review previous work. In Section 3, we introduce the dynamic graph construction problem and formulate the major part of our selectivity estimation problem accordingly. Section 4 is a general description of our method. In Section 5, we describe the off-line phase, i.e., the construction of a small version of a huge graph. In Section 6, we discuss many strategies which will be used to do actual estimation. In Section 7, we propose a method to build a different kind of catalog. We formulate the problem as a least square calculation and show that it can be solved very efficiently. In Section 8, we report our experimental results. Finally, in Section 9, we suggest some problems for future research.

2 The Problem and Previous Work

2.1 The Problem

Suppose we have a relational database which has a table $PART$. A record (a row) in $PART$ stores information about a particular part. A part has many characteristics, e.g., it may have a color and a shape. Each characteristic is represented by an attribute of $PART$. Suppose $PART$ has attributes COLOR and SHAPE. If we want to find out all parts which have color GREEN and shape CIRCLE, we may use the standard SQL predicate LIKE to issue the following query:

COLOR LIKE GREEN AND SHAPE LIKE CIRCLE

If we want to find out all parts whose colors are green-like and whose shapes are circle-like, we may use the following query:

COLOR LIKE $*$GREEN$*$ AND SHAPE LIKE $*$CIRCLE$*$

where "$*$" represents the wildcard, i.e., $*$ represents any (empty or nonempty) character string.

For a given query, the optimizer will first estimate how many rows in the database satisfy the predicate specified by the query. We can state this problem more formally as follows.

Let F be an M by N table, i.e., F is a relation with M rows and N columns. Each row represents one tuple of a table, and each column represents one field of a specific type. For example, a column might be of numeric type, which indicates that all data in this column must have a numeric value. In this paper, we are mainly interested in data of alphanumeric type and we assume that all columns are of alphanumeric type. We use $F(i,*)$, $F(*,j)$ and $F(i,j)$ to denote the ith row, jth column, and (i,j) element of F, respectively. Note that any element $F(i,j)$ is a character string.

Suppose we are given a query P, which is a set of patterns,

$$P = \{(i_1, p_{i_1}), (i_2, p_{i_2}), \ldots, (i_k, p_{i_k})\},$$

where each i_j ($1 \le i_j \le N$) indicates one column and each p_{i_j} is a pattern string. We define $F(P)$ as the set of rows which contain all the given patterns in the corresponding columns, i.e.,

$$F(P) = \{F(i,*) | F(i, i_j) \text{ contains } p_{i_j} \text{ as a substring}, 1 \le j \le k\}.$$

Then the selectivity $s(P)$ is defined as

$$s(P) = \frac{|F(P)|}{M}.$$

Our goal is to estimate $s(P)$ for any given query P, i.e., we want to estimate the number of rows which contain the given patterns.

In this paper, we only study the special case when $N = 2$, i.e., we only consider two columns.

2.2 Suffix Tree: The Krishnan-Vitter-Iyer Method

Krishnan, Vitter and Iyer have developed a method (henceforth referred to as the KVI method) to estimate alphanumeric selectivity for the one-column case. Their method is based on the key data structure of suffix tree. In the following, we briefly describe their method (for more details, see [4, 5].)

First, let's briefly review suffix trees. Let s be a string of length l. We will use s_i to denote the suffix of s starting at position i, $0 \le i \le l-1$ (we count the positions starting from position 0 on).

A suffix tree T for s is a rooted tree which has the following properties:

1. Every edge (u, v) has an associated substring of s as its label. Sometimes we consider that string as one associated with the child node v.
2. For any node v, the first symbols of the substrings associated with v's children are all different.
3. For any node v, let *path*(v) be the path from the root to v. The string represented by v, denoted by $p(v)$,

is the concatenation of the strings associated with the edges along the path *path*(v). We also say $p(v)$ corresponds to node v, and v is the *locus* of $p(v)$. Any suffix is represented by a unique leaf node.

4. Since any substring of s is the prefix of some suffix, for any substring σ of s, we can find a unique node v such that σ is the prefix of $p(v)$, and σ is not a prefix of the parent of v. We call v the extended locus of σ, and denote it by $v(\sigma)$.

For a suffix tree T and a string s, if we can find a path from the root to a node v whose corresponding string contains s as a prefix, i.e., we can find an extended locus of s, then we say we can match s in T successfully.

The above definition can be generalized to a set of strings in the obvious way. Moreover, for a set of strings S, when constructing its suffix tree T, for each node v which represents a substring $p(v)$, we can assign a count $c(v)$ to v which is the number of strings in S that contain $p(v)$ as a substring.

For more details about suffix trees and their generalization, see [1, 7].

Now let us turn to the KVI method. There are two phases. In an *off-line* phase (e.g., load, reorganization), a suffix tree T for F is built. Each node of T corresponds to some substring that occurs in some row in F, and vice versa. For a substring x, we denote its corresponding node in T by $v(x)$. The number of rows in F which contain x as a substring is denoted by $c(x)$ which is called the *true count* of x.

When T is constructed, pruning is done. For a given *prune count* p, if $c(x)$ is smaller than p, we informally call x a *light substring* , and $v(x)$ a *light node*. All light nodes will be pruned off. The remaining tree is the so-called *catalog* (strictly speaking, the data structure to be stored in the catalog) which is the output of the off-line phase. (We still use T to denote this final tree.)

In the *on-line phase* (e.g., query optimization), when a predicate "LIKE $*p*$" is given, by running the KVI method against it, an estimated selectivity for the pattern $*p*$ is obtained, that is, an estimation on the number of all rows which contain p as a substring is obtained.

Since T must be limited in its size, many "infrequent" substrings have to be pruned out. Thus it would happen quite often that a given pattern cannot be matched by any node in T. This mismatch does not necessarily imply that that pattern never occurs in F, and we still need to estimate the selectivity of these "unlikely" substrings. In the KVI method, several *matching strategies* have been suggested to deal with these mismatches.

The main advantage for using suffix trees is that it allows very efficient storage of all substrings. Actually the space complexity is linear in the total length of the represented strings. But still, we cannot store the whole suffix tree since the database catalog size is rather limited.

3 Problem Formulation

In this section, we introduce a problem called *Dynamic Graph Construction* . We then formulate the off-line phase of our two-column estimation problem as a dynamic graph construction problem where the target is a bipartite graph.

3.1 Dynamic Graph Construction

A (directed/undirected) graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of (directed/undirected) edges. A *bipartite graph* is an undirected graph in which V can be partitioned into two sets V_1 and V_2 such that any edge has its one end vertex in V_1 and the other end vertex in V_2. A *weighted graph* is a graph for which each edge has an associated *weight* given by a *weight function* $w : E \rightarrow N$, where N is the set of weights. In this paper we only consider non-negative integer weights, i.e., N is the set of non-negative integers. Also, we may assign a weight to each vertex and we still call the graph a weighted graph. For a weighted graph, let c_V and c_E be two given threshold values. Any vertex with weight $\geq c_V$ is called a *heavy vertex* and any edge with weight $\geq c_E$ is called a *heavy edge* . Others are called *light vertices* or *light edges* . The two values c_V and c_E are called *vertex threshold* and *edge threshold* respectively. For a graph G and the given threshold values c_V and c_E, we delete all edges which are not heavy. We also delete all vertices which are neither heavy nor incident to any heavy edge. The resulted graph is called the *heavy subgraph with respect to the specified threshold values* . For precise graph theoretic terminology, we refer to [2].

Let us consider three operations on a graph: insert/delete a vertex v, insert/delete an edge e, and change the weight of a vertex or an edge. When deleting a vertex v, all edges incident to v will also be deleted.

Suppose there is a weighted undirected graph $G = (V, E, w)$ which has a discrete time parameter t, i.e., at any time t, there is a graph $G_t = (V_t, E_t, w_t)$. At time 0, there is an initial graph G_0 which is usually empty. Suppose at time t, we are given G_{t-1} and a set of operations O_t. Then G_t is defined as the graph obtained by applying all operations in O_t (possibly in some specified order) to G_{t-1}.

Now we define the **Dynamic Graph Construction Problem** (DGCP for short). Let $c_V, c_E \in N$ be given threshold values. Let G_0 be a given initial graph. Let $\{O_t\}$ be an on-line sequence of operation sets. At time t, only $\{O_i | i \leq t\}$ are known. Let $n \in N$. The goal is to construct the heavy subgraph of G_n with respect to c_V and c_E.

The seemingly most natural method to solve DGCP is deleting all light edges and/or light vertices after the whole G_n has been constructed. As we will see, this is not preferable and even impossible for very large database applications because G_n will become huge and we simply don't have enough space to store G_n. Instead, we will keep pruning off light edges and/or light vertices during the construction process. Consequently, the resulted subgraph will be an approximation to the true solution.

For our problem, we have a table with two columns. We will construct two suffix trees, one for each column. At the same time, we maintain all relations between the two columns by creating links between the two suffix trees.

We are now ready to describe our problem more formally.

3.2 The Problem: Revisited

Let F be an $M \times 2$ table as in Subsection 2.1. Let $S_j = \{F(i,j) | 1 \le i \le M\}$ be the set of strings contained in column j, $j \in \{1,2\}$. For any $0 \le k \le M$, let $T_{k,1}$ be the suffix tree formed from the strings contained in column 1 of the first k rows; similarly, let $T_{k,2}$ be the suffix tree formed from the strings contained in column 2 of the first k rows.

Define $V(k,j)$ to be the set of nodes in tree $T_{k,j}$, $j = 1,2$. Define weighted graph $G_k = (V_k, E_k, w_k)$, where $V_k = V(k,1) \cup V(k,2)$; $(u,v) \in E_k$ if and only if $u \in V(k,1)$, $v \in V(k,2)$, and for some row $i \le k$, $F(i,1)$ contains $p(u)$ and $F(i,2)$ contains $p(v)$; $w_k(u,v)$ is the number of such rows. For any node v corresponding to a string x, $w_k(v)$ is the number of rows within the first k rows which contain x as a substring.

Let G_0 be the empty graph. Let c_V and c_E be two threshold values (to be determined later). The goal of the off-line phase is to construct the heavy subgraph of G_M (with respect to c_V and c_E).

Note that each node v in the suffix trees has its own node weight $c(v)$ which is obtained in the usual way, i.e., each suffix tree is constructed based on one column and the node counts are obtained consequently. The tree edges in each suffix tree are not considered as graph edges, although we will need to use them to trace tree paths.

Graph G_M is usually huge. For example, if we have a table of $200K$ rows and two columns, each column has about l characters, then each suffix tree is of size $O(Ml)$ and G_M may contain $O(M^2 l^2)$ edges. Suppose each tree has $300K$–$450K$ nodes, then there will be about 9×10^{10}–20×10^{10} edges which cannot be handled easily.

4 Method: A General Description

As mentioned earlier, there are two phases: an off-line phase and an on-line phase. The goal of the off-line phase is to construct a catalog of reasonable size which will be used in the on-line phase to do the actual selectivity estimation.

As in [4, 5], we use suffix trees as our basic data structures. In the off-line phase, we first construct two (large) suffix trees, one for each column. The correlation between those two columns will be represented by weighted edges between the two trees. Then we prune the trees to reduce their size significantly and obtain two much smaller trees. The correlation information of the two trees is stored in a matrix of reasonable size. The two small trees and the associated matrix constitute the catalog.

As mentioned before, the simplest method (method I) is to construct the whole graph G_M during one scan of the table. We already know that this method is not practical due to storage limitation. Note that for the off-line phase, a large amount of storage is available, e.g., we can easily store the two separate suffix trees. The problem is that we cannot afford to store the relation between those two trees since the storage size then is the product of the sizes of the two individual suffix trees.

We may use the following two-pass alternative (method II). We scan the database twice. During the first scan, we construct two suffix trees T_1 and T_2 independently, where T_i is for column i, $i = 1,2$. When the scan is finished, each node in the trees has a node weight. Based on some threshold value, we prune the two trees and obtain two smaller trees (we still call them T_1 and T_2 respectively). During the second scan, we establish the desired relationship. Specifically, for any row, if substring x occurs in column 1, substring y occurs in column 2, and both x and y have corresponding nodes $v(x)$ and $v(y)$ in T_1 and T_2 respectively, then we build a weighted edge between the two nodes. If that edge already exists, we simply modify its weight.

Method II simulates method I very well as far as the final result (the heavy subgraph of G_M) is concerned. This is because the end nodes of a heavy edge are likely to be heavy too. Method II scans the table twice, undesirable in many applications.

Our method can somewhat be thought of as a one-pass implementation of method II. We will build approximations to the final heavy subgraph. We partition the whole database into sections and bring in each section only once. Once we have one section available, we can do some preprocessing and obtain all necessary information. Those information is then used to update the current approximate solution.

More specifically, during the scan, we maintain two suffix trees of reasonable size, together with a matrix which represents the current relationship between the two trees. The two suffix trees and the matrix constitute an approximate solution. They are updated periodically, i.e., they are updated once every section is brought in and preprocessed.

Note that our method can be used to solve the general DGCP problem. That is, we build the dynamic graph by pe-

riodically pruning out light vertices/edges. The method will work well if the node weight and edge weight are correlated "smoothly", i.e., the end vertices of heavy edges are likely to be heavy.

In the on-line phase, we use the output of the off-line phase to do selectivity estimation. Since the catalog is very small, not all information about the graph G_M is recorded accurately and we must use some heuristic strategies to do the estimation. We mainly adapt the various strategies of [4, 5] and generalize them to the two-column case.

5 Off-line Phase

In this section we describe the details of the off-line phase. Note that the input to the off-line phase is an M by 2 table F, where M is very large. Suppose l_i is the maximum length of any string occured in column i.

We maintain five data structures during the off-line phase: two suffix trees T_1 and T_2, each of size m; an $m \times m$ matrix R; and two arrays of pointers L_1 and L_2. Here m is a parameter determined by the storage available. (In our experiments, we choose $m = 1000$.)

We choose another parameter n which is also determined by the storage available. (In our experiments, we choose $n = 1000$). We bring in and preprocess data in F section by section, with each section containing n rows. At any time, the sections already brought in and preprocessed form a *view* of F.

Each node v in T_1 (T_2) has a (node) count $c(v)$ which is the number of rows in the current view of F that contain $p(v)$ as a substring in the first (second) column. Each matrix entry $R(u, v)$ is an approximation to the number of rows in the current view of F that contains $p(u)$ in column 1 and $p(v)$ in column 2. We call $R(u, v)$ the *common count* between $p(u)$ and $p(v)$. Note that node counts are node weights and common counts are edge weights.

When we scan a section, we construct two arrays L_1 and L_2, both of size n, where each $L_1(i)$ is again an array of size $h_i = l_i(l_i + 1)/2$. Note that h_i is an upper bound on the number of substrings which could occur in column i of any individual row. $L_1(i)$ ($1 \leq i \leq n$) records pointers to all suffix tree nodes in T_1 which correspond to some substring in column 1 of row i of the current section. L_2 is defined similarly for column 2.

Suppose we already have the current versions of T_1, T_2, R, L_1 and L_2. In scanning the next section, for each row i, we insert all suffixes of column j ($j = 1, 2$) into tree T_j, and record the corresponding pointers into $L_j(i)$. When the scan of this section is completed, both suffix trees grow (significantly) with the node weights updated. Now we prune out all light nodes and keep only the heaviest m nodes in each tree. Note that this pruning might cut off some old nodes and add some new nodes. Relabeling is done to make sure that all nodes receive a valid numbering between 1 and m. The goal is to maintain two suffix trees T_1 and T_2 so that T_i ($i = 1, 2$) contains the heaviest m nodes with respect to column i of the current view of F.

Next, we update the matrix R according to the two pointer arrays L_1, L_2 and the updated suffix trees. Specifically, for all $1 \leq i \leq m$, for any pointers $p_1 \in L_1(i)$ and $p_2 \in L_2(i)$, if p_1 points to an old node u in T_1 and p_2 points to an old node v in T_2, we increase $R(u, v)$ by 1; if p_1 points to a new node u in T_1 and/or p_2 points to a new node v in T_2, we delete the old entries $R(u, *)$ (and/or $R(*, v)$) and insert the new entries. When finished with all n rows, we will have a new matrix R which records the correlation between nodes in the updated suffix trees. We clear the two arrays L_1, L_2 and proceed to the next section.

After the scan is finished, we obtain two suffix trees T_1 and T_2 of size m and an $m \times m$ correlation matrix R. We further prune these trees and reduce the size of R. Specifically, we prune off all light edges and light nodes according to some threshold value c_E and only maintain their corresponding entries in R. In our implementation, the value c_E is chosen in such a way that after the pruning, there are ≤ 100 nodes remaining in either suffix tree (note that if all incident edges of a node have been pruned off, the node will also be pruned off).

In our implementation we have modified the above method a little. In pruning the two trees, we only delete a node if all its adjacent edges are light.

Finally, we obtain the following: two suffix trees T_1, T_2 of sizes t_1 and t_2 respectively, and a $t_1 \times t_2$ matrix R, where $t_1, t_2 \leq 100$. For simplicity, we assume $t_1 = t_2 = 100$. T_1, T_2 and R constitute the output of the off-line phase.

The final catalog has a very interesting and useful property due to the special implementation of pruning, that is, if substring x has a corresponding node u in T_1 and substring y has a corresponding node v in T_2, then the matrix entry $R(u, v)$ gives a good estimation $R(u, v)/M$ for $s(x, y)$. Note that if we prune off all light edges, it is possible that x, y have corresponding nodes u, v in the trees but the entry $R(u, v)$ does not reflect the common count between x and y.

Also note that to further reduce the size of the catalog (which is of course desirable in any circumstance), we can use adjacent lists to represent the matrix R. In our experiments, however, R is very dense. Moreover, matrix representation allows faster search. So we include matrix R as part of the catalog.

6 On-line Phase

In the on-line phase, a query $P = (p_1, p_2)$ is given. We need to estimate the selectivity $s(P)$ based on the information contained in the catalog which consists of two small

suffix trees T_1, T_2 and a matrix R. Note that this catalog is an approximation to the heavy subgraph of the weighted graph G_M.

We will use the following general strategy. First, we search the two patterns p_1 and p_2 in T_1 and T_2 respectively, trying to match them with some nodes in the trees. If we can find two matching nodes, say node u (for p_1) and node v (for p_2), we return $R(u,v)/M$ as an estimation to $s(P)$.

In most cases we cannot find exact matches for the two given patterns. In those cases we will use some *mismatch strategy* to estimate $s(P)$.

In [4, 5] many mismatch strategies have been developed for the one-column problem. Our main job here is to adapt those strategies to cope with two columns.

As in [4, 5], there are two classes of mismatch strategies, most of them are based on a *greedy parsing* of the given query patterns.

In the next subsections, we describe various mismatch strategies in details. But first let us describe greedy parsing.

Let T be a suffix tree and σ be a string. We search T, starting from the root, and try to find the longest prefix of σ which matches the substring represented by a node in T. Let this prefix be $\sigma(0)$. Next we start from the root again, try to search for the longest match with a prefix of the remaining substring, and so on. If at some time we cannot even find one match right from the root, we take the first character of the remaining string as a search result.

The parsing results is a partition of σ, $\sigma = \sigma(0)\sigma(1)\cdots\sigma(m)$, where $|\sigma(i)| > 0$ for all $0 \leq i \leq m$, and each $\sigma(i)$ is either the maximal match of σ_j in T ($j = |\sigma(0)| + \cdots + |\sigma(i-1)|$), or else the single character at position j of σ if no non-null prefix of σ_j successfully matches in T.

For a given query $P = (\alpha, \beta)$, we will use $e(X; P)$ to denote the estimate of $s(P)$ using mismatch strategy X. When X is known from the context, we omit X from our notation.

6.1 Independence-based Strategies

In using independence-based strategies, we parse the given patterns into many sub-patterns which can be matched in the two suffix trees. We then assume independence among all sub-patterns and give our overall estimation based on the selectivity of those independent matchable sub-patterns.

There are seven independence-based strategies. We denote them by $I_i, 1 \leq i \leq 7$.

Strategy I_1: Let α and β be greedily parsed as follows:

$$\alpha = \alpha(0)\cdots\alpha(m), \beta = \beta(0)\cdots\beta(n) \qquad (1)$$

Then $e(P)$ is defined as

$$e(P) = \prod_{i=0}^{m}\prod_{j=0}^{n} e(\alpha(i), \beta_(j)),$$

where $e(\alpha_(i), \beta_(j))$ is an appropriate matrix entry in R divided by M (the number of rows in the database). Here we assume that if one of $\alpha(i)$ and $\beta(j)$ does not have an exact match in its corresponding tree, i.e., there is not a corresponding entry in R, we let $e(\alpha(i), \beta(j)) = 0$.

Basically, this strategy assume independence among different subpatterns.

Strategy I_2: This is the same as I_1 except that if $e(I_1; P) = 0$, we set $e(I_2, P) = c_E/M$ where c_E is the edge threshold value (also called *common prune count*). That is, we assign the common prune count to all "unlikely" patterns.

Strategy I_3: This is similar to I_1, but we modify the estimation of $e(\alpha(i), \beta(j))$: if one of $\alpha(i)$ and $\beta(j)$ does not have an exact match in its corresponding tree, we set $e(\alpha(i), \beta(j)) = c_E$.

Strategy I_4: Suppose $|\alpha| = l_1$, $|\beta| = l_2$. Let $L_1 = l_1(l_1+1)/2$, $L_2 = l_2(l_2+1)/2$. Consider all suffixes α_i of α and all suffixes β_j of β, where $1 \leq i \leq l_1, 1 \leq j \leq l_2$. We define $e(I_4; P)$ as

$$e(I_4; P) = \sum_{i=0}^{l_1-1}\sum_{j=0}^{l_2-1} w_{ij} e(I_1; \alpha_i, \beta_j),$$

where w_{ij} is a weight. In our implementation, we choose $w_{ij} = w1_i w2_j$ where $w1_i = (l_1 - i)/L_1$ and $w2_j = (l_2 - j)/L_2$.

The rationale behind this strategy is that we expect the average selectivity of all suffixes would give a good estimation on the selectivity of the string itself.

Strategy I_5: In I_4, replace I_1 by I_2.

Strategy I_6: In I_4, we replace the two weight functions $w1, w2$ by

$$w1_i = 2^{l_1-i}/L_1, w2_j = 2^{l_2-j}/L_2,$$

and we also change L_1 and L_2 correspondingly, i.e.,

$$L_1 = 2^1 + \cdots + 2^{l_1} = 2^{l_1+1} - 2, L_2 = 2^1 + \cdots + 2^{l_2} = 2^{l_2+1} - 2.$$

Strategy I_7: In I_6, replace I_1 by I_2.

6.2 Child Estimation-based Strategies

This class of strategies are based on estimations on the children patterns of the given patterns.

Suppose we are given two strings, say s and t. Let the first segments of the greedy parsings of s and t be $s(0)$ and $t(0)$, respectively. Let the corresponding node of $s(0)$ in T_1 be u, and let the corresponding node of $t(0)$ in T_2 be v. Suppose v has children $v_1, \ldots, v_k$. We define a quantity Δ as follows:

$$\Delta = R(u, v) - \sum_{i=1}^{k} R(u, v_i).$$

We further define an "uncounted" count $uc(s, t)$ of s with respect to t. If either s or t is an *impossible pattern* , we define $uc(s, t) = -1$. Here by impossible pattern we mean a pattern which cannot occur in a suffix tree before it is pruned. Otherwise we define

$$uc(s,t) = \begin{cases} \Delta & \text{if } \Delta > 0 \\ c_E & \text{if } \Delta < 0 \end{cases}$$

We then estimate the number of pruned children of v as

$$nc(s,t) = \begin{cases} -1 & \text{if } uc(s,t) < 0 \\ \lceil uc(s,t)/c_E \rceil & \text{otherwise} \end{cases}$$

Finally we have the following estimation on s with respect to t:

$$es(s,t) = \begin{cases} 0 & \text{if } uc(s,t) < 0 \\ \frac{uc(s,t)}{c_E \times M} & \text{otherwise} \end{cases}$$

Now we are ready to describe all the child estimation-based strategies. There are four of them.

Strategy CE_1: We consider all the suffixes of the two given patterns α and β. We estimate $s(P)$ according to the following formula:

$$e(P) = \min_{0 \le i < l_1} \min_{0 \le j < l_2} es(\alpha_i, \beta_j).$$

This strategy is based on the fact that if two strings occur in the same row, any suffix of the first string and any suffix of the second string must occur in that row as well.

Strategy CE_2: Consider the greedy parsings (1) of α and β. We then have

$$e(P) = \prod_{i=0}^{m} \prod_{j=0}^{n} \frac{es(\alpha(i), \beta(j))}{e(\alpha(i), \beta(j))}$$

That is, we consider the children of all subpatterns as independent of each other.

Strategy CE_3: This is similar to CE_1. We still define

$$e(P) = \min_{0 \le i < l_1} \min_{0 \le j < l_2} es'(\alpha_i, \beta_j),$$

but we change es as follows:

$$es'((\alpha_i, \beta_j)) = \begin{cases} e(\alpha_i, \beta_j) & \text{if there is an exact match} \\ es(\alpha_i, \beta_j) & \text{otherwise} \end{cases}$$

Strategy CE_4: Consider the greedy parsing (1). We use the following formula (derived from Bayes rule) to estimate $s(P)$:

$$e(P) = e(\alpha(0), \beta(0)) \prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \frac{es(\alpha(i), \beta(j))}{e(\alpha(i), \beta(j))}$$

7 Depth-based Estimation

7.1 The Basic Method

Recall that all the previous discussions are based on a catalog consisting of two suffix trees T_1, T_2, both of size 100, and a 100×100 matrix R. In turn, these are obtained from two larger suffix trees of size m ($m = 1000$ in our implementation) and a matrix of order m. Now we begin from there and produce another catalog which is much smaller than the current catalog.

Suppose tree T_1 has depth d_1 and tree T_2 has depth d_2. For any $0 \le i \le d_1, 0 \le j \le d_2$, we consider the average weight of all edges which link a node of depth i in T_1 to a node of depth j in T_2. This average weight can be obtained easily from the m by m matrix R. We store it as the (i, j) entry of a $(d_1 + 1) \times (d_2 + 1)$ matrix D. Matrix D is called the *depth statistics table* , or DST.

Then we prune the two trees as before, obtaining two small suffix trees of size 100. Note that we don't need node weights anymore. Instead, we only need to maintain the structures of those trees. We still call them T_1 and T_2.

As a result, we obtain a new catalog which consists of T_1, T_2, and the DST table D. The size of D depends on the depths of the original large suffix trees, and is usually small. For example, in our experiments, $d_1 = d_2 = 7$, and we only need to maintain a matrix of order 7.

In the on-line phase, for any given patterns $P = (\alpha, \beta)$, we first do a greedy parsing and obtain

$$\alpha = \alpha(1) \cdots \alpha(m), \beta = \beta(1) \cdots \beta(n).$$

Then we search for those sub-patterns in the corresponding trees and find out their depths. We use $d(T, x)$ to denote the depth of the node corresponding to string x in tree T. If x does not have a corresponding node in T, we consider it as corresponding to a pruned node at depth 1 and set $d(T, x) = 1$.

Now we define the depth of α as a weighted average of the depths of its sub-patterns,

$$d(\alpha) = \sum_{i=1}^{m} w_i \times d(T_1, \alpha(i)). \tag{2}$$

Similarly, we define the depth of β as

$$d(\beta) = \sum_{j=1}^{n} w_j \times d(T_2, \beta(j)). \tag{3}$$

In the above formulas, w_i's are weights to be determined later.

Once we have determined $d(\alpha)$ and $d(\beta)$, we look into the DST table D and return $D(d(\alpha), d(\beta))$ as our estimation $e(P)$. Since $d(\alpha)$ and $d(\beta)$ may not be integers, we need to adjust them. Keeping this in mind, we arrive at the following formula:

$$e(\alpha, \beta) = \frac{D(\lfloor d(\alpha) \rfloor, \lfloor d(\beta) \rfloor) + \dot{d}(\alpha) \times \dot{d}(\beta) \times \dot{D}(\alpha, \beta)}{M}$$

where: $\dot{d}(\alpha) = d(\alpha) - \lfloor d(\alpha) \rfloor$, $\dot{d}(\beta) = d(\beta) - \lfloor d(\beta) \rfloor$, and $\dot{D}(\alpha, \beta) = D(\lceil d(\alpha) \rceil, \lceil d(\beta) \rceil) - D(\lfloor d(\alpha) \rfloor, \lfloor d(\beta) \rfloor)$.

Depending on the choice of the weights, we have three different depth estimation-based strategies.

Strategy DE_1**:** Choose $w_i = 1/i$.

Strategy DE_2**:** Choose $w_i = 1/(2i - 1)$.

Strategy DE_3**:** Choose $w_i = 1/2^i$.

In our experiments, we found that DE_1 and DE_2 are better than DE_3. An interesting observation is that the accuracy of the estimation is very sensitive to the choice of weight functions. This suggests that we should try to find some good weight functions.

In the next subsection, we formulate the problem of finding a weight function as a least square problem and we then show that the latter problem can be solved by collecting necessary information in the off-line phase.

7.2 A Least Square Problem

For simplicity, we discuss the case when the database table F has only one column. Suppose each row contains a string of length at most l. Therefore, any substring σ in the database can be greedily parsed into at most l sub-patterns against any suffix tree.

Suppose in the off-line phase, we have constructed the whole suffix tree T_0 for this single column. Then any substring s occuring in F has a depth in T_0, and we denote this depth by $d_0(s)$. We can enumerate all substrings contained in F in any particular order. Suppose there are a total of L substrings in the enumeration, then the depth of the ith substring s_i in T_0 is denoted by y_i, $1 \le i \le L$. Note that $L \le Ml(l+1)/2$.

Suppose we prune T_0 and obtain a much smaller tree T (of size 100). Suppose the depth of T is d. For any substring s_i, we can use greedy parsing on s_i against T and obtain

$$s_i = s_i(0) \cdots s_i(n),$$

where $n \le l - 1$. This parsing will give the following equation (according to (2) and (3)):

$$d(T_0, s_i) = \sum_{j=0}^{n} w_j d(T, s_i(j)),$$

where $0 \le d(T, s_i(j)) \le d$. We can rewrite it as

$$y_i = \sum_{j=0}^{n-1} w_j x_{ij}, 1 \le t \le L \tag{4}$$

where we use x_{ij} to denote the depth of the jth sub-pattern of s_i in the small tree T.

Since we can construct both T_0 and T easily during the off-line phase, we can consider all y_i's and x_{ij}'s as known numbers. We need to solve equation (4) for the to-be-determined weights $w_0, w_1, \ldots, w_{l-1}$.

Let $Y = (y_1, \ldots, y_L)^T \in R^{L \times 1}$, $X = (x_{ij}) \in R^{L \times l}$, and $w = (w_0, \ldots, w_{l-1})^T \in R^{l \times 1}$. Then we can rewrite equation (4) as

$$Y = Xw \tag{5}$$

Our goal is to solve equation (5).

Equation 5 is very unlikely to be solvable. Actually we don't need to solve it at all. We only need to solve the following Least Square problem:

$$\min \sum_{i=1}^{L} (y_i - \sum_{j=0}^{l-1} x_{ij} w_j)^2 \tag{6}$$

In other words, we only need to "solve" equation 5 for an approximate solution.

Left multiplying X^T on both sides of 5, we obtain

$$X^T X w = X^T Y.$$

Denote $B = X^T X \in R^{l \times l}$, $Z = X^T Y \in R^{l \times 1}$. Note that B and Z are very small matrices now. So we only need to solve the following small system of equations:

$$Bw = Z. \tag{7}$$

The problem is how to obtain B and Z. By examing the definition of B and Z, we immediately see that each entry in both B and Z can easily be accumulated during a second scanning of the whole database table F once we have constructed T_0 and T.

It is important to notice that the two-pass scanning is needed mainly for determining the weight w. We can conduct several experiments on different database tables (each needs to be scanned twice) and try to find a good choice for the weight function w. Once we have such a good weight function w available, we can use it in other applications if we want to use a depth-based estimation method, in the hope that w is a good approximation to the true weight functions of current applications.

We point out the fact that in the experiment, the depth-based estimation method performs no better than other methods and strategies, even though we use a weight function obtained from solving a least square problem as above. This is because the method itself has inherent limitations.

8 Experiments

We conduct two groups of experiments. In group one, we test our method using artificial data. In group two, we test our method against real customer data.

8.1 Experiments using artificial data

We conduct our experiments according to the following guideline:

Data Generation: Generate large database tables with two alphanumeric columns. Generate query patterns involving the two columns.

Off-line Construction: Use the methods of Section 5 and Section 7 to build catalogs of reasonable size.

On-line Estimation: Use all the mismatch strategies of Section 6 and Section 7 to estimate the selectivity of query patterns.

Error Analysis: Compare the estimated selectivity with the exact selectivity.

8.1.1 Data Generation

Since no similar studies have been done before, no existing benchmark is available for our experiment. We must generate our own experimental data. According to the nature of the problem, we need a database table which contains two correlated columns of alphanumeric type.

There is an emerging industry standard Transaction Processing Council (TPC) benchmark, known as the TPC-D benchmark [15], that involves the predicates such as the LIKE predicate. Tables involving either numeric or alphanumeric data can be generated using TPC-D. The TPC-D data is used in the KVI method to conduct all the experiments. Unfortunately, all the alphanumeric type columns of TPC-D data are independent of each other.

We introduce correlation factor into the generation of TPC-D tables. More specifically, we generate our table F as follows. First, we generate the first column COL_1, using the standard TPC-D method. We then generate the second column based on the content of the first column.

There are 92 base patterns specified in the benchmark, where each of the patterns is a color, e.g., "green". Each entry of COL_1 is a pair of two randomly chosen base colors separated by an underscore "_". Thus, each element in COL_1 is of the form $p_1_p_2$, where both p_1 and p_2 are base colors.

Each element of the second column, COL_2 is generated based on the content of COL_1 in the same row. To generate COL_2 elements, we partition the 92 base colors into six groups, g_1, g_2, g_3, g_4, g_5, and g_6. Each group is partitioned into four equal size subgroups. The partition is shown in Table 1.

group name	*group size*	*subgroup size*
g_1	4	1
g_2	8	2
g_3	12	3
g_4	16	4
g_5	20	5
g_6	32	8

Table 1. Partition of the 92 base patterns

Assume that at the ith row, COL_1 is $p_1_p_2$, we'll generate the corresponding COL_2 element $q_1_q_2$ as follow: First we determine which subgroups contain p_1 and p_2. Suppose p_1 is in subgroup g_i and $p2$ is in subgroup g_j. With a probability of λ, we randomly choose a color in g_i and a color in g_j as q_1 and q_2 respectively. With a probability of $1-\lambda$, we randomly and independently choose two colors from the 92 base colors as q_1 and q_2 respectively. In our experiments, we choose $\lambda = 80\%$.

We can see that the smaller the subgroup g_i is, the stronger the correlation between p_1 and q_1 will be. The same is true for p_2 and q_2. Thus, the correlation between the two columns depends on the partition of the base color. The correlation also depends on λ.

In our experiments, we have generated a table F of $200K$ rows, i.e., $M = 200K$.

Our query pattern is of the form (p, q), where p and q are base colors. In our experiments, we have generated 476 queries .

8.1.2 Off-line Construction

In the off-line phase, we process F section by section, with each section containing 1000 rows. At the end of the processing of each section, we always obtain two trees of size 1000 (nodes) and a 1000×1000 matrix R.

After the whole table has been processed, we further prune the two trees and obtain T_1 and T_2, where T_1 has 96 nodes and T_2 has 100 nodes. Correspondingly, we reorganize R and obtain a 96 $\times$ 100 matrix R. T_1, T_2 and R constitute the catalog.

To use the depth-based estimation methods, we need to construct a different catalog, which (in our experiment) consists of two suffix trees T_1 and T_2, plus a small 7×7 DST table. The two trees are the same as before. The DST table stores depth-based common count information. The size 7 comes from the fact that the trees before the final pruning both have depth 6.

8.1.3 On-line Estimation

Although the final catalog contains two suffix trees, no query can be matched exactly. This is natural because the two trees can only record substring information due to its small size. So we must use one of the mismatch strategy to estimate selectivity.

We test all the mismatch strategies. The performance of those strategies is reported in the next subsection.

8.1.4 Error Analysis

We measure the performance of any mismatch strategy by computing its relative error. That is, for a given pattern P, if the true selectivity is $s(P)$, the estimated value (based on one mismatch strategy) is $e(P)$, we compute the relative error $(e(P) - s(P))/s(P)$. Note that we allow negative relative errors in our discussion, but it can never be less than -100%. Also note that based on the table F, we can do a simple exhaust search and find the exact value of the selectivity for any given query.

We plot the cumulative number of patterns along the y axis against the relative error on the x axis: a point (x, y) on the graph means that y patterns have relative error $\leq x\%$.

In the experiment, both strategy I_1 and strategy I_3 give 0 as the value of $e(P)$ for almost all patterns. This is not surprising because those strategies estimate $s(P)$ as the product of the selectivity of many subpatterns of P. Strategy I_2 either under estimate or over estimate: all zero values for subpatterns are replaced by edge weight (a large number) and all small values remain small. These three strategies are not acceptable and are not shown.

The performance of other strategies for the 476 patterns is shown in Figure 1–Figure 3. As we can see from the graphs, $I_4, I_5, CE_1, CE_3, DE_1$ and DE_4 are the best 6 strategies and we collect them in Figure 4.

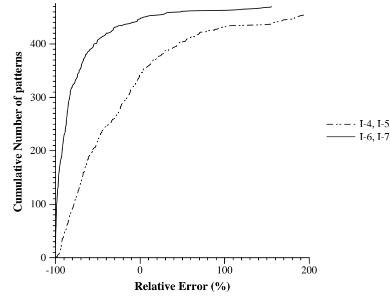

Figure 1. Performance of the Independence-based strategies for 476 patterns.

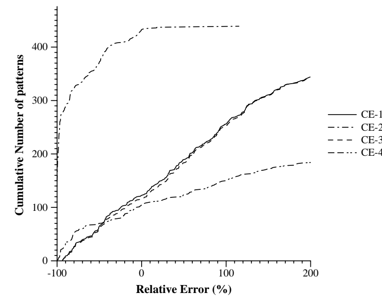

Figure 2. Performance of the Child Estimation-based strategies for 476 patterns.

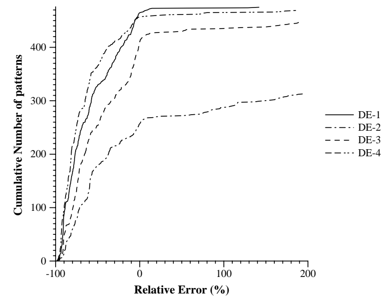

Figure 3. Performance of the Depth Estimation-based strategies for 476 patterns.

8.2 Experiments using real data

Our experiments in this group are all based on customer data and query obtained from one of our DB2 customers.

8.2.1 Description of data and query

The original database table contains N rows ($N \simeq 1$ million) and 36 columns. One of the columns, column 34, has a name SPECL_INSTR_TEXT. This column is of the varchar type (variable length character strings), with a maximum length 80 (characters). For example, the following sentences occur in this column,

- DONT JUMP FENCE
- SMALL DOG NO PITT/ SCE LOCK ON GATE

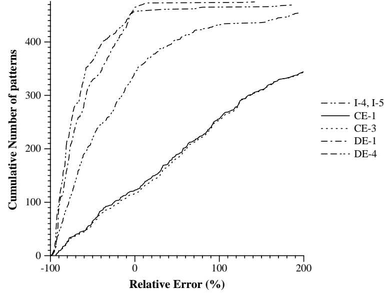

Figure 4. Graph of the best performing strategies for 476 patterns.

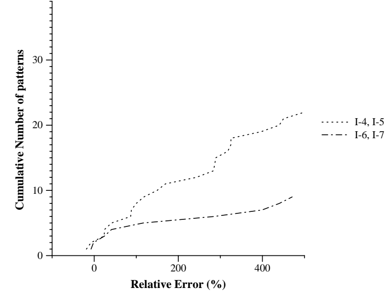

Figure 5. Performance of the Independence-based strategies for 39 DOG query pairs.

- ENT LEFT DOG ON RGT

For this particular column, users (home meter readers) were interested in asking queries of the following form (the *DOG query*):

SELECT *
FROM T
WHERE SPELCL_INSTR_TEXT LIKE '%BAD%'
OR SPELCL_INSTR_TEXT LIKE '%BEWARE%'
OR SPELCL_INSTR_TEXT LIKE '%DANGEROUS%'
OR SPELCL_INSTR_TEXT LIKE '%DOBIE%'
OR SPELCL_INSTR_TEXT LIKE '%DOBY%'
OR SPELCL_INSTR_TEXT LIKE '%DOG%'
OR SPELCL_INSTR_TEXT LIKE '%MEAN%'
OR SPELCL_INSTR_TEXT LIKE '%PIT%'
OR SPELCL_INSTR_TEXT LIKE '%VICIOUS%'
OR SPELCL_INSTR_TEXT LIKE '%DG%'

We single out this particular column from the table and form a file with N rows. We found that about 4/5 of the rows are empty. For convenience, we delete all empty rows and obtain our final file F based on which we will conduct our experiments. Incidently, the size of F is 200K, the exact same size as the file KVI used in their experiments. Note that the original experiments in [5] were based on the TPC-D benchmark where the input file is generated by artificial means. Our data are chosen from real database and it has a very different distribution. There are ten patterns in the DOG query and their selectivities vary from 0.000185 to 0.11072.

8.2.2 Using our method on DOG query

Although DOG query only involves one column in the table, the multiple patterns in the query make it very interesting to apply our two-column method to it. We just assume that the two columns in our method are exact the same and by doing that, we can apply our method directly to estimate the selectivity for the one-column two-pattern query :

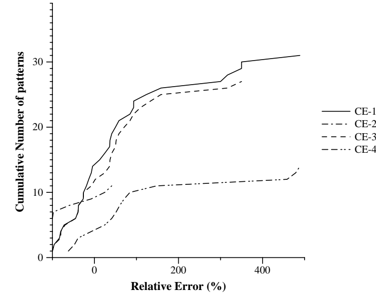

Figure 6. Performance of the Child Estimation-based strategies for 39 DOG query pairs.

$$COL \text{ like } *p_1* \text{ AND } COL \text{ like } *p_2*$$

One interesting thing to notice is that for this type of queries, the two trees build on the off-line stage are exactly the same and the final n by n matrix is symmetric. So we just need to keep one tree and half of the entries in the matrix in the catalog.

In our experiment, we generate all possible pairs of the ten single patterns. There are $10 * 9/2 = 45$ of them. Among them, 6 have selectivity of exact 0 which means the pattern pairs never appear simultaneously in the same row. So we only consider the remaining 39 pairs. Their exact selectivities vary from 0.000005 to 0.010995.

We notice that same as the KVI method for the real data, to obtain a reasonable accurate estimation for the DOG query pattern pairs, we need to keep a much larger tree and matrix in the catalog. Figure 5–Figure 7 plot the performance of different strategies for a catalog tree of 1000 nodes.

We found that very similar to the experiments for artificial data, $I_4, I_5, CE_1, CE_3, DE_1$ and DE_4 have the best performance strategies for real data. Their performance is plotted in Figure 8.

We point out that DE_1 and DE_4 are very attractive because the space used in the caltog in these depth-based es-

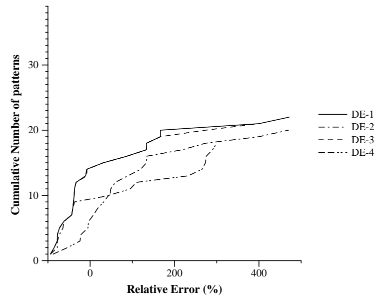

Figure 7. Performance of the Depth Estimation-based strategies for 39 DOG query pairs.

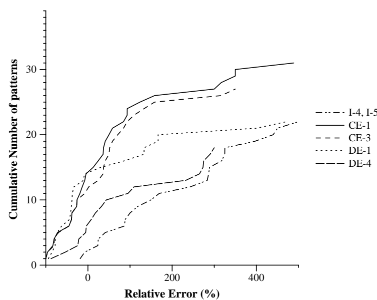

Figure 8. Graph of the best performing strategies for 39 DOG query pairs.

timation strategies is much smaller than those used in other strategies.

9 Conclusions

One important conclusion, from our experiments is that estimation of alphanumeric selectivity for real data is considerably more difficult than for artificial data. The data distribution can be arbitrary and larger space allocation in the catalog is recommended. Since this is so, there is an opportunity for data compression techniques.

A more interesting problem is to consider correlated columns of mixed data type. For example, we may consider two columns, one of them is of alphanumeric type and another is of numeric type. We may expect that these mixed data type problems can be solved much more efficiently.

Finally, we may consider using other data structures to handle alphanumeric data. Although the suffix tree is a linear space data structure for representing character strings, it is a irregular tree and the hidden constant is not small. It would be very interesting to find some other data structures for our problem.

References

[1] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic Dictionary Matching, *Journal of Computer and System Sciences*, 49: 208–222, 1994.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, The MIT Press, 1993.

[3] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching, *Proceedings of SIGMOD 1984*, 47–57.

[4] P. Krishnan. *Online Prediction Algorithms for Databases and Operating Systems*, PhD thesis, Brown University, April 1995.

[5] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards, *Proceedings of SIGMOD 1996*, 282–293.

[6] M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems, *ACM Computing Surveys*, 20(3): 191–221, 1988.

[7] E. M. McCreight. A Space-Economical Suffix Tree Const ruction Algorithm, *Journal of ACM*, 23(2): 262–272, 1976.

[8] H. Samet. The Quadtree and Related Hierarchical Data Structur es, *Computing Surveys*, 16(2): 187–280, 1984.

[9] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 23–34, 1979.

[10] G. P. Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition, *Proceedings of SIGMOD 1984*, 256–276.

[11] A. Swami and K. Schiefer. On the Estimation of Join Result Sizes, *Proc. of EDBT 1994*, 287–300.

[12] A. Tsang and M. Olschanowsky. A Study of Database 2 Customer Queries, IBM STL Report TR 03.413, April 1991.

[13] J. D. Ullman, *Principles of Database sSystems*, 2 nd edition, Computer Science Press, 1982.

[14] M. Wang, T. Beavin, H. S. Tie, and B. Iyer. A Method for Estimating Filter Factors for Multi-Column with Correlations, *IBM STL Report*, 1995.

[15] Transaction Processing Performance Council. TPC BE NCHMARK D, April 1995.