Substring Selectivity Estimation

Article · April 1999

DOI: 10.1145/303976.304001 · Source: CiteSeer

Raymond T. Ng
University of British Columbia
456 PUBLICATIONS **25,928** CITATIONS

Divesh Srivastava
AT&T
489 PUBLICATIONS **21,908** CITATIONS

Substring Selectivity Estimation

H. V. Jagadish[*]
U of Illinois, Urbana-Champaign
jag@cs.uiuc.edu

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

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

Abstract

With the explosion of the Internet, LDAP directories and XML, there is an ever greater need to evaluate queries involving (sub)string matching. Effective query optimization in this context requires good selectivity estimates. In this paper, we use pruned count-suffix trees as the basic framework for substring selectivity estimation.

We present a novel technique to obtain a good estimate for a given substring matching query, called MO (for Maximal Overlap), that estimates the selectivity of a query based on all maximal substrings of the query in the pruned count-suffix tree. We show that MO is provably better than the (independence-based) substring selectivity estimation technique proposed by Krishnan et al. [6], called KVI, under the natural assumption that strings exhibit the so-called "short memory" property. We complement our analysis with an experiment, using a real AT&T data set, that demonstrates that MO is substantially superior to KVI in the quality of the estimate. Finally, we develop and analyze two selectivity estimation algorithms, MOC and MOLC, based on MO and a constraint-based characterization of all possible completions of a given pruned count-suffix tree. We show that KVI, MO, MOC and MOLC illustrate an interesting tradeoff between estimation accuracy and computational efficiency.

[*]This work was done when the author was at AT&T Labs–Research, Florham Park, NJ 07932, USA.

[†]This work was done when the author was on sabbatical at AT&T Labs–Research, Florham Park, NJ 07932, USA.

1 Introduction

One often wishes to obtain a quick estimate of the number of times a particular value occurs as a substring 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 [1]. With the growing importance of the Web, LDAP (Lightweight Directory Access Protocol) directory servers [2], XML, and other text-based information stores, substring queries are becoming increasingly common, and accurate estimation of substring selectivity is becoming a pressing concern.

A commonly used data structure for indexing substrings in a database is the suffix tree [15, 8], 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 σ, one can locate all the desired matches using the suffix tree. To estimate substring selectivity, Krishnan et al. [6] considered two variations of the suffix tree: (i) a *count-suffix tree*, which maintains a count, C_α, for each substring α in the tree; and (ii) a *pruned count-suffix tree*, which retains only those substrings α (and their counts) for which C_α exceeds some prune threshold p. The *substring selectivity estimation* problem can be formally stated as follows:

> Given a pruned count-suffix tree $\mathcal{T}$ with a prune threshold p, and a substring query σ, estimate the fraction C_σ/N, where N is the count associated with the root of $\mathcal{T}$.

Based on the independence assumption, Krishnan et al. proposed variations of a basic selectivity estimation algorithm, which we refer to as the KVI algorithm. These algorithms were compared experimentally based on the TPC-D benchmark [14], but no formal analysis was provided in [6].

1.1 Our Contributions

In this paper, we formally address the substring selectivity estimation problem, and make the following contributions:

- In Section 2, we identify two useful meanings of C_α: (i) the number of strings in the database containing α as a substring, and (ii) the number of occurrences of α as a substring in the database of strings. We then define p-suffix trees (p for "presence") and o-suffix trees (o for "occurrence") as count-suffix trees that use the first or second interpretation of C_α respectively.
- In Section 3, we present a novel selectivity estimation algorithm MO (for "Maximal Overlap") that estimates the selectivity of σ, based on all maximal substrings, β_i, of σ in the pruned count-suffix tree. We demonstrate that MO is provably better than KVI, under the natural assumption that strings exhibit the so-called "short memory" property. We complement our analysis with an experiment, using a real AT&T data set, that demonstrates that MO is substantially superior to KVI in the quality of the estimate.
- In Sections 4 and 5, we develop constraint-based characterizations of all (o- or p-) suffix trees that are possible *completions* of a given pruned (o- or p-) suffix tree. Based on a sound approximation of this constraint-based characterization, we develop and analyze two selectivity estimation algorithms, MOC (for "Maximal Overlap with Constraints") and MOLC (for "Maximal Overlap on Lattice with Constraints"), for o-suffix trees. In Section 6, we show that KVI, MO, MOC and MOLC illustrate an interesting tradeoff between estimation accuracy and computational efficiency.

1.2 Related Work

The work most closely related to ours is that by Krishnan et al. [6], which has been described briefly above. We will present a more detailed comparison in the technical sections later.

Histograms have long been used for selectivity estimation in databases [12, 9, 7, 3, 4, 10, 5]. They have been designed to work well for numeric attribute value domains, and one can obtain good solutions to the histogram construction problem using known techniques (see, e.g., [10, 5]). For string domains, and the substring selectivity estimation problem, one could continue to use histograms by sorting substrings based on the lexicographic order, and associating the appropriate counts. However, in this case, a histogram bucket that includes a range of consecutive lexicographic values is not likely to produce a good approximation, since the the number of times a string occurs as a substring is likely to be very different for lexicographically successive substrings.

An alternative, suggested by Ioannidis and Poosala [4], is to create an end-biased histogram over a frequency sort of the attributes. The idea is to store exact values for as many of the more commonly occurring values (the high-frequency values) as possible, and then to place *all* the remaining values (the low-frequency values) in a single bucket. A query returns an exact count if it asks about a high-frequency value. Otherwise, it returns a default value, typically the mean of the frequencies over all low-frequency values.

When the end-biased histogram is applied to the string domain, it has a close parallel to a pruned count-suffix tree. The high-frequency values in the end-biased histogram correspond to nodes retained in the pruned count-suffix tree. The low-frequency values correspond to nodes pruned away. With this approach of estimating the selectivity of substring queries, if α_1 has been pruned, the same (default) value is returned for α_1 and $\alpha_1\alpha_2$, irrespective of the length of α_2. Pruned count-suffix trees do better by taking domain knowledge into account.

2 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 lowercase 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.

2.1 Suffix Trees

A *suffix tree* [15, 8] 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, $C_{\mathbf{ana}}$ would be 1, but with the second interpretation, $C_{\mathbf{ana}}$ 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.

Definition 2.1 [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. [6] 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 sequel, 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 just a quick estimate of the counts, it suffices to store a *pruned count-suffix tree* (PST) [6]. 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 prune 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 [6], even though our results apply to other well formulated, such as the level-based pruning rule, with appropriate obvious modifications.

We illustrate an example PST, with prune threshold $p = 5$, in Figure 1. Labels are presented for (among others) substrings related to the database string **jones**, with counts C_α shown in parenthesis for some of the nodes in the PST.

Definition 2.2 [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 $\mathcal{T}$. ▮

2.2 Strings

For a string α, we use $\alpha[j]$ to denote the character at the j-th position in α, and more generally $\alpha[i..j]$ to denote the substring starting at the i-th position and ending at the j-th 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 2.3 [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. ▮

3 MO and KVI: Selectivity Estimation Algorithms

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

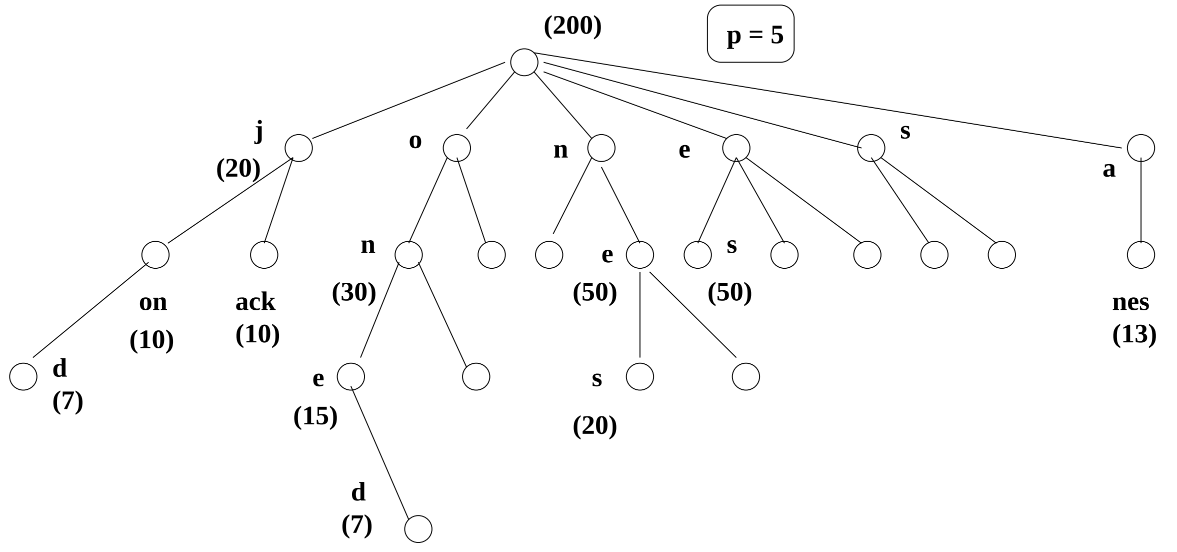

Figure 1: Example PST

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 substring query $\sigma = \alpha_1 \ldots \alpha_w$. Then, $Pr(\sigma)$ can be written as:

$$\begin{aligned} Pr(\sigma) &= Pr(\alpha_w|\alpha_1 \ldots \alpha_{w-1}) * Pr(\alpha_1 \ldots \alpha_{w-1}) \\ &= \ldots \qquad (1) \\ &= Pr(\alpha_1) * (\Pi_{j=2}^{w} Pr(\alpha_j|\alpha_1 \ldots \alpha_{j-1})) \end{aligned}$$

Intuitively, $Pr(\alpha_j|\alpha_1 \ldots \alpha_{j-1})$ denotes the probability of occurrence of α_j given that the preceding string $\alpha_1 \ldots \alpha_{j-1}$ has been observed.

3.1 Algorithm KVI

We denote the independence-based strategy I_1' presented in [6] 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 [6].

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

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

A detailed description of the KVI algorithm is given in Figure 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 (i) $\sigma = \alpha_1 \ldots \alpha_w$; (ii) α_1 is the longest prefix of σ that can be found in the PST $\mathcal{T}$; and (iii) for all $j > 1$, 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 Figure 2, there is also the 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 3.1 [KVI Estimation]
Consider the PST shown in Figure 1. The substring query $\sigma =$ **jones** is parsed into **jon** and **es**. Accordingly, KVI(**jones**) is given by:

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

3.2 Algorithm MO: Maximal Overlap

Given a substring query σ, our *maximal overlap* 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: $\sigma = \beta_1[\beta_2 - (\beta_1 \oslash \beta_2)] \ldots [\beta_u - (\beta_{u-1} \oslash \beta_u)]$.

```
Algorithm KVI
Input: a PST $\mathcal{T}$ with prune 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 $\sigma$ found 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($KVI$);
}
```

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

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

Figure 2: The KVI and MO Estimation Algorithms

Example 3.2 [MO Parsing]
For the PST shown in Figure 1, the substring query $\sigma = \textbf{jones}$ is parsed into $\beta_1 = \textbf{jon}$, $\beta_2 = \textbf{one}$ and $\beta_3 = \textbf{nes}$. Accordingly, $\beta_1 \oslash \beta_2$ and $\beta_2 \oslash \beta_3$ are the strings **on** and **ne**, respectively. ▮

With respect to Equation (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 α_j given the preceding string $\alpha_1 \ldots \alpha_{j-1}$ as follows:

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

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 Figure 2. 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 3.3 [MO Estimation]
To continue with Example 3.2, $MO(\textbf{jones})$ is computed as follows:

$$\begin{aligned} Pr(\textbf{jones}) &= Pr(\textbf{jon}) * Pr(\textbf{e}|\textbf{jon}) * Pr(\textbf{s}|\textbf{jone}) \\ &\approx Pr(\textbf{jon}) * Pr(\textbf{e}|\textbf{on}) * Pr(\textbf{s}|\textbf{ne}) \\ &= (C_{\textbf{jon}}/N) * (C_{\textbf{one}}/C_{\textbf{on}}) * \\ &\quad (C_{\textbf{nes}}/C_{\textbf{ne}}) \\ &= 1\% \end{aligned}$$ ▮

3.3 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 [13], to suggest modeling such sequences by Markov chains.

Recall that to estimate $Pr(\alpha_j|\alpha_1 \ldots \alpha_{j-1})$, MO allows partial conditional dependence and uses $Pr(\alpha_j|\beta_{j-1} \oslash \beta_j)$, whereas KVI assumes complete conditional independence and uses $Pr(\alpha_j)$. While it is not universally true that $Pr(\alpha_j|\beta_{j-1} \oslash \beta_j)$ 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 3.1 *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.

3.4 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 roundoff 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 the 5% of nodes with the highest counts.

Following the methodology used in [6], 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, 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 Figure 3 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 over-estimate by a wide margin, whereas MO has a roughly 50-50 chance of over-estimating and under-estimating.

The second column of the table in Figure 3 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 to 0, the more accurate is the estimate. MO again is more accurate than KVI, even though both appear to give acceptable estimates for negative queries.

4 Using Count-Suffix Tree Constraints

Example 4.1 [*o*-Suffix Tree Constraints] Suppose the PST in Figure 1 is a pruned *o*-suffix tree. For the substring query **jes**, both KVI and MO estimate $Pr(\mathbf{jes})$ as $Pr(\mathbf{j}) * Pr(\mathbf{es}) = 2.5\%$.

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 $C_\alpha \geq \Sigma C_{\alpha\alpha_1}$, for strings $\alpha\alpha_1$ corresponding to the children nodes (not all descendant nodes) of α in the PST. Specifically, for the PST in Figure 1, observe that $C_{\mathbf{j}} = C_{\mathbf{jon}} + C_{\mathbf{jack}}$. 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 4.1 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 $C_\alpha \geq \Sigma C_{\alpha\alpha_1}$ 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 rest of this paper, we show that more can be done using pruned *o*-suffix tree constraints for developing accurate estimation algorithms.

4.1 *o*-Suffix Tree Constraints

There are three components contributing to C_α in an *o*-suffix tree. First, α appears as a string

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

Figure 3: Estimation Accuracy Comparisons

(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 Equation (4) below. Third, α can appear as a proper, non-suffix, substring of a string in $\mathcal{D}$; this is the second term in Equation (4).

Definition 4.1 [$ConSuffix(\alpha)$] Given a string α, we define the following equality:

$$\begin{aligned} C_\alpha &= O_\alpha + \Sigma_{a_1 \in \mathcal{A}}(C_{\alpha a_1}) + \\ &\quad \Sigma_{a_2 \in \mathcal{A}}(C_{a_2\alpha} - \Sigma_{a_3 \in \mathcal{A}}(C_{a_2 \alpha a_3})) \end{aligned} \quad (4)$$

as $ConSuffix(\alpha)$. ▮

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 4.2 [$ConPrefix(\alpha)$] Given a string α, we define the following equality:

$$\begin{aligned} C_\alpha &= O_\alpha + \Sigma_{a_1 \in \mathcal{A}}(C_{a_1\alpha}) + \\ &\quad \Sigma_{a_2 \in \mathcal{A}}(C_{\alpha a_2} - \Sigma_{a_3 \in \mathcal{A}}(C_{a_3 \alpha a_2})) \end{aligned} \quad (5)$$

as $ConPrefix(\alpha)$. ▮

4.2 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 Equations (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 4.3 [$ConPrune(\alpha, \mathcal{T}, p)$] Given a pruned o-suffix tree $\mathcal{T}$, with prune threshold p, denote the count of α in $\mathcal{T}$ as k_α. We define the following constraint:

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

as $ConPrune(\alpha, \mathcal{T}, p)$. ▮

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

Definition 4.4 [$ConComp(\mathcal{T}, p)$] For a pruned o-suffix tree $\mathcal{T}$, with prune threshold p, define the following set of constraints:

$$\begin{gathered} \{ConSuffix(\alpha) | \alpha \in \mathcal{A}^*\} \cup \\ \{ConPrefix(\alpha) | \alpha \in \mathcal{A}^*\} \cup \\ \{ConPrune(\alpha, \mathcal{T}, p) | \alpha \in \mathcal{A}^*\} \end{gathered}$$

as $ConComp(\mathcal{T}, p)$. ▮

The following result characterizes the set of all completions of a given PST $\mathcal{T}$.

Theorem 4.1 *Consider a pruned o-suffix tree $\mathcal{T}$, with prune 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.

4.3 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 4.1) to agree with this estimate. Example 4.1 illustrates such a situation. In the sequel, we seek to *improve* the MO estimate whenever this estimate is infeasible.

Given a substring query σ, determining whether $MO(\sigma)$ is feasible, wrt $ConComp(\mathcal{T}, p)$, is NP-hard [11]. In our effort to check efficiently whether $MO(\sigma)$ is feasible, we need to soundly 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 prune threshold p. Observe that, in Example 4.1, 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 4.5 [*l*- **and** *r*-**Parents**] Given a string α of length m, we call the strings $\alpha[1..(m-1)]$ and $\alpha[2..m]$ the *l-parent* (*l* for left) and the *r-parent* (*r* for right) of α. ▮

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

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

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

Definition 4.6 [*l*- **and** *r*-$ConPar(\alpha, \mathcal{T})$] Consider a pruned *o*-suffix tree $\mathcal{T}$. Given a string $\alpha = \alpha_1 a_1$ not in $\mathcal{T}$, we denote the inequality:

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

as *l*-$ConPar(\alpha, \mathcal{T})$.

Similarly, given a string $\alpha = a_1 \alpha_1$ not in $\mathcal{T}$, we denote the inequality:

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

as *r*-$ConPar(\alpha, \mathcal{T})$. ▮

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.

Definition 4.7 [$anc(\alpha, \mathcal{T})$, $ConProj(\alpha, \mathcal{T}, p)$] Consider a pruned *o*-suffix tree $\mathcal{T}$, with prune threshold p, and a string α not in $\mathcal{T}$.

Define the set $anc(\alpha, \mathcal{T})$ to be the smallest set such that: (i) $\alpha \in anc(\alpha, \mathcal{T})$, and (ii) 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 set of 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 4.2 [$ConProj$(**jones**, $\mathcal{T}, p$)] Consider the pruned *o*-suffix tree $\mathcal{T}$ shown in Figure 1, with prune 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 $C_{\mathbf{jones}}$:

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

This simplifies to the single inequality $C_{\mathbf{jones}} \leq 3$. ▮

Theorem 4.2 *Given a pruned o-suffix tree $\mathcal{T}$, with prune threshold p, and a string α not in $\mathcal{T}$, $ConProj(\alpha, \mathcal{T}, p)$ is (i) a sound-approximation of the projection of $ConComp(\mathcal{T}, p)$ on C_α, and (ii) of the form $C_\alpha \leq v_\alpha$, for some value v_α.* ▮

```
Algorithm MOC
Input: a PST $\mathcal{T}$ with prune 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($MOC$);
}
```

Figure 4: The MOC Estimation Algorithms

4.4 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 Figure 4.

Example 4.3 [Estimating MOC(jes)]
Consider the pruned *o*-suffix tree in Figure 1, and the substring query **jes**. As shown in Example 4.1, $MO(\textbf{jes}) = KVI(\textbf{jes}) = 2.5\%$. The constraint $ConProj(\textbf{jes}, \mathcal{T}, p)$ is given by:

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

As a result, $MOC(\textbf{jes})$ 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 leads to the following two results, which summarize the relative behavior of the MO and MOC algorithms.

Theorem 4.3 *Consider a pruned o-suffix tree $\mathcal{T}$. Then, it is the case that $MOC(\sigma) \leq MO(\sigma)$ for all σ.* ▮

Theorem 4.4 *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.* ▮

5 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.

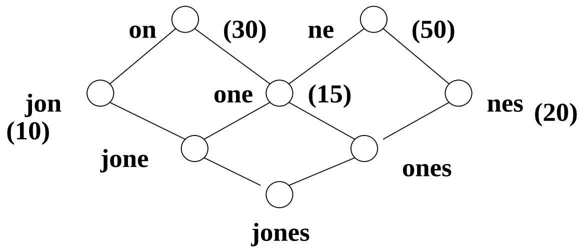

Figure 5: $\mathcal{L}_{\textbf{jones}}$

5.1 The String Completion Lattice

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

Definition 5.1 [String Completion Lattice]
For α not in PST $\mathcal{T}$, we define the *string completion lattice* of α wrt $\mathcal{T}$, denoted $\mathcal{L}_\alpha$, inductively as follows: (i) α is a node in $\mathcal{L}_\alpha$; (ii) 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: (i) If α_1 is in $\mathcal{T}$, $depth(\alpha_1) = 0$. (ii) 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 5.1 [String Completion Lattice]
Consider the PST $\mathcal{T}$ shown in Figure 1, and the substring query **jones**. In this case, a relevant fragment of $\mathcal{L}_{\textbf{jones}}$ is given in Figure 5. Nodes with counts correspond to strings in $\mathcal{T}$. ▮

5.2 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 6 shows the MOL estimation algorithm. It is easy

```
Algorithm MOL
Input: a PST $\mathcal{T}$ with prune 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    $Pr(\alpha) = Pr(\gamma_1) * Pr(\gamma_2)/Pr(\gamma_1 \oslash \gamma_2)$; }
  4. $MOL = Pr(\sigma)$; return($MOL$);
}
```

```
Algorithm MOLC
Input: a PST $\mathcal{T}$ with prune 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    $Pr(\alpha) = Pr(\gamma_1) * Pr(\gamma_2)/Pr(\gamma_1 \oslash \gamma_2)$;
     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($MOLC$);
}
```

Figure 6: The MOL and MOLC Estimation Algorithms

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 5.2 [MOL(**jones**)]
Consider the PST $\mathcal{T}$ in Figure 1, the substring query **jones**, and the string completion lattice $\mathcal{L}_{\mathbf{jones}}$ in Figure 5. MOL first estimates $Pr(\mathbf{jone})$ as:

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

and $Pr(\mathbf{ones})$ as:

$$\begin{aligned} Pr(\mathbf{ones}) &= (C_{\mathbf{one}}/N) * (C_{\mathbf{nes}}/N)/(C_{\mathbf{ne}}/N) \\ &= (C_{\mathbf{one}} * C_{\mathbf{nes}})/(N * C_{\mathbf{ne}}) \\ &= 3\% \end{aligned}$$

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

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

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 5.1 *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.

5.3 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 should expect to do better if at each stage we applied constraints on 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 Figure 6.

Example 5.3 [$MOLC$(**jones**)]
Continuing with Example 5.2, MOLC modifies the MOL estimate $Pr(\mathbf{jone})$ to $1.5\% = 3/200$ because of the following constraint in $ConProj(\mathbf{jone}, \mathcal{T}, p)$ (cf. Example 4.2):

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

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

$$C_{\textbf{ones}} \;\leq\; p \;=\; 5$$

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

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

Lemma 5.1 *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 4.3.

Theorem 5.2 *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 4.4.

Theorem 5.3 *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 Trading Accuracy for Efficiency

Combining the results from Sections 4 and 5, we have

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

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

$$MOLC \leq MOC \leq MO(= MOL) \leq KVI$$

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

Theorem 6.1 *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 $O(m * d)$.*
3. *$Cost(MOC(\sigma))$ is $O(m * s * d)$.*
4. *$Cost(MOLC(\sigma))$ is $O(m^2 * s * d)$.* ▮

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. When a constraint is an $r\text{-}ConPar(\alpha, \mathcal{T})$ constraint, determining the counts of its terms requires traversing $O(s)$ paths, each of length $O(d)$.[2] This gives the $O(m * s * d)$ 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:

$$MOLC \geq MOC \geq MO \geq KVI$$

7 Conclusions and Future Work

In this paper, we formally addressed the substring selectivity estimation problem for both pruned *p*- and *o*-suffix trees. We presented several estimation algorithms, MO, MOC and MOLC, based on probabilistic and constraint satisfaction approaches, and formally compared them with previously known techniques. For *p*-suffix trees, $MO(\sigma)$ is a good choice. For *o*-suffix trees, $MOLC(\sigma)$ is a good choice, unless the string completion lattice is "large", in which case $MOC(\sigma)$ represents a good balance between estimation accuracy and computational efficiency.

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

Many interesting problems remain open. We mention a couple of them. We have shown that *MOLC* is our most accurate estimation algorithm; can one show that *MOLC* is in fact optimal in terms of minimizing error? We have worked with a pre-determined PST; how can one choose an optimal pruned suffix tree, given a certain amount of space?

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] Transaction Processing Performance Council (TPC), 777 N. First Street, Suite 600, San Jose, CA 95112, USA. *TPC Benchmark D (Decision Support)*, May 1995.

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