The VLDB Journal (2000) 9: ♣–♣

The VLDB Journal
© Springer-Verlag 2000

One-dimensional and multi-dimensional substring selectivity estimation

H.V. Jagadish[1], **Olga Kapitskaia**[2], **Raymond T. Ng**[3], **Divesh Srivastava**[4]

[1] University of Michigan, Ann Arbor; E-mail: jag@umich.edu
[2] Pôle Universitaire Léonard de Vinci; E-mail: Olga.Kapitskaia@devinci.fr
[3] University of British Columbia; E-mail: rng@cs.ubc.ca
[4] AT&T Labs – Research, 180 Park Avenue, Bldg 103, Florham Park, NJ 07932, USA; E-mail: divesh@research.att.com

Edited by M.P. Atkinson. Received: April 28, 2000 / Accepted: July 11, 2000

Abstract. With the increasing importance of XML, LDAP directories, and text-based information sources on the Internet, 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 multiple dimensions. Effective query optimization in this context requires good selectivity estimates. In this paper, we use pruned count-suffix trees (PSTs) as the basic data structure for substring selectivity estimation. For the 1-D problem, we present a novel technique called MO (Maximal Overlap). We then develop and analyze two 1-D estimation algorithms, MOC and MOLC, based on MO and a constraint-based characterization of all possible completions of a given PST. For the k-D problem, we first generalize PSTs to multiple dimensions and develop a space- and time-efficient probabilistic algorithm to construct k-D PSTs directly. We then show how to extend MO to multiple dimensions. Finally, we demonstrate, both analytically and experimentally, that MO is both practical and substantially superior to competing algorithms.

Key words: String selectivity – Maximal overlap – Short memory property – Pruned count-suffix tree

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%`). Such predicates are pervasive in data warehouse queries, because of the presence of "unclean" data [HS95]. With the growing importance of XML, LDAP directories, and other text-based information stores on the Internet, substring queries are becoming increasingly common.

Furthermore, in many situations with these applications, a query may specify substrings to be matched on multiple alphanumeric attributes or dimensions. The query [(`name` *like* `%jones%`) AND (`tel` *like* `973360%`) AND (`mail` *like* `%research.att.com`)] is one example. Often the attributes mentioned in these kinds of multi-dimensional queries may be correlated. For the above example, because of the geographical location of the research labs, people that satisfy the query (`mail` *like* `%research.att.com`) may have an unexpectedly high probability to satisfy the query (`tel` *like* `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 lead to gross inaccuracy.

1.1 The data structure

A natural question that arises is which data structure does one use for substring selectivity estimation. Histograms have long been used for selectivity estimation in databases (see, e.g.,[SAC+79,MD88,LN90,Ioa93,IP95,PIHS96,JKM+98]). They have been designed to work well for numeric attribute value domains. For the string domain, one could continue to use such "value-range" histograms by sorting substrings based on the lexicographic order and computing 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. As a result, we look for a different solution, one that is suitable for the string domain.

A commonly used data structure for indexing substrings in a database is the suffix tree [Wei73,McC76], which is a trie that satisfies the following property: whenever a string α is stored in the trie, then all suffixes of α are stored in the trie as well. Given a substring query, one can locate all the desired matches using the suffix tree. Krishnan et al. [KVI96] proposed an interesting variation of the suffix tree: the *pruned count-suffix tree* (PST), which maintains a count, C_α, for each substring α in the tree and retains only those substrings α (and their counts) for which C_α exceeds some pruning threshold. In this paper, following [KVI96], we use PSTs as the basic summary data structure for substring selectivity estimation.

To estimate substring selectivity in multiple dimensions, we need to generalize the PST to multiple dimensions. Here, only those k-D substrings $(\alpha_1, \ldots, \alpha_k)$ for which the count exceeds the pruning threshold are maintained in the tree (along with their counts).

1.2 The problem

The *substring selectivity estimation* problem can be formally stated as follows:

> Given a pruned count-suffix tree $\mathcal{T}$, and a (1-D or k-D) substring query q, estimate the fraction C_q/N, where N is the count associated with the root of $\mathcal{T}$.

The 1-D version of the above problem considers the situation when the pruned tree $\mathcal{T}$ is created for a single attribute (e.g., `name`). The k-D version of the problem considers the case when $\mathcal{T}$ is set up for multiple attributes (e.g., `name` and `tel`).

What we gain in space by pruning a count-suffix tree, we lose in accuracy in the estimation of 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.

1.3 Our contributions

We begin by describing the 1-D problem and its solution first (in Sects. 4–6), and then go on to generalize our results to multiple dimensions (in Sects. 7–10). Specifically, we make the following contributions:

- In Sect. 4, for the 1-D problem, we present a novel selectivity estimation algorithm MO (Maximal Overlap), which estimates the selectivity of the query string σ, based on all maximal substrings, β_i, of σ in the 1-D PST. We demonstrate that MO is provably better than KVI, the independence-based estimation technique developed in [KVI96], using a greedy parsing of σ, under the natural assumption that strings exhibit the so-called short memory property. We also experimentally show that MO is substantially superior to KVI in the quality of the estimate, using a real AT&T data set.
- In Sects. 5 and 6, we develop constraint-based characterizations of all count-suffix trees that are possible *completions* of a given PST. Based on a sound approximation of this constraint-based characterization, we develop and analyze two selectivity estimation algorithms, MOC (Maximal Overlap with Constraints) and MOLC (Maximal Overlap on Lattice with Constraints). In Sect. 6.4, we show that KVI, MO, MOC and MOLC illustrate an interesting tradeoff between estimation accuracy and computational efficiency.
- Turning from the 1-D to the k-D problem, in Sect. 7, we propose a novel k-D generalization of 1-D PSTs, as the basic data structure for solving the k-D problem. Given the enormous sizes of count-suffix trees for large databases, and especially for multiple dimensions, it is essential to obtain PSTs within given memory restrictions. In Sect. 8, we develop a space- and time-efficient probabilistic algorithm to construct a PST without first having to construct the full count-suffix tree.
- In Sect. 9, we develop and analyze two algorithms for multi-dimensional substring selectivity estimation. The first algorithm, called GNO (Greedy Non-Overlap), uses greedy parsing of the k-D query string and generalizes algorithm KVI for the 1-D problem. The second algorithm generalizes algorithm MO from 1-D to k-D and uses all maximal k-D substrings of the query for estimation to take advantage of correlations that may exist between the strings in multiple dimensions.
- In Sect. 10, we compare the accuracy of our two algorithms, GNO and MO and additionally compare them with the default assumption of attribute independence, using a real AT&T 2-D data set. Our results again 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.

2 Related work

Histograms have long been used for selectivity estimation in databases (see, e.g., [SAC$^+$79,MD88,LN90,Ioa93,IP95, PIHS96,JKM$^+$98]), and one can obtain good solutions to the histogram construction problem using known techniques (see, e.g., [PIHS96,JKM$^+$98]). However, as mentioned earlier, conventional histograms have been designed to work well for numeric attribute value domains and do not yield good results for the string domain.

End-biased histograms [IP95] are more closely related to PSTs. The high-frequency values in the end-biased histogram correspond to nodes that are retained in the PST. 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. As expected, this yields poor estimates for substring selectivity.

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 [PI97]. But for the reasons given earlier, this study is not directly applicable to the problem of substring selectivity estimation in multiple dimensions.

The 1-D suffix tree [Wei73,McC76] is a commonly used structure for indexing substrings in a database. 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 submatrices in a database of such matrices (see, e.g., [Gia95, GG96]). The problem of indexing submatrices is clearly a different problem than indexing substrings in multiple correlated dimensions, and the suffix tree developed for the submatrix matching problem does not seem applicable to our problem.

Our problem, despite its importance, appears to have received much less attention in the literature. Notable exceptions are a study of 1-D substring selectivity estimation, presented in [KVI96], and a study of k-D substring selectivity estimation, given in [WVI97]. There are some similarities and several key differences between the study of 1-D and k-D substring selectivity estimation presented in [KVI96,WVI97] and the work presented here:

- First, at a data-structure level, both the 1-D substring selectivity estimation in [KVI96] and our work are based on

PSTs; in fact, 1-D PSTs were proposed in [KVI96]. However, the k-D substring selectivity estimation in [WVI97] is based on k separate 1-D PSTs and a multi-dimensional array. In our case, the estimation is based on a k-D PST, proposed in this paper.

– Second, for constructing pruned data structures without constructing the complete count-suffix trees, only ad hoc heuristics were considered in [KVI96,WVI97], i.e., no quality guarantees were provided. Our approach of direct construction of 1-D and k-D PSTs builds upon the concise sampling technique proposed in [GM98], provides probabilistic guarantees on the number of false positives and false negatives, and gives *accurate* counts for the substrings in the PST.
– Third, for 1-D selectivity estimation, experimental evaluation of various independence-based, child-based and depth-based strategies is provided in [KVI96]. 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; no formal analysis is given in [KVI96].
  For k-D selectivity estimation, a generalization of the KVI algorithm, as well as child-based and depth-based strategies, has been developed in [WVI97]. That generalization does greedy parsing *independently* in each of the k dimensions, using 1-D PSTs, 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 in this paper. As will be shown later, the MO algorithm proposed here is superior to the GNO algorithm for multiple dimensions.

Finally, parts of this paper have appeared in [JNS99] and in [JKNS99].

3 Background and notation

Throughout this paper, we use $\mathcal{A}$ to denote the alphabet; a, b, possibly with subscripts, to denote single characters in $\mathcal{A}$; and Greek lower-case symbols $\alpha, \beta, \gamma, \sigma$, possibly with subscripts, to denote strings of arbitrary (finite) length in $\mathcal{A}^*$. For simplicity, we do not distinguish between a character in $\mathcal{A}$, and a string of length 1.

3.1 Suffix trees

A *suffix tree* [Wei73,McC76] is a trie that stores not only the given database of strings $\mathcal{D} = \{\gamma_1, \ldots, \gamma_n\}$, but also all suffixes of each γ_i. A *count-suffix tree* is a variant of the suffix tree which does not store pointers to occurrences of the substrings α of the γ_i's, but just keeps a count C_α at the node corresponding to α in the tree.

The count C_α can have (at least) two useful meanings in the count-suffix tree. First, it can denote the number of strings in the database $\mathcal{D}$ containing α as a substring. Second, it can denote the number of occurrences of α as a substring in the database $\mathcal{D}$. Suppose $\mathcal{D}$ contains only the string `banana`. With the first interpretation, $C_{\texttt{ana}}$ would be 1, but with the second interpretation, $C_{\texttt{ana}}$ would be 2. Both interpretations are obviously useful in different applications. We differentiate between count-suffix trees, depending on the interpretation of C_α, as follows:

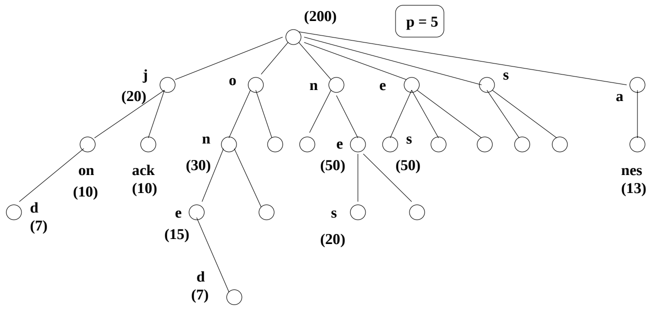

Fig. 1. Example PST

Definition 1 (p- and o-suffix trees) *A* p-suffix tree *is a count-suffix tree, where non-negative integer C_α denotes the number of strings in the database $\mathcal{D}$ containing α as a substring.*

An o-suffix tree *is a count-suffix tree, where non-negative integer C_α denotes the number of occurrences of α as a substring in the database $\mathcal{D}$.* □

Krishnan et al. [KVI96] considered only p-suffix trees, due to their utility for query selectivity estimation. In this paper, we consider both p- and o-suffix trees. Where the distinction does not matter, we simply refer to them as count-suffix trees.

In the following, we use N to denote the count associated with the root of a count-suffix tree. Specifically, for the p-suffix tree, N denotes the number of strings in $\mathcal{D}$, whereas for the o-suffix tree, N denotes the total number of suffixes of strings in $\mathcal{D}$.

The storage requirement of a full count-suffix tree can be prohibitive. When one wishes to obtain only a quick estimate of the counts, it suffices to store a PST [KVI96]. We use $\mathcal{T}$ to denote both pruned p- and o-suffix trees. Pruning is done based on some *pruning rule*. For instance, one could choose to retain only the top k levels of the count-suffix tree. A more adaptive rule is to prune away every node α that has a count $C_\alpha \leq p$, where p is the pruning threshold. We say that a pruning rule is *well formulated* if it prunes every descendant of α when it prunes α. Both pruning rules described above are well formulated. We use the threshold-based pruning rule in this paper for consistency with [KVI96], even though our results apply to other well-formulated pruning rules, such as the level-based pruning rule, with appropriate obvious modifications.

We illustrate an example PST, with pruning threshold $p = 5$, in Fig. 1. Labels are presented for substrings related to the database string `jones`, with counts C_α shown in parentheses for some of the nodes in the PST.

Definition 2 (Completion of a PST) *We say that a count-suffix tree is a* completion *of a PST $\mathcal{T}$ if $\mathcal{T}$ can be obtained by pruning the count-suffix tree.*

Observe that it is possible for the same PST to be generated by pruning many different count-suffix trees. We use $\mathcal{C}(\mathcal{T})$ to denote the set of all completions of PST $\mathcal{T}$. □

3.2 Types of queries supported

Clearly, a PST can be used to estimate the selectivity of substring queries. For example, the query ($\texttt{attr1} = *jone*$) is matched by the node $\alpha = (jone)$ or even $\alpha = (jones)$. In addition to substring queries, queries of the following forms are also prevalent:

– `attr1` beginning with $jone$ (i.e., prefix match),
– `attr1` ending with $jone$ (i.e., suffix match), and
– `attr1` matching $jone$ (i.e., exact match).

Even though in the above example the node $(jone)$ appears to handle the prefix query "`attr1` beginning with $jone$", it really does not. The reason is that if there is, for example, a string $cjone$ in the database then this string alone accounts for one occurrence of $jone$, which is a suffix of $cjone$, in the PST. In other words, the count associated with the node $(jone)$ includes not only strings with $jone$ as the prefix, but indeed all strings with $jone$ as a substring.

It turns out that a simple trick is sufficient to make the PST capable of handling all the variations mentioned above. For each string, we add two special characters: # attached to the beginning and $ appended to the end of the string. As far as insertion into the PST is concerned, these two special characters behave like any other "normal" character in the alphabet. As far as querying is concerned, a prefix match to the string "jone" can be specified as a substring match on the (extended) string "#jone" to the PST. Similarly, suffix and exact matches to string "jone" can be specified as substring matches to the strings "jone$" and "#jone$", respectively.

3.3 Strings

For a string α, we use $\alpha[j]$ to denote the character at the jth position in α, and more generally $\alpha[i \ldots j]$ to denote the substring starting at the ith position and ending at the jth position of α inclusively. If α is obtained as the concatenation of strings α_1 and α_2, we write $\alpha = \alpha_1\alpha_2$; in other words, concatenation is implicitly expressed in terms of adjacency. If α_1 is a prefix of β, then the expression $\beta - \alpha_1$ gives the suffix α_2, where $\beta = \alpha_1\alpha_2$.

Definition 3 (Maximal overlap) *Given strings $\beta_1 = \alpha_1\alpha_2$ and $\beta_2 = \alpha_2\alpha_3$, where α_2 is maximal, we define the* maximal overlap *between a suffix of β_1 and a prefix of β_2, denoted by $\beta_1 \oslash \beta_2$, as α_2. The expression $\beta_2 - (\beta_1 \oslash \beta_2)$ gives α_3.* □

4 KVI and MO: Selectivity estimation algorithms

We begin our study of the substring selectivity estimation problem with the 1-D version. In Sect. 7 and beyond, we will consider the k-D version of the problem.

Employing a frequency interpretation of probability, we use $Pr(\sigma)$ to denote the selectivity of substring query σ, computed using the PST. If σ is found in the PST, we simply compute $Pr(\sigma) = C_\sigma/N$ (where N is the root count). If σ is not found in the PST, then we must *estimate* $Pr(\sigma)$. This is the essence of our *substring selectivity estimation* problem.

Let query $\sigma = \alpha_1 \ldots \alpha_w$. Let $Pr(\alpha_1 \ldots \alpha_j | \alpha_1 \ldots \alpha_{j-1})$ denote the probability of occurrence of $\alpha_1 \ldots \alpha_j$ given that the (prefix) string $\alpha_1 \ldots \alpha_{j-1}$ has already been observed. Then, $Pr(\sigma)$ can be written as:

$$\begin{aligned} Pr(\sigma) &= Pr(\alpha_1 \ldots \alpha_w | \alpha_1 \ldots \alpha_{w-1}) \cdot Pr(\alpha_1 \ldots \alpha_{w-1}) \\ &= \ldots \\ &= Pr(\alpha_1) \cdot [\Pi_{j=2}^{w} Pr(\alpha_1 \ldots \alpha_j | \alpha_1 \ldots \alpha_{j-1})]. \end{aligned} \tag{1}$$

4.1 Algorithm KVI

We denote the independence-based strategy I'_1 presented in [KVI96] as the KVI algorithm. Krishnan et al. empirically showed that this strategy is among their best strategies, and hence we compare our approaches with this strategy. Our techniques and results can be extended in a straightforward manner for comparison with the other independence-based strategies proposed in [KVI96].

The KVI algorithm takes advantage of the information in the PST, and assumes *complete conditional independence*. That is, it estimates each term in (1) as follows:

$$Pr(\alpha_1 \ldots \alpha_j | \alpha_1 \ldots \alpha_{j-1}) \approx Pr(\alpha_j). \tag{2}$$

A detailed description of the KVI algorithm is given in Fig. 2. Given the substring query σ, KVI performs the so-called greedy parsing of σ. It finds a sequence of strings $\alpha_1, \ldots, \alpha_w$ for some w such that (a) $\sigma = \alpha_1 \ldots \alpha_w$ and (b) for all $j \geq 1$, α_j is the longest prefix of $(\sigma - \alpha_1 - \ldots - \alpha_{j-1})$ that can be found in the PST $\mathcal{T}$. As shown in Fig. 2, there is also a boundary case, when the longest prefix of $(\sigma - \alpha_1 - \ldots - \alpha_{j-1})$ that can be found in $\mathcal{T}$ is the null string. In this case, α_j is set to be the first character of $(\sigma - \alpha_1 - \ldots - \alpha_{j-1})$.

Example 1 (KVI estimation) Consider the PST shown in Fig. 1. The substring query $\sigma = \texttt{jones}$ is parsed into `jon` and `es`. Accordingly, $KVI(\texttt{jones})$ is given by:

$$\begin{aligned} Pr(\texttt{jones}) &= Pr(\texttt{jon}) * Pr(\texttt{jones}|\texttt{jon}) \\ &\approx Pr(\texttt{jon}) * Pr(\texttt{es}) \\ &= (C_{\texttt{jon}}/N) * (C_{\texttt{es}}/N) \\ &= 1.25\%. \end{aligned}$$ □

4.2 Algorithm MO: Maximal overlap

Given a substring query σ, our MO algorithm computes *all* maximal substrings $\beta_1, \ldots, \beta_u$ of σ that can be found in the PST $\mathcal{T}$. These maximal substrings $\beta_1, \ldots, \beta_u$ satisfy collectively the condition: $\sigma = \beta_1[\beta_2 - (\beta_1 \oslash \beta_2)] \ldots [\beta_u - (\beta_{u-1} \oslash \beta_u)]$.

Example 2 (MO parsing) For the PST shown in Fig. 1, the substring query $\sigma = \texttt{jones}$ is parsed into $\beta_1 = \texttt{jon}$, $\beta_2 = \texttt{one}$ and $\beta_3 = \texttt{nes}$. Accordingly, $\beta_1 \oslash \beta_2$ and $\beta_2 \oslash \beta_3$ are the strings `on` and `ne`, respectively. □

With respect to (1), the query string can be decomposed into adjacent strings, α_i, as follows: $\alpha_1 = \beta_1$, and $\alpha_j = \beta_j - (\beta_{j-1} \oslash \beta_j), j > 1$. Then, MO estimates the conditional probability of $\alpha_1 \ldots \alpha_j$ given the prefix string $\alpha_1 \ldots \alpha_{j-1}$ as follows:

$$\begin{aligned} Pr(\alpha_1 \ldots \alpha_j | \alpha_1 \ldots \alpha_{j-1}) &\approx Pr(\beta_j | \beta_{j-1} \oslash \beta_j) \\ &= Pr(\beta_j)/Pr(\beta_{j-1} \oslash \beta_j). \end{aligned} \tag{3}$$

```
Algorithm KVI
Input: a PST $\mathcal{T}$ with pruning threshold $p$, and
       root count $N$; a substring query $\sigma$
Output: the estimate $KVI(\sigma)$

{ 1. $i$ = 1;
  2. While ($\sigma$ not equal null) {
     2.1   $\gamma$ = the longest prefix of $\sigma$ in $\mathcal{T}$;
     2.2   If ($\gamma$ equal null) {
     2.3      $\alpha_i = \sigma[1]$;
     2.4      $Pr(\alpha_i) = p/N$}
           Else {
     2.5      $\alpha_i = \gamma$;
     2.6      $Pr(\alpha_i) = C_{\alpha_i}/N$}
     2.7   $\sigma = \sigma - \alpha_i$;
     2.8   $i = i + 1$}
  3. $KVI = \Pi_i \ Pr(\alpha_i)$; return($KVI$)
}
```

Fig. 2. The KVI estimation algorithm

```
Algorithm MO
Input: a PST $\mathcal{T}$ with pruning threshold $p$, and
       root count $N$; a substring query $\sigma$
Output: the estimate $MO(\sigma)$

{ 1. $i = 1$; $\beta_0$ = null; $k = 0$;
  2. While ($\sigma$ not equal null) {
     2.1   $\gamma = \sigma[1 \ldots j]$ = the longest prefix of $\sigma$ in $\mathcal{T}$;
     2.2   If ($\gamma$ equal null) {
     2.3      $\beta_i = \sigma[1]$;
     2.4      $Pr(\beta_i) = p/N$;
     2.5      $k = 1; i = i + 1$}
     2.6   Else if ($j > k$) {
     2.7      $\beta_i = \gamma$;
     2.8      $Pr(\beta_i) = C_{\beta_i}/C_{\sigma[1 \ldots k]}$;
     2.9      $k = j; i = i + 1$}
     2.10  $\sigma = \sigma - \sigma[1]$; $k = k - 1$}
  3. $MO = \Pi_i \ Pr(\beta_i)$; return($MO$)
}
```

Fig. 3. The MO estimation algorithm

That is, MO captures the *conditional dependence of α_j on the immediately preceding (maximal overlap) substring $\beta_{j-1} \oslash \beta_j$ of σ.*

A more detailed description of the MO algorithm is given in Fig. 3. The algorithm keeps track of the positions of maximal substrings β_i of σ found in the PST $\mathcal{T}$, as well as the overlaps between them, using the (integer) position variables j and k. Once more, in the boundary case when some character in σ is not in the PST $\mathcal{T}$, the same solution is adopted as in KVI.

Example 3 (MO estimation) To continue with Example 2, $MO(\texttt{jones})$ is computed as follows:

$$\begin{aligned}
&Pr(\texttt{jones}) \\
&\quad = Pr(\texttt{jon}) \cdot Pr(\texttt{jone}|\texttt{jon}) \cdot Pr(\texttt{jones}|\texttt{jone}) \\
&\quad \approx Pr(\texttt{jon}) \cdot Pr(\texttt{one}|\texttt{on}) \cdot Pr(\texttt{nes}|\texttt{ne}) \\
&\quad = (C_{\texttt{jon}}/N) \cdot (C_{\texttt{one}}/C_{\texttt{on}}) \cdot (C_{\texttt{nes}}/C_{\texttt{ne}}) \\
&\quad = 1\%.
\end{aligned}$$
□

	Positive queries	Negative queries
	(avg. relative error)	(avg. standard error)
MO	−28%	0.08
KVI	+326%	0.15

Fig. 4. Estimation accuracy comparisons

4.3 MO versus KVI

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 [Sha51], to suggest modeling such sequences using Markov chains.

Recall that to estimate $Pr(\alpha_1 \ldots \alpha_j | \alpha_1 \ldots \alpha_{j-1})$, MO allows partial conditional dependence and uses the estimate $Pr(\beta_j | \beta_{j-1} \oslash \beta_j)$, whereas KVI assumes complete conditional independence and uses the estimate $Pr(\alpha_j)$. While it is not universally true that $Pr(\beta_j | \beta_{j-1} \oslash \beta_j)$ is a better estimate than $Pr(\alpha_j)$ for all distributions, we can establish the following result for strings that exhibit the short memory property:

Theorem 1 *Suppose that the strings in the database $\mathcal{D}$ exhibit the short memory property with memory length L. Consider a PST $\mathcal{T}$ and a substring query σ. Let $\beta_1, \ldots, \beta_n$ be the maximal substrings of σ in $\mathcal{T}$. Then, if $\forall i > 1 : \beta_{i-1} \oslash \beta_i$ has length $\geq$ L, then $MO(\sigma)$ is a better estimate (in terms of log ratio) than $KVI(\sigma)$.* □

Note that we have used the standard metric of log ratio to compare the goodness of a probability estimate.

In general, determining L is not practical, especially in the presence of updates, and the MO strategy of conditioning based on the longest preceding subsequence in the PST is a rational strategy.

4.4 Experimental evaluation

To complement our theoretical analysis presented above, we present preliminary experimental results comparing the quality of the estimates computed by KVI and MO.

We implemented both KVI and MO in C. We paid special attention to ensure that MO is not affected by round-off errors. The results reported below were obtained using a real AT&T data set containing information about over 100,000 employees. In particular, the reported results are based on the last name of each employee and on a pruned tree that keeps roughly 5% of the nodes with the highest counts.

Following the methodology used in [KVI96], we considered both "positive" and "negative" queries. Positive queries are strings that were present in the un-pruned tree (or in the database), but that were pruned. We used relative error, i.e., (estimated count − actual count)/actual count, as the metric for measuring the accuracy. We randomly picked 50 positive queries of variable length, of variable actual counts, and to cover different parts of the pruned tree. The results reported below give the average relative error over the 50 queries.

Negative queries are strings that were not in the un-pruned tree (or in the database). 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, i.e., the square root of the mean squared error.

The first column of the table in Fig. 4 compares the estimation accuracy between MO and KVI for positive queries. The average relative error of MO is -28%, whereas the corresponding error of KVI is $+326\%$. A detailed examination of each of the 50 queries used indicates that KVI has a strong tendency to overestimate by a wide margin, whereas MO has a roughly 50-50 chance of overestimating and underestimating.

The second column of the table in Fig. 4 compares the estimation accuracy between MO and KVI for negative queries. Because the actual count of a negative query is 0, the closer the average standard error is to 0, the more accurate the estimate. MO again is more accurate than KVI, even though both appear to give acceptable estimates for negative queries.

5 Using count-suffix tree constraints

While MO provides a better estimate for the substring selectivity of query string σ than KVI, it is possible that both estimates are infeasible, i.e., there may be no completion of PST $\mathcal{T}$ such that the count C_σ in the completion is equal to MO(σ) or to KVI(σ). The following example illustrates this possibility.

Example 4 (o-Suffix tree constraints) Suppose the PST in Fig. 1 is a pruned o-suffix tree. For the substring query `jes`, both KVI and MO estimate $Pr(\texttt{jes})$ as $Pr(\texttt{j}) \cdot Pr(\texttt{es}) = 2.5\%$.

Since the counts C_α in an o-suffix tree record the number of occurrences of α in the database $\mathcal{D}$, it must be the case that $C_\alpha \geq \sum C_{\alpha\alpha_1}$, for strings $\alpha\alpha_1$ corresponding to the children nodes (not all descendant nodes) of α in the PST. Specifically, for the PST in Fig. 1, observe that $C_{\texttt{j}} = C_{\texttt{jon}} + C_{\texttt{jack}}$. Hence, no completion of $\mathcal{T}$ can have a non-zero count corresponding to the string `jes`. Thus, using the constraints, one can infer that the substring selectivity of `jes` must be 0. □

Let us now repeat the exercise of Example 4 using a pruned p-suffix tree. The *key* difference between pruned p-suffix tree constraints and pruned o-suffix tree constraints is that the relationship $C_\alpha \geq \sum C_{\alpha\alpha_1}$ does not hold for pruned p-suffix trees. Instead, only a much weaker relationship, $C_\alpha \geq C_{\alpha\alpha_1}$, holds for each child node $\alpha\alpha_1$ of α in the pruned p-suffix tree. For example, for the `jes` query, the database $\mathcal{D}$ might have 10 strings containing both `jack` and `jon`, allowing for additional strings containing `jes`.

In the next two sections, we show that more can be done using pruned o-suffix tree constraints for developing accurate estimation algorithms.

5.1 o-Suffix tree constraints

There are three components contributing to C_α in an o-suffix tree. First, α appears as a string (by itself) in $\mathcal{D}$; we denote this number by O_α.[1] Second, α can appear as a suffix of a string in $\mathcal{D}$; this is the third term in (4) below. Third, α can appear as a proper, non-suffix, substring of a string in $\mathcal{D}$; this is the second term in (4).

Definition 4 *(ConSuffix(α)) Given a string α, we define ConSuffix(α) to be the following equality:*

$$C_\alpha = O_\alpha + \sum_{a_1 \in \mathcal{A}} (C_{\alpha a_1}) + \sum_{a_2 \in \mathcal{A}} \left(C_{a_2 \alpha} - \sum_{a_3 \in \mathcal{A}} (C_{a_2 \alpha a_3}) \right). \tag{4}$$

□

Alternatively, one can express the above three components contributing to C_α in an o-suffix tree in terms of prefixes, instead of suffixes. Then, we get the following definition:

Definition 5 *(ConPrefix(α)) Given a string α, we define ConPrefix(α) to be the equality:*

$$C_\alpha = O_\alpha + \sum_{a_1 \in \mathcal{A}} (C_{a_1 \alpha}) + \sum_{a_2 \in \mathcal{A}} \left(C_{\alpha a_2} - \sum_{a_3 \in \mathcal{A}} (C_{a_3 \alpha a_2}) \right). \tag{5}$$

□

5.2 Characterizing completions $\mathcal{C}(\mathcal{T})$

We can now characterize the set of all completions, $\mathcal{C}(\mathcal{T})$, of a pruned o-suffix tree $\mathcal{T}$. First, for each completion, it must satisfy (4) and (5) for each string in the completion. Second, the completion must agree with the "semantics" of $\mathcal{T}$, which is formalized as follows:

Definition 6 *(ConPrune($\alpha, \mathcal{T}, p$)) Given a pruned o-suffix tree $\mathcal{T}$, with pruning threshold p, denote k_α to be the count of α in $\mathcal{T}$. We define ConPrune($\alpha, \mathcal{T}, p$) to be the following constraint:*

$$\begin{aligned} C_\alpha &= k_\alpha, \text{ if } \alpha \text{ in } \mathcal{T}, \\ &\leq p, \text{ otherwise.} \end{aligned} \tag{6}$$

□

Definition 7 *(ConComp($\mathcal{T}, p$)) For a pruned o-suffix tree $\mathcal{T}$, with pruning threshold p, define ConComp($\mathcal{T}, p$) to be the set of constraints: $\{$ ConSuffix(α)$|\alpha \in \mathcal{A}^*\} \cup \{$ ConPrefix (α)$|\alpha \in \mathcal{A}^*\} \cup \{$ ConPrune($\alpha, \mathcal{T}, p$)$|\alpha \in \mathcal{A}^*\}$.* □

The following result, easily established by induction on the height of the tree, characterizes the set of all completions of a given PST $\mathcal{T}$.

Theorem 2 *Consider a pruned o-suffix tree $\mathcal{T}$, with pruning threshold p. An o-suffix tree is a completion of $\mathcal{T}$ if and only if the counts associated with its strings satisfy ConComp($\mathcal{T}, p$).* □

A straightforward corollary of the above result is that we only need to consider strings α in *ConComp*($\mathcal{T}, p$) that are bounded in length by N, the root count of $\mathcal{T}$.

A similar exercise can be repeated to give a complete characterization of completions of a pruned p-suffix tree.

[1] In general, $\mathcal{D}$ can have multiple occurrences of the same string.

5.3 Projection constraints

It is possible that the estimate $MO(\sigma)$ [and $KVI(\sigma)$], which uses only "local" information from $\mathcal{T}$, is *infeasible*, i.e., it is impossible for any completion of $\mathcal{T}$ (as characterized by Theorem 2) to agree with this estimate. Example 4 illustrates such a situation. In the following, we seek to *improve* the MO estimate whenever this estimate is infeasible.

Given a substring query σ, determining if $MO(\sigma)$ is feasible, with respect to *ConComp*$(\mathcal{T}, p)$, is NP-hard [Sch86]. In our effort to check efficiently whether $MO(\sigma)$ is feasible, we need to approximate *ConComp*$(\mathcal{T}, p)$, where a *sound approximation* of a set of constraints is one whose solution space is a superset of that of the original set of constraints. A simple sound approximation is the set

$$\{ConPrune(\alpha, \mathcal{T}, p) | \alpha \in \mathcal{A}^*\},$$

which only requires that strings not in $\mathcal{T}$ have counts that do not exceed the pruning threshold p. Observe that in Example 4 this sound approximation would consider the MO (and KVI) estimate to be feasible (since $p/N = 5/200 = 2.5\%$). We show that it is possible to obtain a "better" sound approximation of *ConComp*$(\mathcal{T}, p)$, without sacrificing a polynomial-time check of the feasibility of $MO(\sigma)$.

Definition 8 **(l- and r-parents)** *Given a string α of length m, we call the strings $\alpha[1 \dots (m-1)]$ and $\alpha[2 \dots m]$ the* l-parent *(l for left) and the* r-parent *(r for right) of α.* □

Observe that by rearranging the terms in (5) and dropping non-negative terms, we get the following inequality:

$$\begin{aligned} C_{a\alpha} &= C_\alpha - O_\alpha - \sum_{a_1 \in \mathcal{A}, a_1 \neq a} (C_{a_1\alpha}) \\ &\quad - \sum_{a_2 \in \mathcal{A}} \left(C_{\alpha a_2} - \sum_{a_3 \in \mathcal{A}} (C_{a_3 \alpha a_2}) \right) \\ &\leq C_\alpha - \sum_{a_1 \in \mathcal{A}, a_1 \neq a} (C_{a_1\alpha}) \\ &\leq C_\alpha - \sum_{a_1 \in \mathcal{A}, a_1 \neq a, a_1\alpha \in \mathcal{T}} (C_{a_1\alpha}). \end{aligned}$$

This and the symmetric inequality obtained by using (4) are formalized below.

Definition 9 *(l- **and** r-ConPar$(\alpha, \mathcal{T})$) Given a pruned o-suffix tree $\mathcal{T}$, and a string $\alpha = \alpha_1 a_1$ not in $\mathcal{T}$, we denote l-ConPar$(\alpha, \mathcal{T})$ to be the inequality:*

$$C_{\alpha_1 a_1} \leq C_{\alpha_1} - \sum_{a_2 \in \mathcal{A}, a_2 \neq a_1, \alpha_1 a_2 \in \mathcal{T}} (C_{\alpha_1 a_2}).$$

Similarly, given a string $\alpha = a_1\alpha_1$ not in $\mathcal{T}$, we denote r-ConPar$(\alpha, \mathcal{T})$ to be the inequality:

$$C_{a_1 \alpha_1} \leq C_{\alpha_1} - \sum_{a_2 \in \mathcal{A}, a_2 \neq a_1, a_2\alpha_1 \in \mathcal{T}} (C_{a_2 \alpha_1}).$$

□

Now, given a string α not in the PST $\mathcal{T}$, one can use l-*ConPar*$(\alpha, \mathcal{T})$ and r-*ConPar*$(\alpha, \mathcal{T})$ to obtain constraints on the count of C_α in terms of the counts of its l- and r-parents (as well as the counts of "siblings" of α in $\mathcal{T}$). If a parent string is not in $\mathcal{T}$, one can obtain analogous constraints on its count. Iterating this process until *all* the l- and r-parents are in $\mathcal{T}$ gives us a set of *projection constraints*, denoted *ConProj*$(\alpha, \mathcal{T}, p)$, which is a sound approximation of *ConComp*$(\mathcal{T}, p)$. We formalize this below.

```
Algorithm MOC
Input: a PST $\mathcal{T}$ with pruning threshold $p$, and
       root count $N$; a substring query $\sigma$
Output: the estimate $MOC(\sigma)$
{ 1. $MOC = MO(\sigma)$;
  2. let ConProj$(\sigma, \mathcal{T}, p)$ be of the form $C_\sigma \leq v_\sigma$;
  3. if $(MOC > v_\sigma/N)$ $\{MOC = v_\sigma/N\}$
  4. return($MOC$);
}
```

Fig. 5. The MOC estimation algorithm

Definition 10 *($anc(\alpha, \mathcal{T})$, ConProj$(\alpha, \mathcal{T}, p)$) Consider a pruned o-suffix tree $\mathcal{T}$, with pruning threshold p, and a string α not in $\mathcal{T}$.*

Define the set $anc(\alpha, \mathcal{T})$ to be the smallest set such that: (a) $\alpha \in anc(\alpha, \mathcal{T})$ and (b) if $\alpha_1 \in anc(\alpha, \mathcal{T})$ and α_2 is an l- or an r-parent of α_1, such that α_2 not in $\mathcal{T}$, then $\alpha_2 \in anc(\alpha, \mathcal{T})$. Intuitively, $anc(\alpha, \mathcal{T})$ is the set of all ancestors of α that are not in $\mathcal{T}$.

Define ConProj$(\alpha, \mathcal{T}, p)$ as the projection of the following constraints on C_α: $\{ConPrune(\alpha_1, \mathcal{T}, p) \mid \alpha_1 \in \mathcal{T}\} \cup \{l\text{-}ConPar(\alpha_1, \mathcal{T}) \mid \alpha_1 \in anc(\alpha, \mathcal{T})\} \cup \{r\text{-}ConPar(\alpha_1, \mathcal{T} \mid \alpha_1 \in anc(\alpha, \mathcal{T})\} \cup \{ConPrune(\alpha, \mathcal{T}, p)\}$. □

Example 5 (*ConProj*(`jones`, $\mathcal{T}, p$)) Consider the pruned o-suffix tree $\mathcal{T}$ shown in Fig. 1, with pruning threshold $p = 5$. For the substring query `jones`, anc(`jones`, $\mathcal{T}$) is the set {`jones`, `jone`, `ones`}. Assume all relevant nodes are as shown. *ConProj*(`jones`, $\mathcal{T}, p$) is given by the projection of the constraints below on $C_{\texttt{jones}}$:

$$\begin{aligned} C_{\texttt{jones}} &\leq p = 5 \\ C_{\texttt{jones}} &\leq C_{\texttt{jone}} \\ C_{\texttt{jones}} &\leq C_{\texttt{ones}} \\ C_{\texttt{jone}} &\leq C_{\texttt{jon}} - C_{\texttt{jond}} = 10 - 7 \\ C_{\texttt{jone}} &\leq C_{\texttt{one}} = 15 \\ C_{\texttt{ones}} &\leq C_{\texttt{one}} - C_{\texttt{oned}} = 15 - 7 \\ C_{\texttt{ones}} &\leq C_{\texttt{nes}} - C_{\texttt{anes}} = 20 - 13. \end{aligned}$$

This simplifies to the single inequality $C_{\texttt{jones}} \leq 3$. □

Putting all of the above together, we have the following theorem:

Theorem 3 *Given a pruned o-suffix tree $\mathcal{T}$, with pruning threshold p, and a string α not in $\mathcal{T}$, ConProj$(\alpha, \mathcal{T}, p)$ is (a) a sound-approximation of the projection of the constraints ConComp$(\mathcal{T}, p)$ on C_α, and (b) of the form $C_\alpha \leq v_\alpha$, for some value v_α.* □

5.4 Algorithm MOC: Maximal overlap with constraints

We use the constraints *ConProj*$(\sigma, \mathcal{T}, p)$ to create a new estimation algorithm, which we call *maximal overlap with constraints* (MOC), and present in Fig. 5.

Example 6 (Estimating $MOC(\texttt{jes})$) Consider the pruned o-suffix tree in Fig. 1, and the substring query `jes`. As shown in Example 4, $MO(\texttt{jes}) = KVI(\texttt{jes}) = 2.5\%$. The constraint *ConProj*$(\texttt{jes}, \mathcal{T}, p)$ is given by:

$$C_{\texttt{jes}} \leq C_{\texttt{j}} - C_{\texttt{jo}} - C_{\texttt{ja}} \ = \ 20 - 10 - 10$$

As a result, $MOC(\texttt{jes})$ would return 0, which is the *only* feasible value. □

Intuitively, if $MO(\sigma)$ is a feasible value for C_σ in *ConProj*$(\sigma, \mathcal{T}, p)$, the estimate $MOC(\sigma)$ is the same as $MO(\sigma)$. Otherwise, $MOC(\sigma)$ is set to the largest possible feasible value, v_σ, of C_σ. This directly leads to the following two results, which summarize the relative behavior of the MO and MOC algorithms:

Theorem 4 *Consider a pruned o-suffix tree $\mathcal{T}$. Then, it is the case that $MOC(\sigma) \leq MO(\sigma)$ for all σ.* □

Theorem 5 *Consider a pruned o-suffix tree $\mathcal{T}$. Then, $MOC(\sigma)$ is a better estimate (in terms of log ratio) than $MO(\sigma)$ for all σ.* □

6 Lattices and constraints

The $MOC(\sigma)$ estimate improves on the $MO(\sigma)$ estimate by "applying" constraints that relate C_σ to various C_α in the pruned o-suffix $\mathcal{T}$, such that α is a substring of σ. However, it should be possible, in principle, to "apply" the MOC algorithm one step at a time to all members of $anc(\sigma, \mathcal{T})$ and obtain an even better algorithm than MOC. In this section, we explore this possibility, and propose a new algorithm, MOLC, which validates our intuition.

6.1 The string completion lattice

We first formalize the notion of a step-at-a-time computation using a *string completion lattice*, defined below.

Definition 11 (String completion lattice) *For α not in PST $\mathcal{T}$, we define the* string completion lattice *of α with respect to $\mathcal{T}$, denoted $\mathcal{L}_\alpha$, inductively as follows: (a) α is a node in $\mathcal{L}_\alpha$ and (b) for any node α_1 in $\mathcal{L}_\alpha$, the l-parent and r-parent of α_1 are also nodes in $\mathcal{L}_\alpha$. There is an (undirected) edge (α_1, α_2) in $\mathcal{L}_\alpha$, if α_1 is an l-parent or an r-parent of α_2.*

The depth *of a node α_1 in $\mathcal{L}_\alpha$ is defined inductively as follows: if α_1 is in $\mathcal{T}$, $depth(\alpha_1) = 0$, otherwise $depth(\alpha_1) = 1 + max\{depth(\gamma_1), depth(\gamma_2)\}$, where γ_1, γ_2 are the l-parent and r-parent of α_1.* □

Example 7 (String completion lattice) Consider the PST $\mathcal{T}$ shown in Fig. 1, and the substring query `jones`. In this case, a relevant fragment of $\mathcal{L}_{\texttt{jones}}$ is given in Fig. 6. Nodes with counts correspond to strings in $\mathcal{T}$. □

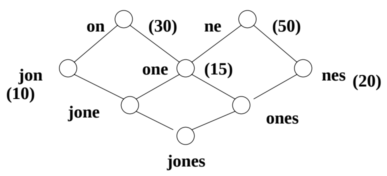

Fig. 6. $\mathcal{L}_{\texttt{jones}}$

```
Algorithm MOL
Input: a PST $\mathcal{T}$ with pruning threshold $p$, and
        root count $N$; a substring query $\sigma$
Output: the estimate $MOL(\sigma)$

{ 1. construct $\mathcal{L}_\sigma$;
  2. for all nodes $\alpha \in \mathcal{L}_\sigma$ of depth 0, $Pr(\alpha) = C_\alpha/N$;
  3. process nodes $\alpha$ in ascending order of $depth \geq 1$: {
     3.1    set $\gamma_1, \gamma_2$ the $l$- and $r$-parent of $\alpha$;
     3.2    $Pr(\alpha) = Pr(\gamma_1) \cdot Pr(\gamma_2)/Pr(\gamma_1 \oslash \gamma_2)$ }
  4. $MOL = Pr(\sigma)$; return($MOL$);
}
```

Fig. 7. The MOL estimation algorithm

```
Algorithm MOLC
Input: a PST $\mathcal{T}$ with pruning threshold $p$, and
        root count $N$; a substring query $\sigma$
Output: the estimate $MOLC(\sigma)$

{ 1. construct $\mathcal{L}_\sigma$;
  2. for all nodes $\alpha \in \mathcal{L}_\sigma$ of depth 0, $Pr(\alpha) = C_\alpha/N$;
  3. process nodes $\alpha$ in ascending order of $depth \geq 1$: {
     3.1    set $\gamma_1, \gamma_2$ the $l$- and $r$-parent of $\alpha$;
     3.2    $Pr(\alpha) = Pr(\gamma_1) \cdot Pr(\gamma_2)/Pr(\gamma_1 \oslash \gamma_2)$;
     3.3    let ConProj$(\alpha, \mathcal{T}, p)$ be $C_\alpha \leq v_\alpha$;
     3.4    if $(Pr(\alpha) > v_\alpha/N)$ { $Pr(\alpha) = v_\alpha/N$ } }
  4. $MOLC = Pr(\sigma)$; return($MOLC$);
}
```

Fig. 8. The MOLC estimation algorithm

6.2 Lattice-based estimation

As a step towards our goal of obtaining a step-at-a-time constraint-based estimation algorithm, we first extend the maximal overlap (MO) estimation algorithm to the lattice, and refer to it as the *maximal overlap on lattice* (MOL) algorithm. Figure 7 shows the MOL estimation algorithm. It is easy to show by induction on the depth that all terms on the right-hand side of step 3.2 are known each time the step is executed. Intuitively, the MOL algorithm repeatedly applies the MO algorithm to "complete" the fragment of the lattice that "supports" the given substring query.

Example 8 ($MOL(\texttt{jones})$) Consider the PST $\mathcal{T}$ in Fig. 1, the substring query `jones`, and the string completion lattice $\mathcal{L}_{\texttt{jones}}$ in Fig. 6. MOL first estimates $Pr(\texttt{jone})$ as

$$\begin{aligned} Pr(\texttt{jone}) &= (C_{\texttt{jon}}/N) \cdot (C_{\texttt{one}}/N)/(C_{\texttt{on}}/N) \\ &= (C_{\texttt{jon}} \cdot C_{\texttt{one}})/(N \cdot C_{\texttt{on}}) \\ &= 2.5\% \end{aligned}$$

and $Pr(\texttt{ones})$ as

$$Pr(\texttt{ones}) = (C_{\texttt{one}}/N) \cdot (C_{\texttt{nes}}/N)/(C_{\texttt{ne}}/N)$$

$$= (C_{\texttt{one}} \cdot C_{\texttt{nes}})/(N \cdot C_{\texttt{ne}})$$
$$= 3\%.$$

Then MOL estimates $Pr(\texttt{jones})$ as

$$\begin{aligned} Pr(\texttt{jones}) &= Pr(\texttt{jone}) \cdot Pr(\texttt{ones})/Pr(\texttt{one}) \\ &= (Pr(\texttt{jone}) \cdot Pr(\texttt{ones}) \cdot N)/C_{\texttt{one}} \\ &= 1\%, \end{aligned}$$

giving the same estimate as MO. □

The identical estimates by MO and MOL in the above example are not a coincidence, as shown by the following result:

Theorem 6 *Consider a PST $\mathcal{T}$. Then it is the case that $MOL(\sigma) = MO(\sigma)$, for all σ.* □

The proof is by induction on the depth of the string completion lattice of a substring query σ. It is reassuring to know that the MOL estimate is identical to the MO estimate. In particular, this means that the MO algorithm described earlier is sufficient to obtain the effect of full lattice completion. However, the incorporation of constraints has a positive effect over $MOC(\sigma)$, as we see next.

6.3 Algorithm MOLC

The MOL algorithm obtains estimates for the selectivities at multiple intermediate nodes and uses these as a basis to estimate the final answer. However, some of these intermediate estimates may be infeasible with respect to the constraints discussed previously. We would expect to do better if at each stage we applied constraints to the intermediate estimates and used these constrained estimates to determine the final desired answer. The algorithm, *maximal overlap on lattice with constraints* (MOLC), modifies MOL along the lines of MOC, and is shown in Fig. 8.

Example 9 ($MOLC(\texttt{jones})$) Continuing with Example 8, MOLC modifies the MOL estimate $Pr(\texttt{jone})$ to 1.5% because of the following constraint in *ConProj*$(\texttt{jone}, \mathcal{T}, p)$ (see Example 5):

$$C_{\texttt{jone}} \leq C_{\texttt{jon}} - C_{\texttt{jond}} \ = \ 3.$$

Similarly, MOLC modifies the MOL estimate $Pr(\texttt{ones})$ to $2.5\% = 5/200$ because of the following constraint in *ConProj*$(\texttt{ones}, \mathcal{T}, p)$:

$$C_{\texttt{ones}} \leq p \ = \ 5.$$

Consequently, the MOLC estimate $Pr(\texttt{jones})$ is reduced to $0.5\% = (3 \cdot 5)/(15 \cdot 200)$. Note that this is lower than the MO and MOC estimates. □

The following lemma is the key to establishing the subsequent theorems:

Lemma 1 *Consider a pruned o-suffix tree $\mathcal{T}$, a substring query σ, and the string completion lattice $\mathcal{L}_\sigma$. Then, for any node $\alpha \in \mathcal{L}_\sigma$, if step 3.4 of Algorithm MOLC lowers $Pr(\alpha)$, then the estimates for all nodes below α in $\mathcal{L}_\sigma$ are also reduced.* □

The following result is similar to Theorem 4.

Theorem 7 *Consider a pruned o-suffix tree $\mathcal{T}$. Then, it is the case that $MOLC(\sigma) \leq MOC(\sigma)$, for all σ.* □

The major result of this section is the following analog to Theorem 5.

Theorem 8 *Consider a pruned o-suffix tree $\mathcal{T}$. Then, it is the case that $MOLC(\sigma)$ is a better estimate (in terms of log ratio) than $MOC(\sigma)$, for all σ.* □

6.4 Trading accuracy for efficiency

Combining the results from Sects. 5 and 6, we have

$$0 \leq MOLC(\sigma) \leq MOC(\sigma) \leq MO(\sigma)[= MOL(\sigma)] \leq 1$$

for the values of the estimates produced by the various algorithms. The estimate $KVI(\sigma)$ can be anywhere in the $[0,1]$ range.

In terms of the error, expressed as the log ratio, using the various estimation algorithms, we have

$$\text{MOLC} \leq \text{MOC} \leq \text{MO}(= \text{MOL}) \leq \text{KVI}.$$

To understand the tradeoff between computational cost and estimation error, we study the computational costs of the various estimation algorithms.

Theorem 9 *Let s be the size of the alphabet $\mathcal{A}$. Let m be the length of the substring query σ. Assume a unit cost for each level that the PST is traversed, and that all traversals work their way down from the root. Let d be the depth of the PST. Then, the worst-case time costs of the various estimation algorithms are given by:*

1. *$Cost(KVI(\sigma))$ is $O(m)$.*
2. *$Cost(MO(\sigma))$ is $O(m \cdot d)$.*
3. *$Cost(MOC(\sigma))$ is $O(m \cdot s \cdot d)$.*
4. *$Cost(MOLC(\sigma))$ is $O(m^2 \cdot s \cdot d)$.* □

The costs of computing the estimates $MOC(\sigma)$ and $MOLC(\sigma)$ are dominated by the cost of computing the projection constraints. In the former case, it suffices to consider $O(m)$ constraints, each of which may have $O(s)$ terms. For an r-*ConPar*$(\alpha, \mathcal{T})$ constraint, determining the counts of its terms requires traversing $O(s)$ paths, each of length $O(d)$.[2] This gives the $O(m \cdot s \cdot d)$ bound. In the latter case, one needs to compute the projection constraints for *each* node in the string completion lattice $\mathcal{L}_\sigma$. In the worst case there are $O(m^2)$ such nodes, leading to the given bound. Hence, in terms of the computational effort (running time) required, the ordering is the opposite of the estimation accuracy ordering:

$$\text{MOLC} \geq \text{MOC} \geq \text{MO} \geq \text{KVI}.$$

7 Developing k-D structures for estimation

So far, in this paper, we have focussed on the 1-D problem. Next we turn our attention to the k-D problem. But before we

[2] One can pre-compute and store two additional constants per node in the PST and eliminate the dependence of the cost on s.

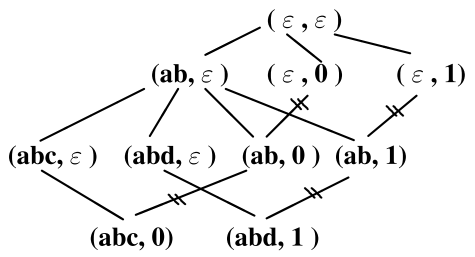

Fig. 9. Example 2-D count-trie

do so, in this section, we first establish a k-D data structure for estimation.

By a *k-D string*, we mean a k-tuple $(\alpha_1, \ldots, \alpha_k)$, where $\alpha_i \in \mathcal{A}^*$ for all $1 \le i \le k$. 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 α_i, $1 \le i \le k$.

7.1 k-D count-tries

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 from node $(\alpha_1, \ldots, \alpha_k)$ to node $(\beta_1, \ldots, \beta_k)$ iff
  - there exists $1 \le i \le k$ such that α_i is an immediate prefix of β_i; and
  - for all $j \ne i$, $1 \le j \le k$, $\alpha_j = \beta_j$.

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. For convenience, we restrict all our discussion for the k-D case to presence counting. Our techniques carry over easily to occurrence counting as well.

Figure 9 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 nodes all have count = 1.

As is done for standard 1-D count-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 Fig. 9, for instance, the node (abd, ε) can simply be stored as (d, ε); we show the prefix in the figure only for clarity.

7.2 k-D count-suffix DAGs

In 1-D, a suffix tree [Wei73,McC76] 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 a k-D string $(\alpha_1, \ldots, \alpha_k)$ in the count-suffix DAG, each of the k-D strings $(\gamma_1, \ldots, \gamma_k)$ is also in the DAG for *all* (improper) suffixes γ_i of α_i, $1 \le i \le k$.

For example, to make the count-trie shown in Fig. 9, 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.

7.3 Compressed representation: k-D count-suffix trees

Note that with a count-suffix DAG each query search begins from the root of the DAG. To answer a query, it thus suffices to ensure that there is a path from the root of the DAG to each node in the DAG. From this standpoint, a k-D count-suffix DAG is an *overkill*, in the sense that there may be *multiple* paths from the root to a node in the DAG (e.g., there are paths from $(\varepsilon, \varepsilon)$ to $(abc, 0)$ through $(\varepsilon, 0)$ as well as through (abc, ε) in Fig. 9). 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.

To do so, we first pick a canonical enumeration of the attributes.[3] 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*:

$$\begin{array}{l}(\alpha_{1,1}, \varepsilon, \ldots, \varepsilon), (\alpha_{1,2}, \varepsilon, \ldots, \varepsilon), \ldots, (\alpha_{1,m_1}, \varepsilon, \ldots, \varepsilon), \\ (\alpha_1, \alpha_{2,1}, \varepsilon, \ldots, \varepsilon), \ldots, (\alpha_1, \alpha_{2,m_2}, \varepsilon, \ldots, \varepsilon), \\ \qquad \cdots \\ (\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k,1}), \ldots, (\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k,m_k}),\end{array}$$

where for all $1 \le i \le k and 1 \le j < m_i$, $\alpha_{i,j}$ 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 Fig. 9, the canonical path from the root passes through the nodes (ab, ε) and (abc, ε). This path is guaranteed to exist 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 Fig. 9, 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,[4] 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 following, we use count-suffix trees and suffix trees interchangeably, for simplicity.

[3] 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.

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

8 Direct construction of PSTs

8.1 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 easy to see that the number of nodes is huge for very large databases and for $k \geq 2$.

To be more precise, first consider a 1-D count-trie. Indexing N strings, each of maximum length L, requires at most $N \cdot L$ nodes, assuming no sharing. For a 1-D count-suffix tree, because of all the suffixes, the same database requires $O(N \cdot L)$ strings, each of maximum length L. Thus, the total number of nodes, assuming no sharing, is $O(N \cdot L^2)$. With sharing of nodes between suffixes of a given string, the total number of nodes can be reduced to $O(N \cdot L)$. 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 $O(N \cdot L^k)$ 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 $O(N \cdot L^{2k})$ nodes in the k-D count-suffix tree. No sharing of nodes between suffixes is possible here.

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-1}. Even in the 1-D case, it has been argued [KVI96] 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.[5]

8.2 Rules for pruning

A tree can be pruned by using 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 $p \cdot N$. (We 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 $p \cdot N$ as the corresponding count threshold.) The threshold may be fixed a priori, or, for the approximate, probabilistic construction algorithms presented next, 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 $p \cdot N$ threshold by themselves. However, 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 Sect. 9.

8.3 Inadequate ways of creating pruned trees

Given the above rules for pruning, the next question is how exactly to create the PST 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 $\{(\gamma_1, \ldots, \gamma_k) \mid \text{ for } all \text{ (improper) suffixes } \gamma_i \text{ of } \alpha_i \text{ for all } 1 \leq i \leq k\}$. 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* PST. Recently, there has been considerable research activity around the creation of synopsis data structures in a fixed amount of space [GM98]. 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" [GM98], Gibbons and Matias developed an incremental maintenance algorithm to maintain a concise sample. In the following, 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) [GM98, 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 $comp(\mathcal{D})$.

[5] 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 be carried out, such as correlation testing, to select subgroups of attributes to be indexed. We do not concern ourselves in this paper on how such a selection can be made.

8.4 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 Sect. 9 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: Do a second 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., ≤ 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 PST 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 Sect. 8.2, the algorithm can simply skip over the strings to be exempted in the first pass, but count them in the second pass.

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 PST.

9 GNO and MO: k-D selectivity estimation algorithms

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 $1 \leq i \leq k$ $\sigma_i \in \mathcal{A}^*$ (and can be the null string), we use the PST 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 algorithms to do so.

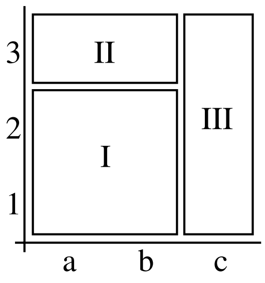

Fig. 10. 2-D query with GNO estimation

9.1 The GNO algorithm

Given query q, the GNO (Greedy Non-Overlap) algorithm applies greedy parsing to obtain non-overlapping k-D substrings of q. This generalizes the KVI algorithm for the 1-D problem. 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 Fig. 10. 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 $GNO(ab, 3)$ and $GNO(c, 123)$.[6] As it turns out, the substrings $(ab, 3)$ (rectangle II) and $(c,123)$ (rectangle III) are found in the pruned tree. Then the estimated selectivity is the product of the three selectivities.

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

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

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. If that were the case, GNO would not have used counts such as $C_{(ab,12)}$ from the pruned tree, and would have simply used counts such as $C_{(ab,\varepsilon)}$ and $C_{(\varepsilon,12)}$.

A skeleton of the GNO algorithm is given in Fig. 11. Step 1 can be implemented by a search of the pruned tree that finds

[6] 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 Sect. 10.

```
Algorithm GNO($\sigma_1, \ldots, \sigma_k$)
{ 1. Find from the pruned tree $(\gamma_1, \ldots, \gamma_k)$ where $\gamma_1$
       is the longest prefix of $\sigma_1$, and given $\gamma_1$, $\gamma_2$ is
       the longest prefix of $\sigma_2$, 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 $\delta_i$ such that $\sigma_i$ equal $\gamma_i\delta_i$.
       4.2 If ($\delta_i$ not equal to null)
             $gno = gno \cdot GNO(\gamma_1, \ldots, \gamma_{i-1}, \delta_i, \sigma_{i+1}, \ldots, \sigma_k)$.
     }
  5. Return $(gno)$.
}
```

Fig. 11. Pseudo code of algorithm GNO

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 $O(|\sigma_1| \cdot \ldots \cdot |\sigma_k|)$ times. This brings us back to the unit-cube pruning exemption rule mentioned in Sect. 8.2. The product $|\sigma_1| \cdot \ldots \cdot |\sigma_k|$ 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 pruning 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 Sect. 10.

In terms of formal properties of GNO, the following theorem shows that GNO generalizes the KVI algorithm. Given a k-D PST $\mathcal{T}$, 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 Fig. 9, 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 10 *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 $proj(\mathcal{T}, i)$.* □

9.2 The MO algorithm: Example

MO for multiple dimensions tries to find maximum overlapping substrings just as in the 1-D case. The complication is that the nature of overlap can now be considerably more complex. To illustrate, consider again the 2-D query $(abc, 123)$ shown in Fig. 10. While GNO finds three 2-D non-overlapping substrings, MO finds overlapping substrings. In Fig. 12, to highlight the comparison between MO and GNO, we assume that

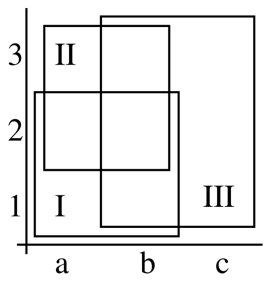

Fig. 12. 2-D query with MO estimation

MO also finds three substrings, corresponding to the ones shown in Fig. 10. (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 $(ab, 23)$ (rectangle II) and $(bc, 123)$ (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:

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

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 II in Fig. 12. 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 III, we take the product of probabilities $Pr\{(ab, 12)\}$, $Pr\{(ab, 23)\}$ and $Pr\{(bc, 123)\}$, basically counting all three rectangles. To compensate for double counting, we divide the product by the three 2-way intersections: (a) $Pr\{(ab, 2)\}$ between I and II; (b) $Pr\{(b, 12)\}$ between I and III; and (c) $Pr\{(b, 23)\}$ between II and III.

However, by dividing the 2-way intersections, we have "overcompensated." Specifically, the substring $(b, 2)$ is initially counted three times in the product, but is then discounted 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 is the 3-way intersection between the three initial substrings.

9.3 The MO algorithm: Pseudo code

The counting exercise illustrated in the above example is generalized in Fig. 13, which gives a skeleton of the k-D 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

```
Algorithm 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 $S$ to $\{(\lambda_1, 1), \ldots, (\lambda_u, 1)\}$, and $i$ to 1.
  3. Repeat {
     3.1 Initialize $S_{\text{new}}$ to $\emptyset$.
     3.2 For all $(\alpha, w) \in S$ such that $w$ equal $i$
           For all $1 \leq j \leq u$ {
           If [($\alpha$ not equal $\lambda_j$) and
              ($\alpha \cap \lambda_j$ non-empty)],
              add ($\alpha \cap \lambda_j$, $i+1$) to $S_{\text{new}}$.
           }
     3.3 $S = S \cup S_{\text{new}}$, and $i++$
     } until ($S_{\text{new}}$ equal $\emptyset$)
  4. Initialize $mo$ to 1.
  5. For all $(\alpha, w) \in S$ {
     5.1 Get count $C_\alpha$ from the pruned tree.
     5.2 If ($w$ is an odd integer), $mo = mo \cdot (C_\alpha/N)$
           Else $mo = mo/(C_\alpha/N)$
     }
  6. Return ($mo$).
}
```

Fig. 13. Pseudo code of algorithm MO

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 $w \leq u$.

After all 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 PST. 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, 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, but otherwise in the denominator.

9.4 The MO algorithm: Properties

A natural question to ask at this point is if Step 5.2 is "correct." As motivated in the example shown in Fig. 12, 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 2 *For any (α, 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. 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 discounted $\binom{w}{2}$ 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, $[-\sum_{j=1}^{w} (-1)^j \binom{w}{j}] = 1$. □

Next, we investigate how the k-D MO algorithm discussed in this section generalizes the 1-D MO algorithm presented in the first half of the paper.

Suppose for the query $abcde$, 1-D MO finds three maximal substrings: abc, bcd, and cde. Then 1-D MO, as presented earlier, gives the following estimate:

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

On the other hand, the k-D MO procedure shown in Fig. 13 gives the following estimate:

$$Pr\{abcde\} \approx \frac{(C_{abc}/N) \cdot (C_{bcd}/N) \cdot (C_{cde}/N) \cdot (C_c/N)}{(C_{bc}/N) \cdot (C_{cd}/N) \cdot (C_c/N)}.$$

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 each other out, 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. Thus 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 and vice versa for the other dimensions.

That k-D MO is a proper generalization of 1-D MO is easy to show by considering the nature of overlap possible in one dimension. Since each maximal substring has a new starting position, and the length of any maximal substring is finite, there can be at most a finite number of overlapping strings, and these can be ordered based on their starting positions. Wherever strings k and $k+2$ overlap, we must also have $k+1$ overlap. Thus, the three-overlap term exactly cancels the 2-apart 2-way overlap term. Similarly, where strings k and $k+3$ overlap, we must also have $k+1$ and $k+2$, leading to two as yet unaccounted for 3-way overlap terms (k,$k+1$,$k+3$) and (k,$k+2$,$k+3$), which exactly cancel the 3-apart 2-way overlap term and the 4-way overlap term. Proceeding thus, we can argue that all terms except two-way overlap cancel amongst neighboring terms. This leads to the following result:

Theorem 11 *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 Fig. 13 for q using $\mathcal{T}$ is identical to the estimate given by the 1-D MO algorithm 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. It is proved by showing that for any term with multiple dimensions expressed as the product of the corresponding component 1-D terms (due to independence between dimensions), all the one-dimensional terms generated will cancel.

Theorem 12 *Suppose the k dimensions are mutually independent, i.e., for all nodes $(\alpha_1, \ldots, \alpha_k)$ in the k-D count-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. In the worst case, Step 1 requires $O(|\sigma_1|^2 \cdot \ldots \cdot |\sigma_k|^2)$ 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.

10 Experimental evaluation

10.1 Experimental setup

We implemented the k-D MO and GNO algorithms. They were written in C. We paid special attention to ensure that MO is not affected by round-off 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 [KVI96], we considered both “positive” and “negative” queries and used relative error as one of the metrics for measuring accuracy. 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 underestimation to overestimation, we adjust an overestimated count by the pruning count, whenever the former is greater than the latter, i.e., [min(estimated count,pruning count) − actual count]/actual count.

	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)

Fig. 14. Estimation accuracy for positive queries

While relative error measures accuracy in relative terms, mean squared error 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.

10.2 MO versus GNO: Positive queries

The table in Fig. 14 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 underestimates, but the underestimation becomes less serious as the actual counts themselves become smaller. On the other hand, no such trend can be said about MO. Sometimes it underestimates, and other times it overestimates. But there cannot be any doubt that MO is the winner.

In Sect. 9.1, we point out that there are many different ways 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 Fig. 11, we also implemented and experimented with the other version. In general, there are some slight differences in the estimations. However, in terms of accuracy, the other version remains as poor.

10.3 MO versus GNO: Negative queries and runtime

The mean standard error for negative queries (averaged 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

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

Fig. 15. Estimation accuracy: the independence assumption

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

Fig. 16. Estimation accuracy for large-area positive queries

to compute, MO usually takes $O(10^{-4})$ seconds (on a 225 MHz machine). Nonetheless, we believe that the extra effort is worthwhile.

10.4 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 adjust the settings to get two equal-sized PSTs, 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.

Nevertheless, Fig. 15 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 the 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 results less accurate than those of 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.) However, using MO still gives better results than relying on the independence assumption.

10.5 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., $|\sigma_1| \cdot |\sigma_2|$) covered by $q = (\sigma_1, \sigma_2)$ is between 5 and 12. Two-dimensional strings corresponding to a smaller area tend to always be kept in the pruned tree. Figure 16 shows results for positive queries with "large areas," which are 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, compared with the alternatives, MO is still the best. Finding a way to improve accuracy with large-area positive queries is an interesting open problem.

11 Conclusions and future work

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

In this paper, we formally addressed the substring selectivity estimation problem, using PSTs. We presented several estimation algorithms based on probabilistic and constraint satisfaction approaches, compared them with previously known techniques, both formally and experimentally, and demonstrated the advantages of the MO family of estimation algorithms.

Many open problems remain. Whereas our techniques are substantially better than previously known techniques, we do not know yet if they are "optimal." Also, we have assumed the pruned suffix tree as a given in most of the foregoing – Is it possible to adjust the pruning technique to minimize estimation error? Is this adjustment sensitive to the choice of estimation algorithm? Finally, we have dealt with multiple string matches in parallel, but not yet included possible sharing between strings to be matched. Such sharing is likely to be common as we consider path queries in the context of XML. For example, univ.dept.name=CS AND univ.dept.bldg.name=Gates. Can one extend our algorithms to such situations?

Acknowledgements. The research effort of H.V. Jagadish was supported in part by the NSF under grant IIS-9986030.
We would like to thank Flip Korn and Zhiyuan Chen for their comments on an earlier version of the paper. We would also like to thank the anonymous reviewers for their constructive suggestions, which helped improve the paper considerably.

References

[GG96] Giancarlo R., Grossi R.: On the construction of classes of suffix trees for square matrices: Algorithms and applications. Inf Comput, 130(2):151–182, 1996

[Gia95] Giancarlo R.: A generalization of the suffix tree to square matrices, with applications. SIAM J Comput, 24(3):520–562, 1995

[GM98] Gibbons P.B., Matias Y.: New sampling-based summary statistics for improving approximate query answers. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 331–342, 1998

[HS95] Hernandez M.A., Stolfo S.J.: The merge/purge problem for large databases. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 127–138, 1995

[Ioa93] Ioannidis Y.: Universality of serial histograms. In: Proceedings of the International Conference on Very Large Databases, pp. 256–267, 1993

[IP95] Ioannidis Y., Poosala V.: Balancing histogram optimality and practicality for query result size estimation. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 233–244, 1995

[JKM$^+$98] Jagadish H.V., Koudas N., Muthukrishnan S., Poosala V., Sevcik K., Suel T.: Optimal histograms with quality guarantees. In: Proceedings of the International Conference on Very Large Databases, pp. 275–286, 1998

[JKNS99] Jagadish H.V., Kapitskaia O., Ng R.T., Srivastava D.: Multi-dimensional substring selectivity estimation. In: Proceedings of the International Conference on Very Large Databases, pp. 387–398, Edinburgh, Scotland, UK, September 1999

[JNS99] Jagadish H. V., Ng R.T., Srivastava D.: Substring selectivity estimation. In: Proceedings of the ACM Symposium on Principles of Database Systems, pp. 249–260, Philadelphia, Pa., June 1999

[KVI96] Krishnan P., Vitter J.S., Iyer B.: Estimating alphanumeric selectivity in the presence of wildcards. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 282–293, 1996

[LN90] Lipton R.J., Naughton J.F.: Query size estimation by adaptive sampling. In: Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 40–46, March 1990

[McC76] McCreight E.M.: A space-economical suffix tree construction algorithm. J ACM, 23:262–272, 1976

[MD88] Muralikrishna M., Dewitt D.: Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 28–36, 1988

[PI97] Poosala V., Ioannidis Y.E.: Selectivity estimation without the attribute value independence assumption. In: Proceedings of the International Conference on Very Large Databases, pp. 486–495, 1997

[PIHS96] Poosala V., Ioannidis Y., Haas P., Shekita E.: Improved histograms for selectivity estimation of range queries. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 294–305, 1996

[SAC$^+$79] Selinger P.G., Astrahan M., Chamberlin D., Lorie R., Price T.: Access path selection in a relational database management system. In: Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 23–34, June 1979

[Sch86] Schrijver A.: Theory of Linear and Integer Programming. Discrete Mathematics and Optimization. Wiley-Interscience, 1986

[Sha51] Shannon C.E.: Prediction and entropy of printed English. Bell Syst Tech J, 30(1):50–64, 1951

[Wei73] Weiner P.: Linear pattern matching algorithms. In: Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory, pp. 1–11, 1973

[WVI97] Wang M., Vitter J.S., Iyer B.: Selectivity estimation in the presence of alphanumeric correlations. In: Proceedings of the IEEE International Conference on Data Engineering, pp. 169–180, 1997