Multi-Dimensional Substring Selectivity Estimation

H. V. Jagadish*
U of Michigan, Ann Arbor
jag@eecs.umich.edu

Olga Kapitskaia
AT&T Labs–Research
olga@research.att.com

Raymond T. Ng
U of British Columbia
rng@cs.ubc.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

With the explosion of the Internet, LDAP directories and XML, there is an ever greater need to evaluate queries involving (sub)string matching. In many cases, matches need to be on multiple attributes/dimensions, with correlations between the dimensions. Effective query optimization in this context requires good selectivity estimates.

In this paper, we use multi-dimensional count-suffix trees as the basic framework for substring selectivity estimation. Given the enormous size of these trees for large databases, we develop a space and time efficient probabilistic algorithm to construct multi-dimensional *pruned* count-suffix trees directly. We then present two techniques to obtain good estimates for a given multi-dimensional substring matching query, using a pruned count-suffix tree. The first one, called *GNO* (for Greedy Non-Overlap), generalizes the greedy parsing suggested by Krishnan et al. [9] for one-dimensional substring selectivity estimation. The second one, called *MO* (for Maximal Overlap), uses all maximal multi-dimensional substrings of the query for estimation; these multi-dimensional substrings help to capture the correlation that may exist between strings in the multiple dimensions. We demonstrate experimentally, using real data sets, that MO is substantially superior to GNO in the quality of the estimate.

* Supported in part by NSF under grant IDM9877060.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

One often wishes to obtain a quick estimate of the number of times a particular substring occurs in a database. A traditional application is for optimizing SQL queries with the *like* predicate (e.g., `name` *like* `jones`). With the growing importance of the Internet, LDAP directory servers, XML, and other text-based information stores, substring queries are becoming increasingly common. Furthermore, in many situations for these applications, a query may specify substrings to be matched on multiple alphanumeric attributes/dimensions. The SQL query ((`name` *like* `mark`) `AND` (`tel` *like* `973360`) `AND` (`mail` *like* `jones`)) is one example. As another example, the LDAP query ([4]) that asks for directory entries in the subtree rooted at the directory entry whose distinguished name (`dn`) is `dc=research,dc=att,dc=com`, and that match the filter (`tel` = `*36087*`) can be modeled as a multi-dimensional string matching query (& (`dn` = `*dc=research,dc=att,dc=com`) (`tel` = `*973360*`)). The first string component `dn` = `*dc=research,dc=att,dc=com` matches all the entries in AT&T Research, and the second component `tel` = `*973360*` specifies a substring match. Often times, the attributes mentioned in these kinds of multi-dimensional queries may be correlated. For the above LDAP example, because of the geographical location of the research labs, people in AT&T Research may have an unexpectedly high probability to satisfy `tel` = `*973360*`. For such situations, assuming attribute independence and estimating the selectivity of the query as a product of the selectivity of each individual dimension can be grossly inaccurate.

In this paper, we study the problem of multi-dimensional substring selectivity estimation, and make

the following contributions:

- We propose a novel generalization of 1-D count-suffix trees [9], referred to as a *k-D count-suffix tree*, as the basic data structure for solving the problem (Section 2). Our trees can handle not only substring matches, but also prefix, suffix and exact matches.

- Given the enormous size of these trees for large databases and for multiple dimensions, it is desirable, and often essential, to try to obtain a compressed representation that satisfies given memory restrictions. To this end, we develop a space and time efficient probabilistic algorithm to construct a k-D *pruned* count-suffix tree without first having to construct the full count-suffix tree (Section 3).

- What we gain in space by pruning a count-suffix tree, we lose in accuracy for estimating the selectivities of those strings that are not completely retained in the pruned tree. Our main challenge, then, is: given a pruned tree, to try to estimate as accurately as possible the selectivity of such strings.

  We develop and analyze two algorithms for this purpose (Section 4). The first algorithm, called *GNO* (for Greedy Non-Overlap), generalizes the greedy parsing suggested by Krishnan et al. for 1-D substring selectivity estimation [9]. The second algorithm, called *MO* (for Maximal Overlap), uses all maximal k-D substrings of the query for estimation, to take advantage of correlations that may exist between strings in the multiple dimensions.

- We present an experimental study, using a real 2-D data set, that compares the accuracy of our two algorithms, GNO and MO, and additionally compares them with the default assumption of attribute independence (Section 5). Our results show the practicality and the superior accuracy of MO, demonstrating that it is possible to obtain freedom from the independence assumption for correlated string dimensions.

1.1 Related Work

1-D suffix tree [18, 11] is a commonly used structure for indexing substrings in a database [9]. One natural generalization of strings is a multi-dimensional matrix of characters. The pattern matching community has developed data structures, also referred to as suffix trees, for indexing sub-matrices in a database of such matrices (see, e.g., [1, 2]). The problem of indexing sub-matrices is clearly a different problem than indexing substrings in multiple correlated dimensions, and the suffix tree developed for the sub-matrix matching problem does not seem applicable to our problem. Our problem, despite its importance, appears to have received much less attention in the literature.

1-D pruned count-suffix trees were studied in [9], and algorithms for the direct construction of the pruned count-suffix tree (i.e., without first constructing the complete count-suffix tree) were proposed. However, those techniques were ad hoc in the sense that no quality guarantees were provided. Our approach of direct construction of a k-D pruned count-suffix tree builds upon the concise sampling technique proposed in [3], provides probabilistic guarantees on the number of false positives and false negatives, and gives *accurate* counts for the substrings in the pruned count-suffix tree.

Histograms have long been used for selectivity estimation in databases [15, 12, 10, 5, 6, 13, 7]. They have been designed to work well for numeric attribute value domains, and one can obtain good solutions to the histogram construction problem using known techniques (see, e.g., [13, 7]). For string domains, and the substring selectivity estimation problem, one could continue to use histograms by sorting substrings based on the lexicographic order, and associating the appropriate counts. However, in this case, a histogram bucket that includes a range of consecutive lexicographic values is not likely to produce a good approximation, since the number of times a string occurs as a substring is likely to be very different for lexicographically successive substrings.

End-biased histograms are more closely related to pruned count-suffix trees [6]. The high-frequency values in the end-biased histogram correspond to nodes retained in the pruned count-suffix tree. The low-frequency values correspond to nodes pruned away. With this approach of estimating the selectivity of substring queries, if α_1 has been pruned, the same (default) value is returned for α_1 and $\alpha_1\alpha_2$, irrespective of the length of α_2.

In spite of the vast literature on histograms, there is very little discussion of histograms in multiple dimensions. A notable exception is the study in [14]. But for the reasons given in the preceding paragraph, this study is not directly applicable to the problem of substring selectivity estimation in multiple dimensions.

A study of 1-D substring selectivity estimation is presented in [9]. Experimental evaluation of various versions of independence-based, child-based and depth-based strategies is given. Among those, a specific version of the independence-based strategies, referred to here as the KVI algorithm, is shown to be one of the most accurate. The GNO algorithm presented here generalizes the KVI algorithm from 1-D to k-D.

In [8], we conducted a formal analysis on 1-D substring selectivity estimation. We compared a suite of algorithms, including KVI and a 1-D version of MO, in terms of the accuracy of their estimates (expressed

as log ratios) and their computational complexities. The MO estimation algorithm presented here for k-D strings generalizes the 1-D version analyzed there. As will be shown later, the generalization is not straightforward.

A study of k-D substring selectivity estimation is given in [17]. There are several key differences between that study and the work presented here. First, at a data structure level, k-D substring selectivity estimation in [17] is based on k separate 1-D pruned count-suffix trees and a multi-dimensional array. In our case, the estimation is based on a k-D count-suffix DAG. Second, for constructing pruned data structures, only ad hoc heuristics are considered in [17]. In our case, we develop a space and time efficient probabilistic algorithm. Third, for selectivity estimation, a generalization of the KVI algorithm, as well as child-based and depth-based strategies are developed in [17]. That generalization does greedy parsing independently in each of the k dimensions, using the 1-D pruned count-suffix trees, and computes an estimate for the k-D substring selectivity based on the information in the multi-dimensional array. This technique can be considered as a simple version of the GNO algorithm proposed here. As will be shown later, the MO algorithm proposed here is superior to the GNO algorithm.

2 k-D Structures for Estimation

In this section, we present k-D generalizations of tries and suffix trees for our multi-dimensional estimation problem. To the best of our knowledge, these data structures have not been studied in the literature.

Throughout this paper, we use $\mathcal{A}$, possibly with subscripts, to denote an alphabet for an attribute; and Greek lower case symbols α, β, to denote strings of finite length ≥ 0 in $\mathcal{A}^*$. For simplicity, we do not distinguish between a character in $\mathcal{A}$, and a string of length 1. We use ε to denote the null string.

By a *k-D string*, we mean a k-tuple $(\alpha_1, \ldots, \alpha_k)$, where $\alpha_i \in \mathcal{A}_i^*$ for all $1 \leq i \leq k$. A *k-D substring* of a given k-D string $(\alpha_1, \ldots, \alpha_k)$ is $(\gamma_1, \ldots, \gamma_k)$, such that γ_i is a (possibly empty) substring of $\alpha_i, 1 \leq i \leq k$.

2.1 k-D Count-Tries

In 1-D, a *count-trie* is a trie that does not store pointers to occurrences of the substrings α in the database. Instead, it keeps a count C_α at the node α in the trie. The count C_α can have (at least) two different meanings. First, it can denote the number of strings in the database $\mathcal{D}$ containing α as a substring; we call this *presence*-counting. Second, it can denote the number of occurrences of α as a substring in the database $\mathcal{D}$; we call this *occurrence*-counting. Suppose $\mathcal{D}$ contains only the string `banana`. With the first interpretation, C_{ana} would be 1, but with the second interpretation, C_{ana} would be 2. Both interpretations are obviously useful in different applications. In this paper, our exposition focuses only on presence-counting; all the concepts and techniques presented here carry over easily to occurrence-counting.

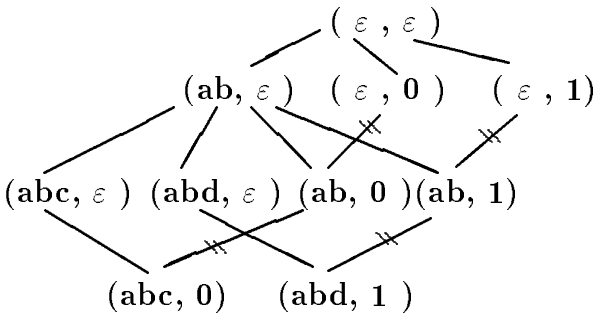

Figure 1: Example 2-D Trie

In k-D, a *count-trie* is a rooted DAG that satisfies the following properties:

- Each node is a k-D string. The root is the k-D string $(\varepsilon, \ldots, \varepsilon)$.
- There is a (directed) edge between two nodes $(\alpha_1, \ldots, \alpha_k)$ and $(\beta_1, \ldots, \beta_k)$ iff:
  - there exists $1 \leq i \leq k$ such that α_i is an immediate prefix of β_i; and
  - for all $j \neq i$, $1 \leq j \leq k$, $\alpha_j = \beta_j$.

By "immediate prefix," we mean that there does not exist another node $(\ldots, \gamma_i, \ldots)$ in the trie, such that α_i is a proper prefix of γ_i, and γ_i is in turn a proper prefix of β_i.

Figure 1 shows the 2-D count-trie for a database with the two 2-D strings $(abc, 0)$ and $(abd, 1)$. The root node $(\varepsilon, \varepsilon)$ and the node (ab, ε) have count = 2, while the remaining ones all have count = 1.

As is done for standard 1-D tries, a simple optimization can be applied to compress k-D count-tries. For any two nodes connected by an edge, there is no need to store the common prefix twice. In Figure 1, for instance, the node (abd, ε) can simply be stored as (d, ε); we show the prefix in the figure only for clarity.

2.2 k-D Count-Suffix DAGs

In 1-D, a suffix tree [18, 11] is a trie that satisfies the following property: whenever a string α is stored in the trie, all suffixes of α are stored in the trie as well. The same property is preserved for k-D *count-suffix DAGs*, which are k-D count-tries. Specifically:

Property P1: for any k-D string $(\alpha_1, \ldots, \alpha_k)$ in the count-suffix DAG, the k-D strings $(\gamma_1, \ldots, \gamma_k)$ are also in the DAG for *all* (improper) suffixes γ_i of $\alpha_i, 1 \leq i \leq k$.

For example, to make the trie shown in Figure 1 a 2-D count-suffix DAG for $(abc, 0)$ and $(abd, 1)$, we need

to add the strings/nodes $(bc, 0)$, (bc, ε), $(c, 0)$, (c, ε), $(bd, 1)$, (bd, ε), $(d, 1)$, and (d, ε), and the corresponding edges.

Just like standard 1-D count-suffix trees, k-D count-suffix DAGs support substring matches. To continue with the simple example above, the query $((\texttt{attr1} = *b*) \,\&\, (\texttt{attr2} = *0*))$ is matched by the node $(bc, 0)$. [1] Similarly, the query $((\texttt{attr1} = *ab*) \,\&\, (\texttt{attr2} = *))$ can be matched by the node (ab, ε). However, it is important to note that count-suffix DAGs cannot handle queries of the forms:

- `attr1` beginning with ab (i.e., prefix match),
- `attr1` ending with ab (i.e., suffix match), and
- `attr1` matching the string abc (i.e., exact match).

Even though in the above example, the node (ab, ε) appears to have handled the prefix query "`attr1` beginning with ab", it really does not. The reason is that if there is a string (cab, α_2), say, in the database, then according to Property P1 above, (ab, γ_2) for all suffixes γ_2 of α_2 must have been inserted in the DAG. In other words, the count associated with the node (ab, ε) includes not only strings with ab as the prefix, but indeed all strings with ab as a substring.

It turns out that a simple trick is sufficient to make the count-suffix DAG capable of handling all the variations mentioned above. For each string, we add two special characters: # attached to the beginning, \$ appended at the end of the string. As far as insertion into the count-suffix DAG is concerned, these two special characters behave like any other "normal" character in the alphabet. For example, for the 2-D string $(abc, 0)$, we first convert it to $(\#abc\$, \#0\$)$, and then insert all relevant strings based on Property P1 above. As far as querying is concerned, a prefix match to the string "ab" can be specified as a substring match on the (extended) string "#ab" to the count-suffix DAG. Similarly, a suffix match (resp., an exact match) to string "ab" can be specified as a substring match on the string "ab\$" (resp., "#ab\$").

2.3 Compressed Representation: k-D Count-Suffix Trees

Even though with the simple trick discussed above, we have augmented the query answering capabilities supported by a count-suffix DAG, each query search still begins from the root of the DAG. From this standpoint, a k-D count-suffix DAG is an overkill in the sense that the edges in the DAG allow a search to begin from any node in the DAG (e.g., from $(\varepsilon, 0)$ to $(abc, 0)$ in Figure 1). Thus, to reduce space, we seek to compress a k-D count-suffix DAG into a k-D *count-suffix tree*, while preserving the desired query answering capabilities.

[1] In a trie, the node $(bc, 0)$ is a compressed representation of two nodes $(b, 0)$ and $((b)c, 0)$.

To do so, we first pick a canonical enumeration of the attributes. [2] Without loss of generality, let us assume that the enumeration order is attributes 1 to k. Then for any node $(\alpha_1, \ldots, \alpha_k)$ in the count-suffix DAG, we define the following path from the root to the node as the *canonical path*:

$$\begin{aligned}&(\alpha_{1,1}, \varepsilon, \ldots, \varepsilon), (\alpha_{1,2}, \varepsilon, \ldots, \varepsilon), \ldots, (\alpha_{1,m_1}, \varepsilon, \ldots, \varepsilon),\\ &\quad (\alpha_1, \alpha_{2,1}, \varepsilon, \ldots, \varepsilon), \ldots, (\alpha_1, \alpha_{2,m_2}, \varepsilon, \ldots, \varepsilon),\\ &\quad\quad \ldots\\ &\quad (\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k,1}), \ldots, (\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k,m_k})\end{aligned}$$

where for all $1 \le i \le k, 1 \le j < m_i$, $\alpha_{i,j}$ is an immediate prefix of $\alpha_{i,j+1}$, and for all $1 \le i \le k$, $\alpha_{i,m_i} \equiv \alpha_i$.

Intuitively, the canonical path of $(\alpha_1, \ldots, \alpha_k)$ corresponds to the path that "completes" first α_1, then α_2 and so on. For example, for the node $(abc, 0)$ in Figure 1, the canonical path from the root passes through the nodes (ab, ε) and (abc, ε). This path is guaranteed to exist already in the DAG.

Finally, to prune a count-suffix DAG to the corresponding count-suffix tree, any edge in the DAG that is not on any canonical path is discarded. In Figure 1, the four edges marked with || are not on any canonical path and are removed to give the count-suffix tree.

As compared with the original count-suffix DAG, the count-suffix tree has the same number of nodes, but fewer edges. Because of the canonical path condition, each node, except for the root, has exactly one parent, [3] reducing the DAG into a tree.

It is important to note that even though we introduce k-D count-suffix trees as pruning the appropriate edges from the corresponding k-D count-suffix DAGs, in practice, a k-D count-suffix tree can be constructed *directly* for a given database, without explicitly constructing the DAG. Effectively, to insert any k-D string, we pick the canonical path as the path for inserting the string into the count-suffix tree.

In the sequel, we use count-suffix trees and suffix trees interchangeably, for simplicity.

3 Construction of Pruned Count-Suffix Trees

3.1 The Necessity of Pruning Nodes

A k-D count-suffix tree compresses the corresponding k-D count-suffix DAG by removing edges not on any canonical path. However, the number of nodes in both structures remain the same. It is obvious that the number of nodes is huge for large databases and for $k \ge 2$.

To be more precise, first consider a 1-D trie. Indexing N strings, each of maximum length L, requires

[2] The choice of the enumeration order turns out to be immaterial from the point of view of selectivity estimation. The only effect it has is on the actual size of the resultant count-suffix tree. Since this is a second order effect, we do not address this issue further in this paper.

[3] In the original DAG, each node may have up to k parents.

at most $N * L$ nodes, assuming no sharing. For a 1-D count-suffix tree, because of all the suffixes, the same database requires $O(N * L)$ strings, each of maximum length L. Thus, the total number of nodes is $O(N*L^2)$.

Now consider a k-D count-trie. Indexing N k-D strings, each of maximum length L, requires $O(L^k)$ possible prefixes for each k-D string, giving a total of $O(N * L^k)$ nodes in the trie. Finally for a k-D count-suffix tree, there are $O(L^k)$ possible suffixes for each k-D string. This gives a grand total of $O(N * L^{2k})$ nodes in the k-D count-suffix tree.

In summary, going from 1 to k dimensions increases the database size by only a factor of k, but it increases the size of the count-suffix tree by a factor of L^{2k-2}. Even in the 1-D case, it has been argued [9, 8] that one cannot afford to store the whole count-suffix tree for many applications and that pruning is required. In the k-D case, the need for pruning becomes even more urgent. [4]

3.2 Rules for Pruning

A tree can be pruned through the use of any well-formulated pruning rule that ensures that when a node is pruned, all its child nodes are pruned as well. In this paper, we consistently use a pruning rule that prunes a node if its count is less than a pruned count threshold $p*N$. (We will shortly be speaking of probabilities of occurrence, and will find it convenient to think of p as the pruned *probability* threshold. If N is the count at the root then, with a frequency interpretation of probability, we get $p * N$ as the corresponding count threshold). The threshold may be fixed a priori, or, for the approximate, probabilistic construction algorithms presented later, the threshold may adjust itself in order to meet given memory restrictions. Since the count associated with any node is guaranteed to be no greater than the count associated with its parent in the tree, our pruning threshold rule is well-formulated.

While the above discusses which nodes to prune, we also have a specific rule that stipulates which nodes *cannot* be pruned, regardless of their counts. These are nodes of the form $(\alpha_1, \ldots, \alpha_k)$ such that for all $1 \leq i \leq k$, the length of α_i is less than or equal to 1. Hereafter, we refer to this as the unit-cube pruning exemption rule. Note that the counts of these nodes are very likely to meet the $p * N$ threshold by themselves. But if they do not, the rule ensures that these nodes are exempted from pruning. The exemption rule is set up to facilitate the selectivity estimation algorithms presented in Section 4.

[4] Because of the dramatic increase in the size of the suffix tree, in practice given k alphanumeric attributes, it is ill advised to blindly build a k-D count-suffix tree. It is expected that some kind of analysis will be carried out, such as correlation testing, to select sub-groups of attributes to be indexed. We do not concern ourselves in this paper on how such a selection can be made.

3.3 Inadequate Ways of Creating Pruned Trees

Given the above rules for pruning, the next question is how exactly to create the pruned count-suffix tree for the given database $\mathcal{D}$. A naive way is to build the full k-D count-suffix tree, and then to apply the pruning rule. For most circumstances, this method is infeasible because the amount of intermediate storage required is tremendous.

Given memory restrictions for creating the pruned tree, we wish to be able to alternate between building and pruning on the fly. An exact strategy to do so is to first form the *completed* database, $comp(\mathcal{D})$, of the given database $\mathcal{D}$ of k-D strings. That is, for each original string $(\alpha_1, \ldots, \alpha_k)$ in $\mathcal{D}$, we form its *completed set* according to Property P1, which is the set $\{(\gamma_1, \ldots, \gamma_k) |$ for *all* (improper) suffixes γ_i of α_i for all $1 \leq i \leq k\}$. We then sort (out-of-memory) the completed database $comp(\mathcal{D})$ lexicographically according to the canonical enumeration of the dimensions. Finally, we can simply build the pruned tree by reading in sorted order, and pruning whenever the given memory is exceeded. This strategy, while exact, is in general too prohibitive in cost, because of the sorting involved on a set many times larger than the original database $\mathcal{D}$. Furthermore, as updates are made to the database, there is no obvious incremental maintenance technique.

For most applications, it may be sufficient to construct an *approximate* pruned count-suffix tree. Recently, there has been considerable research activity around the creation of synopsis data structures in a fixed amount of space [3]. In particular, based on the notion of a concise sample, which is "a uniform random sample of the data set such that values appearing more than once in the sample are represented as a value and a count" [3], Gibbons and Matias developed an incremental maintenance algorithm to maintain a concise sample. In the sequel, we refer to this as the GM algorithm.

For a given amount of working memory space, the GM algorithm gives guarantees on the probabilities of false positives and negatives. To be more precise, we wish to find all *frequent* values, i.e., values occurring at least a certain number of times in the data set. Let us use $\mathcal{F}$ to denote the set of all truly frequent values, and $\hat{\mathcal{F}}$ to denote the set of all frequent values reported based on the concise sample. The GM algorithm provides guarantees on the probability of $\alpha \notin \hat{\mathcal{F}}$ given that $\alpha \in \mathcal{F}$ (i.e., false negative), and the probability of $\alpha \in \hat{\mathcal{F}}$ given that $\alpha \notin \mathcal{F}$ (i.e., false positive) [3, Theorem 7]. Thus, one way to create an approximate pruned suffix tree for a given amount of working memory space is to apply the GM algorithm on $comp(\mathcal{D})$.

3.4 A Two-Pass Algorithm

There are, however, two problems with a direct application of the GM algorithm to our task.

Inversions: Recall that for (k-D) count-tries and count-suffix trees, the count associated with a node must not exceed the count associated with a parent. When applied to $comp(\mathcal{D})$, the GM algorithm does not make that guarantee, and it is possible that based on the concise sample, the relative ordering of the count values are reversed. In fact, it is even possible that while a certain node is reported to have a frequency exceeding a given threshold, some of its ancestors are not reported as such, i.e., node $\alpha \in \hat{\mathcal{F}}$ but some of its ancestors $\beta \notin \hat{\mathcal{F}}$.

Inaccurate counts: While the GM algorithm gives probabilistic guarantees on false positives and negatives, it does not provide guarantees on the relative errors of the reported counts (i.e., the error on C_α). As will be clear in our discussion in Section 4 on selectivity estimation, inaccurate counts in the pruned suffix tree may be compounded to give grossly inaccurate estimates for k-D strings not kept in the tree.

To deal with the above two problems, we augment the GM algorithm into the following two-pass algorithm:

1. Pass 1: Construct $comp(\mathcal{D})$ on the fly and apply the GM algorithm.
2. Pass 2: Conduct an extra pass over the original database $\mathcal{D}$ to obtain exact counts for all the strings in $comp(\hat{\mathcal{F}})$.

The second pass of the above algorithm serves two purposes. First, because counts are obtained for $comp(\hat{\mathcal{F}})$, no inversion is possible. Note that in general because of the GM algorithm, the size of $(comp(\hat{\mathcal{F}}) - \hat{\mathcal{F}})$ should not be large compared with the size of $\hat{\mathcal{F}}$. Second, the extra pass over the original database eliminates any possibility of incorrect counts due to the sampling done by the GM algorithm. If the strings in $comp(\hat{\mathcal{F}})$ can all fit in main memory (e.g., $\leq$ 1 million strings), which is achievable for many computer systems these days, the second pass amounts to a single scan of the database.

Thus, in summary, the above two-pass algorithm represents a space- and time-efficient algorithm for constructing a pruned count-suffix tree directly. It gives probabilistic guarantees on false positives and negatives (via the GM algorithm), and at the same time avoids inversions and inaccurate counts. Furthermore, to implement the unit-cube pruning exemption rule mentioned in Section 3.2, the algorithm can simply skip over the strings to be exempted in the first pass, but count them in the second pass.

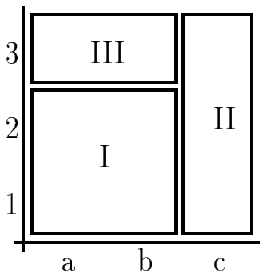

Figure 2: Example 2-D Query with GNO Estimation

When updates $\Delta\mathcal{D}$ are made to the database $\mathcal{D}$, the first pass can be performed in an incremental fashion. Only when there is a change to $\hat{\mathcal{F}}$, is there a need for a pass over $\mathcal{D} \cup \Delta\mathcal{D}$. If there is no change to $\hat{\mathcal{F}}$, then it is sufficient to perform a pass over $\Delta\mathcal{D}$ to update the counts of the existing nodes in the pruned count-suffix tree.

4 k-D Selectivity Estimation Procedures

We now come to the heart of the multi-dimensional substring selectivity estimation problem. Given a k-D query string $q = (\sigma_1, \ldots, \sigma_k)$, where for all $1 \leq i \leq k$ $\sigma_i \in \mathcal{A}_i^*$ (and can be the null string), we use the pruned count-suffix tree to give the selectivity. If q is actually kept in the pruned tree, the exact count C_q can be returned. The challenge is when q is not found, and C_q has to be estimated based on the content of the pruned tree. Below we consider two procedures to do so.

4.1 The GNO Algorithm

Given query q, the GNO (for Greedy Non-Overlap) algorithm applies greedy parsing to q to obtain non-overlapping k-D substrings of q. Before we go into the formal details of the algorithm, we give an example to illustrate the idea.

Consider the 2-D query $(abc, 123)$ shown in Figure 2. The call GNO(abc,123) first finds the longest prefix of abc from the pruned tree, and then from there the longest prefix of 123. In our example, this turns out to be the substring $(ab, 12)$ (rectangle I). Then recursive calls are made to find other substrings to complete the whole query. In our example, the recursive calls are GNO(c,123) and GNO(ab, 3). [5] And as it turns out, the substrings (c,123) (rectangle II) and (ab, 3) (rectangle III) are found in the pruned tree. Then the estimated selectivity is the product of the three selectivities.

[5] Alternatively, the recursive calls can be GNO(c, 12) and GNO(abc, 3). Regardless, in each case, the identified substrings from the pruned tree do not overlap. Experimental results for both alternatives will be presented in Section 5.

Procedure GNO$(\sigma_1, \ldots, \sigma_k)$

1. Find from the pruned tree $(\gamma_1, \ldots, \gamma_k)$ where γ_1 is the longest prefix of σ_1, and given γ_1, γ_2 is the longest prefix of σ_2, and so on.
2. $gno = C_{(\gamma_1,\ldots,\gamma_k)}/N$.
3. If $((\gamma_1, \ldots, \gamma_k)$ equal $(\sigma_1, \ldots, \sigma_k))$, return$(gno)$.
4. For $(i = 1; i \leq k;$ i++) {
   4.1 Compute δ_i such that σ_i equal $\gamma_i\delta_i$.
   4.2 If (δ_i not equal null)
   $gno = gno *$ GNO$(\gamma_1, \ldots, \gamma_{i-1}, \delta_i, \sigma_{i+1}, \ldots, \sigma_k)$.
   }
5. Return(gno).

Figure 3: Pseudo Code of Procedure GNO

Probabilistically, GNO(abc,123) is given by:

$$\begin{aligned} Pr\{(abc, 123)\} &= Pr\{(ab, 12)\} * \\ &\quad Pr\{(c, 123) \mid (ab, 12)\} * \\ &\quad Pr\{(ab, 3) \mid (ab, 12) \& (c, 123)\} \\ &\approx Pr\{(ab, 12)\} * Pr\{(c, 123)\} * \\ &\quad Pr\{(ab, 3)\} \\ &= (C_{(ab,12)}/N) * (C_{(c,123)}/N) * \\ &\quad (C_{(ab,3)}/N) \end{aligned}$$

where N is the count of the root node (i.e., the total number of strings in the database). It is essential to observe that GNO assumes conditional independence *among the substrings*. Note that this is not as simplistic as assuming conditional independence among the attributes/dimensions. For if that were the case, GNO would not have used counts like $C_{(ab,12)}$ from the pruned tree, and would have simply used counts like $C_{(ab,\varepsilon)}$ and $C_{(\varepsilon,12)}$.

A skeleton of the GNO algorithm is given in Figure 3. Step (1) can be implemented by a search of the pruned tree that finds the longest prefix in the order of the dimensions. As usual, the N in Step (2) is the count of the root node.

It should be obvious that in the worst case, GNO searches the pruned tree $O(|\sigma_1| * \ldots * |\sigma_k|)$ times. This brings us back to the unit-cube pruning exemption rule mentioned in Section 3.2. The product $|\sigma_1| * \ldots * |\sigma_k|$ gives the total number of unit-(hyper)cubes for the query. The exemption rule guarantees that the pruned tree has a count for each of the unit-cubes. Depending on the outcome of Step (1), GNO may not need any of the unit-cubes. Strictly speaking, we can do away with the exemption rule, and if a unit-cube is needed but is not found in the pruned tree, we can simply use the prune probability p. We prefer to adopt the exemption rule because in this way, the selectivity of the unit-cube is the most accurate. This accuracy is particularly significant when the actual selectivity is much lower than p, such as for the so-called "negative" queries considered in Section 5.

In terms of formal properties of GNO, the following theorem shows that GNO generalizes the KVI algorithm proposed in [9] and analyzed in [8] for 1-D substring selectivity estimation. In a nutshell, given a 1-D query string σ, KVI(σ) finds the longest prefix γ from the pruned tree, and then makes the recursive call KVI(δ) where $\sigma = \gamma\delta$.

Given a k-D pruned count-suffix tree $\mathcal{T}$, we use the notation $proj(\mathcal{T}, i)$, for some $1 \leq i \leq k$, to denote the subtree of $\mathcal{T}$ such that:

- the set of nodes is given by: $\{\alpha_i \mid$ the node $(\varepsilon, \ldots, \varepsilon, \alpha_i, \varepsilon, \ldots, \varepsilon)$ is in $\mathcal{T}\}$, where α_i can be the null string ε; and
- the set of edges is given by the set of edges in $\mathcal{T}$ connecting only nodes of the form $(\varepsilon, \ldots, \varepsilon, \alpha_i, \varepsilon, \ldots, \varepsilon)$.

For example, the tree shown in Figure 1, when projected on the first dimension, consists of the root node and (ab, ε), (abc, ε) and (abd, ε), and the edges connecting these nodes.

Theorem 4.1 *For any k-D pruned tree $\mathcal{T}$, and k-D query $q = (\varepsilon, \ldots, \varepsilon, \sigma_i, \varepsilon, \ldots, \varepsilon)$, the estimate given by GNO for q using $\mathcal{T}$ is identical to the estimate given by the KVI algorithm for σ_i using $proj(\mathcal{T}, i)$.* ■

4.2 The MO Algorithm: Example

Recall that GNO assumes conditional independence among the substrings. However, it has been observed that complex sequences typically exhibit the following statistical property, called the *short memory property*: if we consider the (empirical) probability distribution on the next symbol a given the preceding subsequence α of some given length, then there exists a length L (the memory length) such that the conditional probability does not change substantially if we condition it on preceding subsequences of length greater than L. Such an observation led Shannon, in his seminal paper [16], to suggest modeling such sequences by Markov chains.

Having said that, we do not intend to determine this magic length L. We believe that determining L is not practical, especially in the presence of updates. However, this points to the fact that there is room for improved estimation accuracy if the overlaps among substrings are taken into consideration. And it is in this aspect that MO tries to excel.

To first illustrate the idea of the MO estimation algorithm, consider again the 2-D query $(abc, 123)$ shown in Figure 2. While GNO finds three 2-D non-overlapping substrings, MO finds overlapping substrings. In Figure 4, to highlight the comparison between MO and GNO, we assume that MO also finds three substrings, corresponding to the ones shown in

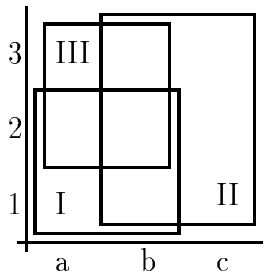

Figure 4: Example 2-D Query with MO Estimation

Figure 2. (In general, MO may find a lot more k-D maximal substrings, i.e., k-D substrings α, β such that α is not a substring of β and vice versa.) While the substring $(ab, 12)$ (rectangle I) remains the same, MO now finds $(bc, 123)$ (rectangle II) and $(ab, 23)$ (rectangle III).

The question now is how to "combine" all these substrings together. Let us begin by considering $(ab, 12)$ and $(ab, 23)$. Probabilistically, we have:

$$\begin{aligned} Pr\{(ab,123)\} &= Pr\{(ab,12)\} * \\ &\quad Pr\{(ab,3) \mid (ab,12)\} \\ &\approx Pr\{(ab,12)\} * \\ &\quad Pr\{(ab,3) \mid (ab,2)\} \\ &= Pr\{(ab,12)\} * \\ &\quad Pr\{(ab,23)\}/Pr\{(ab,2)\} \end{aligned}$$

Thus, unlike GNO, MO does not assume complete conditional independence among the substrings. Whenever possible, it allows conditioning up to the overlapping substring (e.g., $(ab, 2)$) of the initial substrings under consideration (e.g., $(ab, 12)$ and $(ab, 23)$ here).

Operationally, we can view the above probabilistic argument as a counting exercise. When we take the product of $Pr\{(ab, 12)\}$ and $Pr\{(ab, 23)\}$, we are basically counting rectangles I and III in Figure 4. The problem is that we have "double" counted the rectangle corresponding to substring $(ab, 2)$. To compensate, we divide the product with $Pr\{(ab, 2)\}$.

To continue now by taking into consideration rectangle II, we take the product of the probabilities $Pr\{(ab, 12)\}$, $Pr\{(bc, 123)\}$ and $Pr\{(ab, 23)\}$, basically counting all three rectangles. To compensate for double counting, we divide the product by the three 2-way intersections: (i) $Pr\{(b, 12)\}$ between I and II; (ii) $Pr\{(ab, 2)\}$ between I and III; and (iii) $Pr\{(b, 23)\}$ between II and III.

However, by dividing by the 2-way intersections, we have "over compensated". Specifically, the substring $(b, 2)$ is initially counted three times in the product, but is then dis-counted three times in the division of the three 2-way intersections. To make up, we need to multiply what we have so far with $Pr\{(b, 2)\}$, which

Procedure MO$(\sigma_1, \ldots, \sigma_k)$

1. Find from the pruned tree all the maximal k-D substrings of $(\sigma_1, \ldots, \sigma_k)$. Let these be $\lambda_1, \ldots, \lambda_u$ for some u.
2. Initialize multiset S to $\{\{(\lambda_1, 1, 1), \ldots, (\lambda_u, u, 1)\}\}$, and i to 1.
3. Repeat {
   3.1 Initialize multiset S_{new} to $\emptyset$.
   3.2 For all $(\alpha, v, w) \in S$ such that w equal i
   For all $v < j \leq u$ {
   If ($\alpha \cap \lambda_j$ non-empty)
   add $(\alpha \cap \lambda_j,\ j,\ i+1)$ to S_{new}.
   }
   3.3 $S = S \sqcup S_{new}$, and $i{+}{+}$
   } until (S_{new} equal $\emptyset$)
4. Initialize mo to 1.
5. For all $(\alpha, v, w) \in S$ {
   5.1 Get count C_α from the pruned tree.
   5.2 If (w is an odd integer), $mo = mo * (C_\alpha/N)$
   Else $mo = mo/(C_\alpha/N)$
   }
6. Return(mo).

Figure 5: Pseudo Code of Procedure MO

is the 3-way intersection between the three initial substrings.

4.3 The MO Algorithm: Pseudo Code

The counting exercise illustrated in the above example is generalized in Figure 5, which gives a skeleton of the MO algorithm. Step (1) first finds all the maximal k-D substrings of the query q from the pruned tree. Let these be $\lambda_1, \ldots, \lambda_u$ for some u. Then Steps (2) to (3) find all the non-empty 2-way intersections (i.e., $\lambda_i \cap \lambda_j$), 3-way intersections (i.e., $\lambda_i \cap \lambda_j \cap \lambda_l$), and so on, up to w-way intersections for $w \leq u$. A triple $(\alpha, v, w) \in S$ means that α is a w-way intersection, and λ_v is the highest indexed λ_i participating in this intersection. When computing the $w+1$-way intersections using (α, v, w), the condition "for all $v < j \leq u$" in Step (3.2) ensures that the same λ_j does not participate more than once in the intersection. Note that S and S_{new} have to be multisets, not sets, since duplicate occurrences of the k-D substrings among the λ_i and the various intersections need to be preserved for correctness.

After all the possible intersections among $\lambda_1, \ldots, \lambda_u$ are found, Step (5) of MO computes the final estimate. It obtains the appropriate counts from the pruned count-suffix tree. Note that the suffix tree guarantees that if there are nodes corresponding to α and λ_j, then their non-empty intersection $\alpha \cap \lambda_j$ must have a corresponding node in the tree. Thus, for any (α, v, w) in S, the count C_α can always be obtained from the tree in Step (5.1). Finally, Step (5.2) puts the probability (C_α/N) in the numerator or the denominator

depending on whether w is odd or even. That is, if α is a w-way intersection among $\lambda_1, \ldots, \lambda_u$, and w is odd, then the probability appears in the numerator, else in the denominator.

4.4 The MO Algorithm: Properties

A natural question to ask at this point is whether Step (5.2) is "correct". As motivated in the example shown in Figure 4, by "correct", we mean that each substring of query q is counted *exactly* once, i.e., neither over-counting nor over-discounting. We offer the following lemma.

Lemma 4.1 *For any $(\alpha, \_, w)$ in S, representing a w-way intersection, Step (5.2) of MO is correct in that each k-D substring α is counted exactly once.*

Proof sketch. For any w-way intersection α, let us assume, without loss of generality, that α is the intersection of $\lambda_1, \ldots, \lambda_w$. Then: α must have been counted $\binom{w}{1}$ times initially, then dis-counted $\binom{w}{2}$ times due to 2-way intersections, then counted $\binom{w}{3}$ times due to 3-way intersections, and so on. So the total number of times α has been counted and discounted is: $\binom{w}{1} - \binom{w}{2} + \binom{w}{3} - \ldots - (-1)^w \binom{w}{w}$. This can be rewritten as: $(-\sum_{j=1}^{w} (-1)^j \binom{w}{j})$. Now consider the well-known binomial expansion $(1-x)^w = (1 + \sum_{j=1}^{w} (-1)^j \binom{w}{j} x^j)$. By substituting $x = 1$, we get $0 = (1-1)^w = 1 + \sum_{j=1}^{w} (-1)^j \binom{w}{j})$. Hence, $(-\sum_{j=1}^{w} (-1)^j \binom{w}{j}) = 1$. ■

In [8], for 1-D substring selectivity estimation, we presented a 1-D version of MO. Our analysis indicates that the 1-D version enjoys certain desirable properties and forms the basis for obtaining even more accurate selectivity estimations for 1-D substrings. Thus, to allow all those to carry over, it is important that the k-D MO presented here generalizes the 1-D MO analyzed there. Partly to avoid excessive details, and partly to illustrate the complication in generalizing from 1-D to k-D, we resort to the following example.

Suppose for the query *abcde*, 1-D MO finds three maximal substrings: *abc*, *bcd*, and *cde*. Then 1-D MO, as presented in [8], gives the following estimate:

$$Pr\{abcde\} \approx \frac{C_{abc}}{N} * \frac{C_{bcd}}{C_{bc}} * \frac{C_{cde}}{C_{cd}}$$

On the other hand, the k-D MO procedure shown in Figure 5 gives the following estimate for $Pr\{abcde\}$:

$$\frac{(C_{abc}/N) * (C_{bcd}/N) * (C_{cde}/N) * (C_c/N)}{(C_{bc}/N) * (C_{cd}/N) * (C_c/N)}$$

While it is easy to see that both estimates are identical, we must point out two more subtle details:

- In the k-D MO calculation above, there are terms that cancel off each other, notably (C_c/N). While the (C_c/N) term in the numerator corresponds to the 3-way intersection between the three maximal substrings, the (C_c/N) term in the denominator corresponds to the 2-way intersection between *abc* and *cde*. The point here is that the 3-way intersection of *abc*, *bcd*, and *cde* is exactly the 2-way intersection of the first and the last ones.

- The use of the words "first" and "last" precisely underscore the fact that in 1-D, all the maximal substrings can be *linearly ordered* with respect to the query q. Then it is unnecessary to consider any w-way intersections for $w \geq 3$, and even unnecessary to consider the 2-way intersection between λ_i and λ_j for $j > i+1$. In other words, it is sufficient to just consider 2-way intersections of two successive maximal substrings (e.g., the intersection *bc* between *abc* and *bcd*). The complication in k-D is that there is no linear order to fall back on; λ_i may "precede" λ_j in some dimensions, but vice versa for the other dimensions.

The following results establish that k-D MO is a proper generalization of 1-D MO.

Theorem 4.2 *For any k-D pruned tree $\mathcal{T}$, and k-D query $q = (\varepsilon, \ldots, \varepsilon, \sigma_i, \varepsilon, \ldots, \varepsilon)$, the estimate given by the MO algorithm shown in Figure 5 for q using $\mathcal{T}$ is identical to the estimate given by the 1-D MO algorithm presented in [8] for σ_i using $proj(\mathcal{T}, i)$.* ■

When the underlying dimensions are independent of each other, the above theorem can be generalized to the following result.

Theorem 4.3 *Suppose the k dimensions are independent of each other, i.e., for all nodes $(\alpha_1, \ldots, \alpha_k)$ in the suffix tree $\mathcal{T}$, $C_{(\alpha_1,\ldots,\alpha_k)}/N = \Pi_{i=1}^{k}(C_{(\varepsilon,\ldots,\alpha_i,\ldots,\varepsilon)}/N)$. Then for any k-D pruned tree $\mathcal{T}'$ of $\mathcal{T}$, and k-D query $q = (\sigma_1, \ldots, \sigma_k)$, the estimate given by k-D MO for q using $\mathcal{T}'$ is equal to the product of the estimates given by 1-D MO for σ_i using $proj(\mathcal{T}', i), 1 \leq i \leq k$.* ■

Last but not least, let us analyze the complexity of the *MO* algorithm. There are $O(|\sigma_i|^2)$ possible substrings of the 1-D string σ_i. Thus, there are $O(|\sigma_1|^2 * \ldots * |\sigma_k|^2)$ k-D substrings of the k-D string $(\sigma_1, \ldots \sigma_k)$. We check each for presence and maximality in the given pruned count-suffix tree. Hence, in the worst case, Step (1) requires $O(|\sigma_1|^2 * \ldots * |\sigma_k|^2)$ searches of the pruned tree. Step (5) may need another $O(2^u)$ searches of the tree, since in the worst case set S computed in Step (3) may be of size $O(2^u)$. Thus, in terms of worst case complexity, MO is far inferior to GNO. The practical questions, however, are: how much more absolute time is required by MO, and

whether the extra runtime gives better accuracy in return. We rely on experimentation to shed light on these questions.

5 Experimental Evaluation

5.1 Experimental Setup

We implemented the algorithms presented in this paper. They were written in C. We paid special attention to ensure that MO is not affected by roundoff errors. Below we report some of the experimental results we collected. The reported results were obtained using a real AT&T data set containing office information about most of the employees. In particular, the reported results are based on two attributes: the last name and the office phone number of each employee. For these two attributes, the un-pruned 2-D count-suffix tree has 5 million nodes. The results reported here are based on a pruned tree that keeps the top 1% of the nodes (i.e., 50,000 nodes) with the highest counts.

Following the methodology used in [9, 8], we considered both "positive" and "negative" queries, and used relative error as one of the metrics for measuring accuracies. Positive queries are 2-D strings that were present in the un-pruned tree or in the database, but that were pruned. We further divided positive queries into different categories depending on how close their actual counts were to the pruned count. Below we use Pos-Hi, Pos-Med, and Pos-Lo to refer to the sets of positive queries whose actual counts were 36, 20 and 4 respectively, where the pruned count was 40. Each of the three sets above consists of 10 randomly picked positive queries. Those were picked to cover different parts of the pruned tree.

To measure the estimation accuracy of positive queries, we give the average relative error over the 10 queries in the set, i.e., (estimated count − actual count)/actual count. Thus, relative error ranges from −100% to infinity theoretically. Because relative error tends to favor under-estimation to over-estimation, we adjust an over-estimated count by the prune count, whenever the former is greater than the latter, i.e., (min(estimated count,prune count) − actual count)/actual count.

While relative error measures accuracy in relative terms, mean squared error, i.e., (estimated count − actual count)^2, measures accuracy in absolute terms. For some of the cases below, we give the square root of the average mean squared error for positive queries. We refer to this as the average mean standard error.

Negative queries are 2-D strings that were not in the database or in the un-pruned tree. That is, if the un-pruned tree were available, the correct count to return for such a query would be 0. To avoid division by 0, estimation accuracy for negative queries is measured using mean standard error as the metric.

	Pos-Hi	Pos-Med	Pos-Lo
MO	(+4%,3.89)	(+16%,10.35)	(-11%,3.38)
GNO	(-98%,35.3)	(-95%,19.13)	(-90%,3.99)

Figure 6: Estimation Accuracy for Positive Queries

5.2 MO versus GNO: Positive Queries

The table in Figure 6 compares the estimation accuracy between MO and GNO. Each entry in the table is a pair, where the first number gives the average relative error, and the second number gives the average mean standard error. For example, the first pair (-98%, 35.3) for GNO indicates that GNO underestimates by a wide margin, and for a "typical" positive query of actual count being 36, GNO estimates the count to be 36 - 35.3 = 0.7. In contrast, MO gives a very impressive average relative error of 4%, and for a "typical" positive query of actual count being 36, MO estimates the count to be 36 + 3.89 = 39.89.

As the actual counts of the positive queries drop, GNO gradually gives better results. This is simply because GNO always under-estimates, but the under-estimation becomes less serious as the actual counts themselves become smaller. On the other hand, no such trend can be said about MO. Sometimes it under-estimates, and other times it over-estimates. But there cannot be any doubt that MO is the winner.

In Section 4.1, we point out that there are many different combinations to make the recursive calls in Step (4.2) of GNO. For 2-D, there are two ways. Besides the version of GNO as shown in Figure 3, we also implemented and experimented with the other version. In general, there are some slight differences in the estimations. But in terms of accuracy, the other version remains as poor.

5.3 MO versus GNO: Negative Queries and Runtime

The mean standard error for negative queries (average over 10 randomly picked ones) is 0.002 for GNO and 0.01 for MO. While GNO is more accurate for negative queries than MO, the accuracy offered by MO is more than acceptable.

By now it is clear that MO offers significantly more accurate estimates than does GNO. The only remaining question is whether MO takes significantly longer to compute than does GNO. For our three sets of positive queries, MO often finds 12–16 maximal 2-D substrings, whereas GNO uses only 3–5 substrings. Consequently, while GNO takes $O(10^{-6})$ seconds to compute, MO usually takes $O(10^{-4})$ seconds (on a 225 MHz machine). Nonetheless, we believe that the extra effort is worthwhile.

	Pos-Hi	Pos-Med	Pos-Lo	Negative
Indep	-23%	-17%	-27%	0.25
MO	+4%	+16%	-11%	0.01

Figure 7: Estimation Accuracy: the Independence Assumption

5.4 MO versus Two 1-D Exact Selectivities

The next question we explore experimentally is as follows. Since we know that a 2-D count-suffix tree is much larger than two 1-D count-suffix trees (i.e., like comparing the product with the sum), there is always the question of: *given the same amount of memory, and in the presence of pruning, would direct 2-D selectivity estimation give more accurate results than using the product of the two 1-D selectivities?* Because it is difficult to get two equal-sized pruned setting, we did the following:

- On the one hand, we used MO on the 2-D pruned tree we have been using so far. This has 50,000 nodes for a total size of 650 Kbytes.
- On the other hand, we used two *un-pruned* 1-D count-suffix trees. In sum, the two trees have more than 160,000 nodes for a total size of 2.3 Mbytes.

Thus, for the latter setting, we used exact 1-D selectivities, without any estimation involved. Essentially, this is an exercise of comparing MO with applying the independence assumption to k-D selectivity estimation. We gave the independence assumption an unfair advantage over MO by allowing the former three times as much space.

Yet, Figure 7 shows that MO compares favorably for both positive and negative queries. For positive queries, the figure only gives the average relative error; and for negative queries, the figure gives the average mean standard error. For easier comparison, the results of MO are repeated in the figure from earlier discussion.

Despite the fact that exact 1-D selectivities are used, and that more space is given to the independence assumption approach, the approach gives less accurate results than 2-D MO. In particular, for negative queries, 2-D MO appears to be far superior. We can attribute this to the unit-cube pruning exemption rule.

The outcome of this comparison is actually somewhat surprising. Initially we expected that the last name attribute of AT&T employees would be quite independent of their office phone numbers. (For instance, office phone numbers and office fax numbers would be far more correlated.) Yet, using MO still gives better results than relying on the independence assumption.

	MO	Indep	GNO
relative error	33%	-57%	-99%

Figure 8: Estimation Accuracy for Large Area Positive Queries

5.5 Accuracy for Large Area Positive Queries

So far, all the positive queries used are "small area", by which we mean that the "area" (i.e., $|\sigma_1| * |\sigma_2|$) covered by $q = (\sigma_1, \sigma_2)$ is between 5 and 12. 2-D strings corresponding to a smaller area tend to be always kept in the pruned tree. Figure 8 shows results for positive queries with "large areas", which is defined to be ≥ 18.

Compared with the small area positive queries, MO becomes less accurate for large area positive queries. One possible explanation is as follows. The larger the area covered by a query, the greater the number of maximal substrings found. Thus, in finding all w-way intersections, w tends to become a larger number than before. Apparently, inaccuracies incurred in the earlier counts are compounded to give a less accurate final estimate. Nonetheless, as compared with the other alternatives, MO is still the best. Finding a way to improve accuracy on large area positive queries is an interesting open problem.

6 Conclusions and Future Work

Queries involving wildcard string matches in multiple dimensions are becoming more important with the growing importance of LDAP directories, XML and other text-based information sources. Effective query optimization in this context requires good multi-dimensional substring selectivity estimates.

We demonstrated, using a real data set, that assuming independence between dimensions can lead to very poor substring selectivity estimates. This argues for the need to develop compact k-D data structures that can capture the correlations between strings in multiple dimensions, and accurate estimation algorithms that can take advantage of such data structures. In this paper, we presented a k-D extension of the pruned count-suffix tree, and described a space- and time-efficient algorithm for the direct construction of the pruned tree, that provides quality guarantees. We formulated an estimation algorithm, MO, that uses all maximal multi-dimensional substrings of the query for estimation; these multi-dimensional substrings help to capture the correlation that may exist between strings in the multiple dimensions. We showed analytically that MO has certain desirable properties, and established empirically the utility of MO for multi-dimensional substring selectivity estimation.

One very interesting open problem is as follows. Given k alphanumeric attributes in the database, the

optimization problem is to determine the set of pruned count-suffix trees (possibly with different dimensionalities, possibly with overlapping dimensions) that together satisfy a space constraint, and minimizes some error metric. As far as selectivity estimation is concerned, an interesting open problem is the development of more accurate algorithms for the so-called "large area" queries. One possibility is to use constraints that relate the count of one node to the counts of other nodes in a count-suffix tree. Some of those constraints have been applied to the 1-D substring selectivity estimation problem, and have shown to be able to give more accurate estimates [8]. It would be interesting to see what roles constraints can play in multi-dimensional substring selectivity estimation.

Acknowledgements

We would like to thank Nick Koudas and the anonymous reviewers of the paper, for their suggestions that helped improve the content of the paper.

References

[1] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM Journal on Computing*, 24(3):520–562, 1995.

[2] R. Giancarlo and R. Grossi. On the construction of classes of suffix trees for square matrices: Algorithms and applications. *Information and Computation*, 130(2):151–182, 1996.

[3] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 331–342, 1998.

[4] T. Howes and M. Smith. *LDAP: Programming directory-enabled applications with lightweight directory access protocol*. Macmillan Technical Publishing, Indianapolis, Indiana, 1997.

[5] Y. Ioannidis. Universality of serial histograms. In *Proceedings of the International Conference on Very Large Databases*, pages 256–267, 1993.

[6] Y. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 233–244, 1995.

[7] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proceedings of the International Conference on Very Large Databases*, pages 275–286, 1998.

[8] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, PA, June 1999.

[9] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 282–293, 1996.

[10] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, March 1990.

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[12] M. Muralikrishna and D. Dewitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 28–36, 1988.

[13] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 294–305, 1996.

[14] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the International Conference on Very Large Databases*, pages 486–495, 1997.

[15] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1979.

[16] C. E. Shannon. Prediction and entropy of printed english. *Bell systems technical journal*, 30(1):50–64, 1951.

[17] M. Wang, J. S. Vitter, and B. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 169–180, 1997.

[18] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.