Available online at www.sciencedirect.com
SCIENCE DIRECT®
ACADEMIC PRESS
Journal of Computer and System Sciences 66 (2003) 98–132
JOURNAL OF COMPUTER AND SYSTEM SCIENCES
http://www.elsevier.com/locate/jcss

Generalized substring selectivity estimation

Zhiyuan Chen,[a,*] Flip Korn,[b] Nick Koudas,[b] and S. Muthukrishnan[b]

[a] *Department of Computer Science, 4110 Upson Hall, Cornell University, Ithaca, NY 14853 USA*
[b] *AT&T Labs-Research, 180 Park Ave., Florham Park, NJ 07932, USA*

Received 1 October 2000; revised 15 March 2002

Abstract

In a variety of settings from relational databases to LDAP to Web applications, there is an increasing need to quickly and accurately estimate the count of tuples (LDAP entries, Web documents, etc.) matching Boolean substring queries. In providing such selectivity estimates, the correlation between different occurrences of substrings is crucial. Selectivity estimation for generalized Boolean queries has not been studied previously; our own prior work, which is discussed and extended herein, applies to the case of one-dimensional Boolean queries [CKKM00]. Existing methods for the case of multidimensional conjunctive queries approximate selectivities by explicitly storing *cross-counts* of frequently co-occurring combinations of substrings; estimates are obtained by parsing the query into multidimensional substrings corresponding to stored cross-counts and applying probabilistic formulae. The major problem with these methods is that the number of cross-counts stored by known methods increases exponentially with the number of dimensions (a "space dimensionality explosion") due to the need to capture the correlation amongst the dimensions. Hence, given a limited amount of space, none of the existing methods can reliably give accurate estimates. Moreover, these methods do not generalize to Boolean queries gracefully. We present a novel approach to selectivity estimation for generalized Boolean substring queries with a focus on the two cases of (1) conjunctive multidimensional and (2) Boolean queries. Our approach does not explicitly store cross-counts, but rather generates them on-the-fly. We employ a Monte Carlo technique called *set hashing* to succinctly represent the set of tuples containing a given substring as a signature vector of hash values; any combination of set hash signatures gives a cross-count when intersected. Thus, using only linear storage, a large number of cross-counts can be generated including those for complex co-occurrences of substrings. The cross-counts generated by our methods are not exact, but they are adequate for selectivity estimation. We present results from an extensive experimental evaluation of our approach on real data sets. For the case of multidimensional conjunctive queries, our approach achieves better accuracy by an order of magnitude, and scales much more gracefully to higher dimensions, than existing methods. Surprisingly, even though our approach involves generating cross-counts on-the-fly, estimation is very fast, taking 200 μs on a data set of size 6 MB. For the case of Boolean queries, our experiments also demonstrate the

*Corresponding author.
E-mail addresses: zhychen@cs.cornell.edu (Z. Chen), flip@research.att.com (F. Korn), koudas@research.att.com (N. Koudas), muthu@research.att.com (S. Muthukrishnan).

0022-0000/03/$ - see front matter © 2003 Published by Elsevier Science (USA).
PII: S0022-0000(02)00031-4

superiority of this approach over a straightforward independence-based approach wherein correlations are not captured.
© 2003 Published by Elsevier Science (USA).

1. Introduction

In recent years, a new suite of services (such as LDAP directory services), standards (such as XML) and applications (such as E-Commerce) have emerged due to the proliferation of the Internet. Handling large amounts of text is central to such Internet-related technology. Hence, there is resurgence of interest in the storage, management, and query processing of textual data.

In processing textual data, *substring queries* are fundamental, that is, determining the documents (or tuples, respectively) that contain a given query substring. For example, in database systems, the SQL `LIKE` predicate involves a substring query for alphanumeric attributes [DD97]. Similarly, in Information Retrieval, substring queries are common (see [BYRN99] for general introduction, [KT00] for information retrieval on the web, and [MMSZ98] for the currently best known algorithmic bounds for this problem). In emerging applications, however, textual query processing often calls for more general queries. We give examples of two such generalizations of interest here.

1.1. Multidimensional substring queries

In many applications of databases that process textual data, from E-Commerce to LDAP directories, users pose (sub)string queries on multiple string attributes. Such queries are either for exact matches or contain wildcards. For example, in an LDAP directory search, a user might inquire about all people whose first name begins with the letters "Jo" (a prefix match), whose phone number contains the digit sequence "360" (a substring match), and whose age is greater than 35 (a numerical range query). This query can be expressed in SQL using the `LIKE` clause: `WHERE (name LIKE 'Jo%' AND phone LIKE '%360%') AND age > 35`. Notice that queries can specify any combination of exact, prefix, suffix, or proper substring matches.

1.2. Boolean substring queries

The multidimensional substring queries above contain only a conjunction between substring predicates. *Boolean queries* provide more generality, that is, logical expressions composed of substring predicates connected by the Boolean operators `AND`, `OR`, and `NOT`, or recursively composed expressions thereof. Boolean queries have been employed in Information Retrieval systems for decades [SM83]. Two examples of such queries are `peanut OR butter`, which finds documents containing either the word `peanut` or the word `butter` (or both); and `peanut AND NOT butter`, which finds documents containing `peanut` but not containing `butter`; these are examples of a Boolean queries on a single attribute (being the entire document). Due to the proliferation of the Internet, such queries have become ubiquitous, e.g., in Web search engines, online digital libraries, etc. For instance, the AltaVista search engine receives more than 13 million queries per

day of which more than two-thirds involve some Boolean relationship between multiple substrings in a document [SHM98].

More generally, one may consider Boolean queries over substring predicates drawn from multiple attributes. In this paper, we present techniques that can be applied to multidimensional substring estimation with arbitrary Boolean expressions with a focus on two common cases: (1) conjunctive queries on multiple attributes and (2) Boolean queries. We study an important problem associated with such queries, namely, *selectivity estimation*. Given a conjunctive substring query on multiple attributes (respectively, Boolean substring query), the *multi-dimensional substring selectivity estimation* problem (resp., *Boolean substring selectivity estimation* problem) is to estimate the fraction of tuples (entries, documents) in the database that match the substring predicate in each attribute.[1] It is often very useful to obtain a fast and accurate estimation of this selectivity for reasons described below.

1.2.1. Role of substring selectivity estimation

The selectivity of substring queries can be used by the system for query optimization in Information Retrieval systems, for example, to find the best ordering of keywords for filtering. In relational databases, selectivity of substring queries is also useful in optimizing the query execution. In the above example conjunctive query `WHERE (name LIKE 'Jo%' AND phone LIKE '%360%') AND age >35`, knowing the result size of the two-dimensional substring filter versus that of the numerical range filter can determine which selection to process first. Also, if a two-dimensional string index [JKS00] exists on the string attributes and a B-tree index exists on `age`, one can use selectivity estimates to optimize access path selection.

Selectivity estimates may also be useful for query *refinement*. A notorious problem in Information Retrieval is that the number of documents matching a given query is often too large for a user to shift through. Unfortunately, ranking the documents based on relevance is not effective when the search terms are too general [CCT00]. Providing online result sizes has been shown to be helpful to users in refining queries, and has been proposed as a remedy to the low precision problem [TBS97,VWSG97]. Indeed, providing fast estimates is the basis of so-called dynamic queries [TBS97].

Finally, selectivity estimates could serve as an acceptable approximate answer to a `COUNT` query either by itself or in the context of [HHW97] where it will be refined further. Hence, multidimensional and Boolean substring selectivity estimation has many uses.

1.2.2. Informal overview of our techniques and results

We will informally motivate the difficulties that arise in multidimensional and Boolean substring query estimation and sketch how our novel approach that relies on hashing sets addresses them. Multidimensional selectivity estimation for conjunctive queries has been an active area of research for many years [JKNS99,MD98,PI97,PIHS96,MPS99,WVI97]. Most work in this area, however, has focused on numerical attributes and has assumed the existence of a mapping from multidimensional categorical data to fully ordered domains. In the case of multidimensional string queries, such mapping is of no use. If one sorts the strings lexicographically, substrings are not necessarily ordered. Even if one is able to overcome this limitation by explicitly representing

[1] Prefix and suffix constraints can be reduced to a substring constraint, as explained in [JKNS99].

all substrings and approximating the frequency of individual points in the multidimensional space using standard techniques, the domain would be so large and the frequencies so small as to render these techniques impractical. An alternative would be to adopt an end biased histogram [PIHS96] approach and keep only the substrings with the highest counts, subject to constraints on space and approximate the other substrings assuming uniformity. Since the total number of substrings is very large, this approach would be very close in accuracy to one that makes the uniformity assumption, which is highly inaccurate. Furthermore, the situation is aggravated as the dimensionality increases.

New approaches for selectivity estimation tailored to the string domain have been developed in recent years [JKNS99,JNS99,KVI96,WVI97]. We can abstract the following common framework in existing approaches. A precomputation is done to store the number of tuples that contain most frequently co-occurring substrings; we call these the *cross-counts*, to be described later with examples. Online estimation involves parsing the query into subqueries such that the cross-count for the subqueries is available from the precomputation. The effectiveness of a particular approach within this framework relies on judicious utilization of the cross-counts. However, two problems exist. First, the number of cross-counts stored by known methods increases exponentially with the number of dimensions (a "space dimensionality explosion") due to the need to capture the correlation amongst the dimensions. Hence, given limited amount of space, none of previous methods can give accurate estimation. Second, existing methods also cannot be easily and gracefully generalized to selectivity estimation for Boolean queries, because the number of possible correlations for Boolean queries is super-exponential.

This paper is organized in two main parts. In the first part, we present a novel approach to the problem of multidimensional substring selectivity estimation for conjunctive queries, which makes the storage and representation of cross-counts more space-efficient, resulting in better parsing and, therefore, substantially better accuracy. Our basic premise is that exact values of cross-counts are not needed; rather, reasonable approximations thereof may do just as well. Therefore, if such an approximation may be obtained using small space, for a given budget of space, the number of cross-counts stored by our approach will be significantly larger than that of the existing methods. This in turn will lead to more accurate selectivity estimation as we observe. To bring this line of reasoning to fruition, we develop a new approach for multidimensional string selectivity estimation. At the heart of our proposal is a Monte Carlo technique called *set hashing* to succinctly represent the set of tuples containing a given substring as a signature; then, any combination of signatures from the cross product can be intersected to form an approximate cross-count. Using only linear storage, these one-dimensional signatures can be used to approximate a large number multidimensional cross-counts. Thus, whereas existing methods *explicitly* keep statistics to account for the correlation among the attributes using cross-count values, our approach uses set hashing to *implicitly* capture the correlation among the attributes. We perform an extensive set of experiments with real data sets. Results show that for two-dimensional selectivity estimation, our approach is an order of magnitude more accurate than existing methods. The improvements are much larger for higher-dimensional substring selectivity estimation.

In the second part of this paper, we initiate the formal study of the selectivity estimation problem on Boolean substring predicates; to our knowledge, this problem has not been

investigated before. In particular, we formalize two variants of the problem, one with a full index structure on the strings (a suffix tree, in our case), and the other with only a pruned structure. For the problem of substring selectivity estimation that has been studied in the literature, the full suffix tree variant is rather trivial, and is of no interest (in fact, exact selectivity can be determined by walking down the suffix tree with the query substring). However, for Boolean queries, the problem is challenging even with the full suffix tree because one needs to capture the correlation amongst the occurrence of substring predicates as specified by the Boolean relation in the query. The set of all possible Boolean relations amongst the substring predicates, and hence the space of possible correlations, is prohibitively large to explicitly compute, or store. We employ set hashing to succinctly represent the set of strings containing a given substring as a signature vector of hash values. Correlation estimates among substring predicates can then be generated by performing algebraic and set operations on these signatures. Our main technical contribution is a fast algorithm for estimating the selectivity of *any* Boolean query using the set hashing approach for both variants of the problem. In practice, Boolean queries tend to be of small length (that is, few clauses, each containing few substring predicates). For example, 84% of the queries issued at AltaVista involve less than four keywords [SHM98]. Our selectivity estimation algorithms are very effective for such cases.

1.2.3. Map of the paper

The structure of the paper is as follows. Section 2 gives necessary background. We present our approach for the problems of concern in this paper in Section 3. Section 4 presents a thorough experimental evaluation of our algorithms. Finally Section 5 concludes the paper.

2. Preliminaries

In this section, we will formally define the multidimensional selectivity estimation problem with conjunctions and the problem of estimating the selectivity of Boolean substring queries. The problem of selectivity estimation for multidimensional conjunctive queries has significant history; we review the known methods in Section 2.2 and make some key observations in Section 2.2.4. In Section 2.3, we present and defend the basic premise of our work that it suffices to keep certain approximate counts for solving these problems.

2.1. Problem definitions

Let Σ be the alphabet. We denote by Σ^* the set of strings of finite length on Σ. The string $\sigma \in \Sigma^*$ is said to be a substring of $s \in \Sigma^*$ if, for some $\alpha, \beta \in \Sigma^*$, $s = \alpha\sigma\beta$. We shall refer to σ as a *substring predicate* when it is compared with a string s to test if σ is a substring of s. A k-dimensional string $\mathbf{s}$ is a k-tuple $(s_1, \ldots, s_k)$ where each $s_i \in \Sigma^*$, for $1 \leqslant i \leqslant k$; the string $\sigma = (\sigma_1, \ldots, \sigma_k)$ is said to be a substring of $\mathbf{s}$ if, for some $\alpha_i, \beta_i \in \Sigma^*$, $s_i = \alpha_i\sigma_i\beta_i$, for $1 \leqslant i \leqslant k$.

We are given a database $T = \{t = (t_1, \ldots, t_k) \mid t_i \in \Sigma^*\}$ of tuples t on k string attributes.[2] An example of our input is a customer database in which each customer is a tuple with string

[2] There may be other non-string attributes in the database, but they do not concern us here.

attributes `surname`, `address`, and `telno`. A k-dimensional query q is given by $q = (q_1, \ldots, q_k)$ where each $q_i \in \Sigma^*$, and the goal is to determine the fraction of tuples in the database for which q_i is a substring of t for each attribute i.[3] The *multidimensional substring selectivity* problem involves preprocessing the database so that estimation queries can be answered quickly online. Any practical selectivity estimation method should provide acceptable accuracy (say, less than 20%). Also, the data structure used for this purpose must be limited to a small percentage (say, 1%) of the input size.

Now we introduce the more general problem of Boolean query selectivity estimation. To simplify notations, we assume there is only one attribute in the database t in this section. However, our techniques are applicable to databases having multiple attributes and results on such databases are reported in Section 4.

A Boolean expression over substring predicates is defined recursively as follows:

- Any substring predicate $\sigma \in \Sigma^*$ is a Boolean expression.
- If p and q are Boolean expressions, then so are $(p \wedge q)$, $(p \vee q)$, and $\neg p$; here, $\wedge$, $\vee$ and $\neg$ are the well known logical operators `AND`, `OR` and `NOT`, respectively.

There are no Boolean expressions over Σ^* other than those derived from these rules. A Boolean expression is said to be in *conjunctive normal form* (CNF) if it is composed of a conjunction of clauses, where each clause contains a disjunction of predicates, more formally, $(\sigma_{11} \vee \cdots \vee \sigma_{1\ell_1}) \wedge \cdots \wedge (\sigma_{k1} \vee \cdots \vee \sigma_{k\ell_k})$. A Boolean expression is said to be in *disjunctive normal form* (DNF) if it is composed of a disjunction of clauses, where each clause contains a conjunction of predicates, more formally, $(\sigma_{11} \wedge \cdots \wedge \sigma_{1\ell_1}) \vee \cdots \vee (\sigma_{k1} \wedge \cdots \wedge \sigma_{k\ell_k})$.

The Boolean query selectivity estimation problem is as follows. We are given a set of strings $S = \{s \mid s \in \Sigma^*\}$. (An example of our input is a string attribute in a relational database; alternatively, each string could be an HTML document from the Web.) The goal is to determine the fraction of strings in S for which any query q, which is a Boolean expression specified at runtime, evaluates to true; this fraction is denoted $P(q)$. The problem is to preprocess the set S of strings so that online estimates of $P(q)$ can be obtained for any q.

Obviously, any practical selectivity estimation method should provide acceptable accuracy. Also, estimation must be significantly faster than solving the problem exactly, that is, obtaining the precise number of strings that satisfy the Boolean query.

There are two variants of our problem depending on the amount of storage space that may be available. In the first variant, we are allowed space linear in the size of the set S, that is, $O(\sum_{s \in S} |s|)$. Hence, we can build a standard string indexing structure such as the suffix tree which is the trie of all suffices of all strings in S.[4] In the second variant, we are allowed a smaller amount of space than in the first variant, in particular, space sub-linear in the size of S. This would entail *pruning* the suffix tree by keeping only an appropriately sized part of the suffix tree. We refer to the first variant as the *full suffix tree* (FST) case, and the second as the *pruned suffix tree* (PST) case. We embark into our discussion, by first presenting algorithms for the FST case. We then

[3] Since a fraction of tuples translates to probability, throughout this paper we shall adopt the output form to be a probability.

[4] Technically, this is known as the generalized suffix tree when we have more than one string in S, a distinction we do not make here.

present the PST case. In both case, we employ set hashing to capture interesting correlations between string identifiers in the data set.

2.2. Existing methods for multidimensional conjunctive queries

We first review the known methods for conjunction-only multidimensional substring selectivity problem in some detail since we will be able to abstract a common framework and to extend them. In Section 4.1, we compare all such methods with our approach.

2.2.1. The WVI method

The method proposed in [WVI97], called the WVI method henceforth, is based on a variant of suffix trees called *count-suffix trees* wherein each node is augmented with the count of occurrences of its associated substring. The method maintains k suffix trees (one on each dimension), and a matrix storing the *cross-counts*, that is, the number of occurrences of all substring combinations from each dimension. In order to limit the space used, a pruning threshold is chosen and the suffix trees are pruned to obtain *pruned suffix trees* (PSTs), each PST having at most m nodes (for simplicity, here we assume the PST in each dimension has the same number of nodes). The matrix of cross-counts then has $O(m^k)$ entries. The details of this pruning can be found in [WVI97]. Fig. 1 illustrates the structures built from a two-dimensional toy data set in which the cut lines indicate pruning.

WVI uses the so-called *greedy parsing* from [KVI96] applied independently on each dimension of the query q. In the ith dimension, q_i is broken into a sequence of mutually disjoint pieces such that $q_i = q_i(0)\cdots q_i(l_i)$; each $q_i(j),\ 0 \leqslant j \leqslant l_i$ is the longest possible PST match possible from the current position. For each combination $c_i, 1 \leqslant i \leqslant \prod_{j=1}^{k} l_j$ of these pieces from each dimension, their cross-count is obtained from the stored matrix. These cross-counts are "combined" based on the independence assumption which resolves to multiplying the selectivities of the pieces: $\prod_{1 \leqslant i \leqslant \prod_{j=1}^{k} l_j} Pr(c_i)$. We refer to the use of greedy parsing followed by the application of the independence assumption as `ID` parsing. Some variations of the basic ID parsing, such as child and depth-based strategies, are proposed in [WVI97].

Example. Given the pruned structures of Fig. 1 and $q =$ `(abc,123)`, `abc` is parsed into pieces `ab` and `c`; `123` is parsed into pieces `12` and `3`. The subqueries resulting from greedy parsing come from the cross-product of the pieces: `(ab,12)`, `(c,12)`, `(ab,3)` and `(c,3)`, as illustrated in Fig. 3(a). The associated subquery selectivities are then multiplied based on the independence assumption

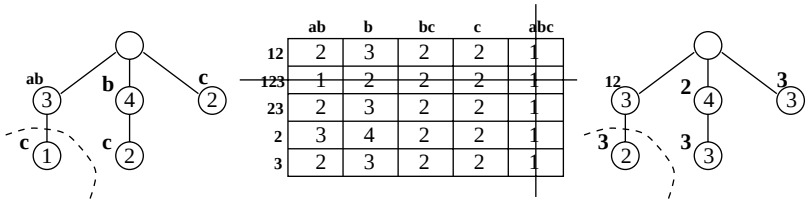


Fig. 1. Pruned structures for WVI method for the following toy data set: `(ab,12)`, `(abc,123)`, `(bc,123)`, `(ab,23)`.

as follows:

$$Pr\{(\texttt{abc},\texttt{123})\} = Pr\{(\texttt{ab},\texttt{12})\} \times Pr\{(\texttt{c},\texttt{12})\} \times Pr\{(\texttt{ab},\texttt{3})\} \times Pr\{(\texttt{c},\texttt{3})\} = \tfrac{2}{4} \times \tfrac{2}{4} \times \tfrac{2}{4} \times \tfrac{2}{4} = \tfrac{1}{16}.$$

2.2.2. The KD method

The method proposed in [JKNS99] is based on a novel generalization of suffix trees they call k-D *count-suffix trees* in which each node corresponds to substring combinations from each dimension and are augmented with counts. Hence we call this the KD method. To capture all the combinations would require space exponential in k, but only a pruned data structure is maintained. Fig. 2 illustrates the resulting structure on the 2-D toy data set in which the cut line indicates pruning.

KD parses a query q using the *maximal overlap* principle from [JNS99]. In a greedy fashion, q is broken into subqueries q'_j overlapping in multiple dimensions, where each q'_j is the match in the tree that overlaps q'_{j-1} the most. The cross-count for each such subquery is obtained by a simple lookup in the k-D tree for the node associated with the subquery. These cross-counts are then combined using conditional probabilities of the subqueries based on the inclusion–exclusion principle from set theory. We refer to the use of maximal overlap followed by conditioning as `MO` parsing.

Example. Given the pruned structure of Fig. 2 and $q = (\texttt{abc},\texttt{123})$, Fig. 3(b) illustrates `MO` parsing. Each rectangle represents a subquery. Rectangle I represents (ab,12), II represents (bc,123), and

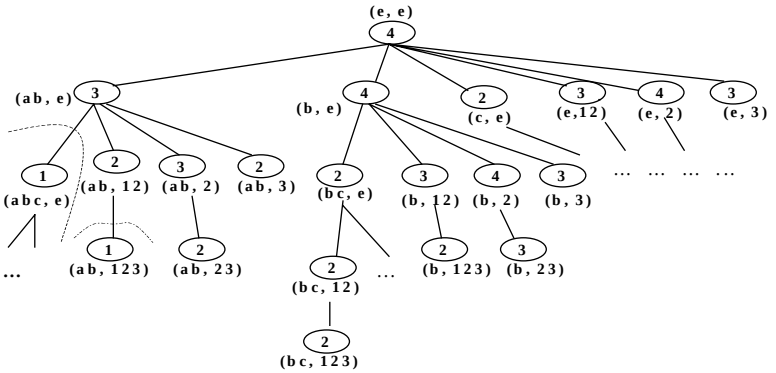


Fig. 2. Pruned structures for JKNS method on the 2-D example.

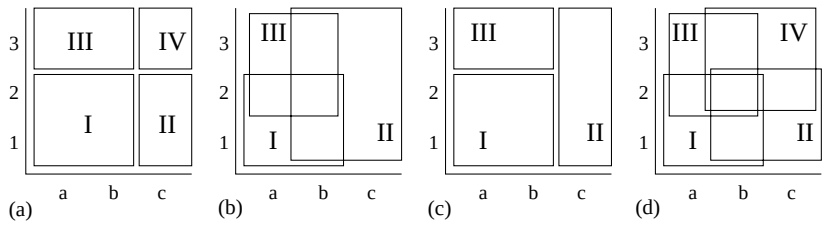


Fig. 3. Resulting parse of query `(abc,123)` on the 2-D example via (a) `ID`; (b) `MO`; (c) `GNO`; and (d) `WVI–MO`.

III represents (ab,23). The subquery selectivities are multiplied based on the inclusion–exclusion formula as follows:

$$\begin{aligned} Pr\{(\mathtt{abc},\mathtt{123})\} &= \frac{Pr\{I\} \times Pr\{II\} \times Pr\{III\} \times Pr\{I \cap II \cap III\}}{Pr\{I \cap II\} \times Pr\{I \cap III\} \times Pr\{II \cap III\}} \\ &= \frac{Pr\{(\mathtt{ab},\mathtt{12})\} \times Pr\{(\mathtt{ab},\mathtt{23})\} \times Pr\{(\mathtt{bc},\mathtt{123})\} \times Pr\{(\mathtt{b},\mathtt{2})\}}{Pr\{(\mathtt{b},\mathtt{12})\} \times Pr\{(\mathtt{ab},\mathtt{2})\} \times Pr\{(\mathtt{b},\mathtt{23})\}} \\ &= \frac{\frac{2}{4} \times \frac{2}{4} \times \frac{2}{4} \times 1}{\frac{3}{4} \times \frac{3}{4} \times \frac{3}{4}} = \frac{8}{27}. \end{aligned}$$

2.2.3. A common framework

Both WVI and KD methods have a common framework with the following steps:

1. *Preprocessing.* A compact structure is constructed by these methods to store cross-counts of substring occurrences in the database, employing pruning techniques to reduce the size.
2. *Query parsing.* A query q is parsed into smaller subqueries q'_j that match nodes in the pruned structure, and have associated cross-counts.
3. *Cross-count lookup.* The count c_j associated with each q'_j, that is, the number of times q'_j occurs in the database, is determined from the stored structure.
4. *Probabilistic estimation.* A probabilistic formula is used to algebraically "combine" the c_j's to derive the selectivity estimate $Pr(q)$.

For example, in the WVI method, preprocessing builds PSTs and the cross-count matrix, parsing is greedy, cross-count lookup involves matrix look-up, and estimation is a product operation. In KD, preprocessing builds a pruned k-D count suffix tree, parsing is based on maximal overlap, cross-count lookup involves traversing the tree, and estimation is a more sophisticated product operation given by inclusion–exclusion. As mentioned before, greedy parsing followed by use of independence assumption is called *ID* parsing, and the use of maximal overlap followed by conditioning is MO parsing. Thus the previous methods can be further defined as WVI–ID and KD–MO in our terminology.

A simple observation that emerges from the description of the common framework is that different structures and parsing strategies can be combined. For example, the KD method may be combined with ID parsing (instead of MO parsing). This KD-ID method was studied in [JKNS99], in which it was called greedy non-overlap (GNO). Fig. 3(c) illustrates GNO parsing on the 2-D example. Similarly, the WVI method may be combined with MO parsing, rather than ID parsing, which we refer as WVI-MO. Fig. 3(d) illustrates WVI–MO parsing on the 2-D example; it has not been previously studied in the literature.

2.2.4. Observations on existing methods

Current multidimensional substring selectivity methods suffer from a "space dimensionality explosion". Both WVI and KD methods consume an exponential amount of space for maintaining cross-counts (WVI method does this in a $\Theta(N^k)$ sized table, KD method in a $\Theta(N^k)$ sized tree structure.) In order to cope with space constraints, they apply pruning. In doing so, they

only maintain cross-counts for combinations of short substrings; as the number of dimensions increases, these substrings get significantly shorter. As a result, the probability of locating any given subquery is small and queries wind up being parsed into many small subqueries. Thus, known methods end up relying on probabilistic estimation for their overall accuracy. This proves to be inaccurate as we will show.

2.3. Approximating cross-count values suffices

The basic premise that led to our work here is that it is not necessary to have exact cross-count values for accurate multidimensional selectivity estimation; rather, approximate values can yield accurate selectivity estimates. This sounds plausible since, in any case, the whole process involves probabilistic estimation using in exact assumptions such as independence or truncated inclusion–exclusion. However, it is not clear a priori that the errors in approximating the cross-counts will not compound the inherent errors in using probabilistic estimation formulas. In this section, we present a simple experiment to study the interaction between approximating cross-count values and overall estimation error. We performed many other experiments (with different data sets, error models, etc.) and our conclusions coincided. Nevertheless, the following description should be taken only as an experimental observation for purposes of intuition, and not as an attempted proof.

We tested our hypothesis that approximating cross-count values does not significantly affect multidimensional selectivity estimation. We used a real AT&T data set of 20 K tuples and two attributes, and tested the existing KD–MO method. We constructed a pruned k-D count-suffix tree, and then perturbed the cross-count value μ in each node with noise; our noise model makes the standard assumption that the error is independently and identically distributed as a Gaussian around the exact value with standard deviation σ i.e., for a cross-count value μ, we replace it with a value $\mu' = \mu \times (1 + r)$, where r is a random number following Gaussian distribution with mean zero and with a certain standard deviation σ (We also make sure μ' is greater than zero). We issued 1000 2-D substring queries randomly sampled from within strings of the data set. The tuples from which these substrings come were chosen uniformly; then, for each attribute, the start position of substrings was chosen uniformly, and the length uniformly varied between 2 and 7 characters.

Fig. 4 reports the average absolute relative estimation error as a function of storage space, that is,

$$E_{abs} = \frac{1}{N} \sum_{q \in Q} \frac{|S_q - S'_q|}{S_q},$$

where S_q and S'_q are the actual and the estimated selectivities, respectively, and Q is the set of queries. To study the effect of noise, we increased the variance of the error from 10% to 500%.

Two observations emerge. First, after implementing them, we learned that previous methods require up to 200% of the database size to achieve less than 50% estimation error. With this space consumption, for this data set, they can tolerate a very large amount of error (up to 500% variance) in approximating cross-counts and yet be just as effective in overall accuracy. In other words, even if individual cross-counts are off by a factor of 6 on average, the estimation process does not produce significantly larger total error. The existing methods do not leverage off this

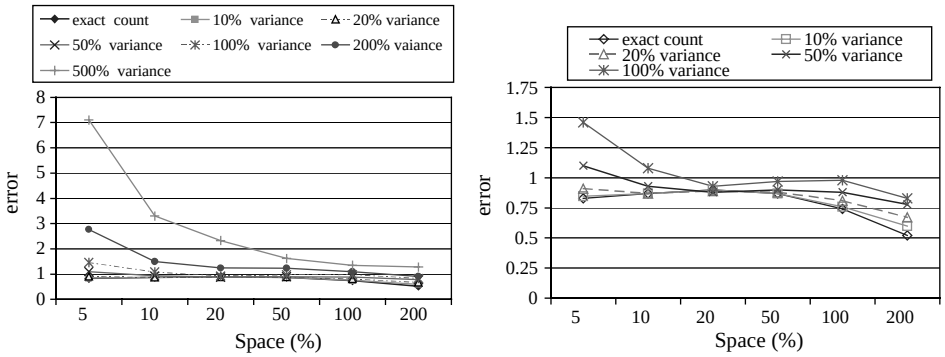


Fig. 4. Effect on accuracy of perturbing cross-counts with Gaussian noise as space is increased. The space is measured as percentage of database size. On the right is an enlarged version of the curves with variance less than or equal to 100%.

flexibility that is available. The second observation is that if one wishes to use only a reasonable amount of space for estimation (say, less than 10% of the database size), then up to 50% variance in cross-count values may be tolerated; further variance does significantly affect overall accuracy. This is relevant to us since we wish to devise methods that use small amount of space.

3. Our approach

In this section we describe our proposed solutions for estimating the selectivity of multidimensional conjunctive and Boolean queries.

3.1. Proposed solution for multidimensional conjunction queries

Our goal is to exploit the observation in the previous section that to perform accurate multidimensional selectivity estimation, it suffices to approximate the cross-counts. The overall reasoning is that if cross-counts can be approximated using small space, then for a given budget of space, one can store information about more combinations of substrings than is currently done. Equivalently, substring combinations that have precomputed cross-counts will be longer than in existing methods. In turn, this will lead to longer parses, less reliance on probabilistic estimation formulas and more accurate selectivity estimation. For this line of reasoning to come to fruition, one needs a reasonably accurate estimate of cross-counts using small space. Of course, one may consider doing away with the cross-counts approach altogether to circumvent a "space dimensionality explosion". Unfortunately, this would rely even more on oversimplified probabilistic assumptions, such as independence between attributes, and was shown to be bad in [JKNS99].

There are some straightforward suggestions for "packing" more cross-counts into small space. Consider the WVI method. Here one may apply standard signal processing techniques such as truncating the smallest cross-count matrix values, or quantizing them via histograms or significant bits. Indeed, this sort of approach was tried unsuccessfully in [WVI97] using depth-based

averages. In the KD method, one may build disjoint pieces of k-D suffix trees on significant portions, or have edges spanning longer paths so insignificant intermediate nodes are omitted, etc. In analyzing these suggestions, our conclusion is that they do not exploit the full structure in the relationship between different cross-counts, do not generalize gracefully to high dimensions, and do not have the potential to provide the substantial improvements that we seek. In what follows, we will describe our set-based approach which we believe is elegant. This approach will result in only a linear amount of storage (linear in the pruned structure), but nevertheless be able to approximate a large number of cross-counts.

3.1.1. Sketch of our approach

We adopt a set-oriented approach. For exposition, let us first ignore any space constraints and assume that an unpruned suffix tree has been built independently for the substrings in each dimension. We denote the string labelling the jth node (assuming some ordering of nodes in a suffix tree) of the ith suffix tree as w_{ij}. This is the string that spells the path from the root of the ith suffix tree to its jth node. We augment each node j in each suffix tree i with set S_{ij}, where S_{ij} contains the row identifiers (RIDs) of each tuple containing w_{ij} as a substring in the ith dimension. Fig. 5 presents an example in two dimensions. For the sample data set shown in Fig. 1, we have created two suffix trees, one for each dimension; each node is augmented with a set containing the RIDs in which each substring occurs in the corresponding dimension.

The selectivity of any substring query can then be computed (without error) by intersecting the S_{ij}'s corresponding to each dimension. For example, let `(abc, 123)` be a 2-D substring query. We locate the nodes j and k in each tree such that $w_{1j} = \texttt{abc}$ and $w_{2k} = \texttt{123}$. Then $\frac{|S_{1j} \cap S_{2k}|}{N}$ gives the selectivity.

The problem is that we do not have the space to store the sets associated with nodes. The size of any set can be as large as N, the number of tuples in the database. Pruning is not likely to be helpful to cope with space limitations because very few nodes on which to base estimations will remain. Our key idea is to employ the Monte Carlo technique of *min-wise independent permutations* [BCFM98,Coh97]. This technique provides an unbiased estimator of the resemblance of two sets, which is the ratio of the size of the intersection over the size of the union of the sets. Min-wise independent permutations employ a collection of hash functions to obtain a compacted representation of the sets. We use hash functions to store the sets associated with each node. The hash functions will serve two purposes. First, the hash value of a set can be

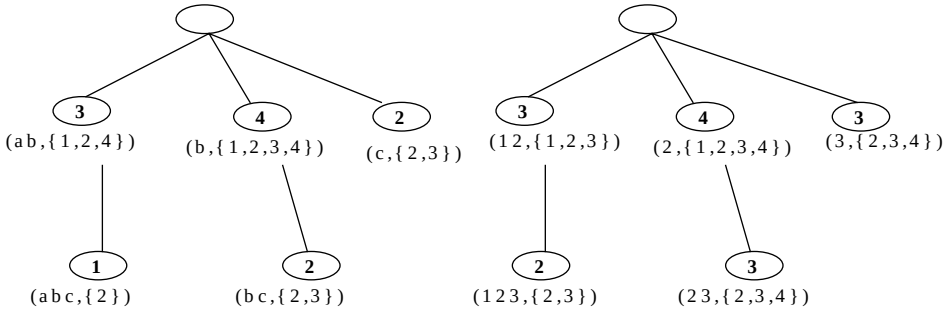


Fig. 5. Set-oriented approach illustrated with a 2-D example.

stored in much smaller space than the set itself (typically 100 to 400 bytes). Second, the hash functions will serve to compute the size of the intersection of any two (more generally, k) sets by employing the technique of set hashing that we introduce. That will provide us the cross-counts needed to apply the probabilistic estimation formulas. The catch, however, is that no hash functions exist that both use small space (significantly less than N) *and* provide exact count of the intersection of any two sets.[5] Hence, we must settle for hash functions that allow some approximation in estimating the intersection sizes. This approach will only use $O(mk)$ space, where m the number of nodes in each PST, which is linear in the size of PSTs. We will show such hash functions in the next section.

To summarize, the sketch of our approach is as follows. We construct PSTs on each dimension and maintain the sets associated with the nodes using their hashed values. From any arbitrary combination of substrings from different dimensions, we can approximate their cross-count by estimating their set intersection cardinality using hashed values. The trade-off here is that a much larger number of cross-counts can be reconstructed than can be stored using previous approaches at the loss of accuracy of these counts. Any query is parsed as before, and the required cross-counts are generated from the set hashes. Finally, probabilistic estimation formulas are employed as before to obtain the overall selectivity. We will now describe the details of the set hash as well as the overall algorithm.

3.1.2. *Introduction to set resemblance estimation*

Min-wise independent permutation is a well known Monte Carlo technique which can be used as an unbiased estimator of the *set resemblance* (denoted ρ) of two sets A and B, that is,

$$\rho = \frac{|A \cap B|}{|B \cup B|},$$

where, for a set S, the notation $|S|$ represents its cardinality. It was introduced by Cohen [Coh97] and Broder et al. [BCFM98]. This technique has been used for finding Web page duplicates [Bro98], for data mining [CDF+00], and for estimating the size of transitive closure [Coh97].

The intuition is as follows. Suppose that "darts" are thrown randomly at the universe U. If two sets A and B have high resemblance, then it is likely that the first dart to hit A will *simultaneously* hit B; for low resemblance, the converse is true. Fig. 6 illustrates this concept. The resemblance can be estimated by Monte Carlo simulation as follows. Say there are ℓ random and independent sequences of dart throws, where in each sequence darts are thrown until a set element is hit. For each sequence, we record the element of set A that is hit in a signature vector S_A; we independently do the same for B to generate S_B, using *the same sequence*. The resemblance ρ can be estimated by dividing the number of darts hitting both A and B by the number hitting A or B (or both), where the numerator is equivalent to the number of matching components in signatures S_A and S_B and the denominator is equivalent to ℓ. Note that the (joint) resemblance can be estimated by two signature vectors which are created *independently*, since the numerator can be determined on-the-fly from the signature vectors.

We proceed with a more detailed description of how to generate the signature S_A from a set A. For each signature vector component, we randomly permute the elements of U from which the

[5] This has been formalized and proven in the area of Communication Complexity [Lov90].

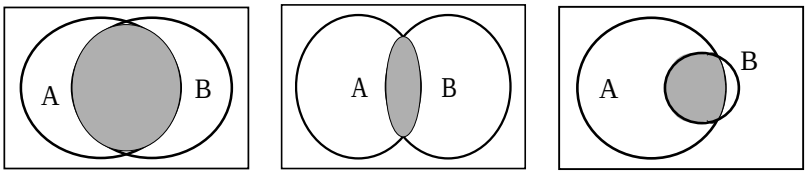


Fig. 6. The main idea behind set hashing: (a) high overlap, high resemblance; (b) low overlap, low resemblance; (c) high overlap, low resemblance.

sets are drawn and record the value of the first element of the permutation which is also an element of A. More formally, let $U = \{1, \ldots, n\}$ and let π be chosen uniformly at random over the set of permutations of U. Define $\min\{\pi(A)\} = \min\{\pi(x) | x \in A\}$. Then

$$Pr(\min\{\pi(A)\} = \min\{\pi(B)\}) = \frac{|A \cap B|}{|A \cup B|}.$$

Note that the *same* permutation is used for both sets A and B. Then for a given set A, the signature vector is generated from permutations $\pi_1, \ldots, \pi_\ell$.

$$S_A = (\min\{\pi_1(A)\}, \min\{\pi_2(A)\}, \ldots, \min\{\pi_\ell(A)\}).$$

To estimate the set resemblance, we divide the total component-wise matches by the signature length. More formally,

$$\hat{\rho} = \frac{|\{i | 1 \leqslant i \leqslant \ell, \min\{\pi_i(A)\} = \min\{\pi_i(B)\}\}|}{\ell}.$$

Theorem 1 (Cohen [Coh97]). *Under this approximation, if ρ is the exact resemblance and ℓ is the length of the signature, then there exists a small constant c such that, for any $0 < \varepsilon < 1$,*

$$Pr(|\hat{\rho} - \rho| \geqslant \varepsilon\rho) \leqslant \mathrm{e}^{-c\varepsilon^2\ell},$$

where e *the base of the natural logarithm. That is, the probability that the estimate $\hat{\rho}$ differs from ρ is exponentially decreasing with ℓ.*

For our purposes, we would eventually need the set resemblance between not just two sets, but rather k sets. Defined more generally,

$$\rho_k = \frac{|A_1 \cap \cdots \cap A_k|}{|A_1 \cup \cdots \cup A_k|}.$$

The ρ we have been discussing so far is the same as ρ_2. Our procedure for estimating ρ_2 can be generalized to estimate ρ_k in a natural way. In particular, $\hat{\rho_k}$, an estimate for ρ_k, can be obtained using the formula

$$\hat{\rho_k} = \frac{|\{i | 1 \leqslant i \leqslant \ell, \min\{\pi_i(A_1)\} = \min\{\pi_i(A_2)\} = \cdots = \min\{\pi_i(A_k)\}\}|}{\ell},$$

where ℓ, as before, is the number of permutations used. We can also generalize the proof from [Coh97] to prove that $\hat{\rho}_k$ is a good estimator for ρ_k for sufficiently large ℓ. The proof is not more instructive for general k than it is for $k = 2$, so we have omitted the proof. It suffices to convince oneself of the following observation for any i:

$$Pr(\min\{\pi_i(A_1)\} = \min\{\pi_i(A_2)\} = \cdots = \min\{\pi_i(A_k)\}) = \rho_k.$$

3.1.3. Estimating intersection size via set hashing

Min-wise independent permutations can be used to estimate the resemblance of two sets; our goal however is to estimate the size of the intersection of two sets. We now introduce set hashing, a technique that uses the resemblance of two sets, to estimate the size of their intersection. The intersection size of two sets A and B, $|A \cap B|$ can be estimated as follows:

Observation 1. *We have*

$$|A \cap B| = \frac{\rho(|A| + |B|)}{1 + \rho}$$

where ρ is set resemblance between A and B.

Proof. We have $|A \cup B| = |A| + |B| - |A \cap B|$ from basic set theory. Dividing both sides by $|A \cap B|$ and rearranging the terms gives us the formula. □

It is important to note that the above intersection sizes do not come with the probabilistic guarantees mentioned in the previous section. However, we show later in our experiments in Section 4.1 that our estimates of the intersection size suffice to provide good estimation for most queries. The natural generalization of the formula above in general would involve many terms because of inclusion–exclusion principle, and we will not be able to obtain all such terms easily. (The readers can convince themselves by considering the case $k = 3$.) In what follows, we provide an alternate procedure that works in general for any k-way intersection, and is quite simple. It relies on the observation below.

Theorem 2. *The signature $S_{A_1 \cup \cdots \cup A_k}$ of $A_1 \cup \cdots \cup A_k$ can be computed from the signature S_{A_j} for sets A_j as follows. For any $i, 1 \leqslant i \leqslant \ell$,*

$$S_{A_1 \cup \cdots \cup A_k}[i] = \min\{S_{A_1}[i], \ldots, S_{A_k}[i]\}.$$

That is, for each component, we choose the minimal value of the signatures of all the sets in that component.

Proof. The proof is straightforward. When computing each signature component of $A_1 \cup \cdots \cup A_k$, $\min\{\pi_i(A_1 \cup \cdots \cup A_k)\} = \min\{\min \pi_i(A_1), \ldots, \min \pi_i(A_k)\}$. □

Our procedure for estimating $|A_1 \cap A_2 \cap \cdots \cap A_k|$ is as follows:

Step 1. Say A_j has the largest size of all A_i's.[6] We first calculate $S_{A_1 \cup \cdots \cup A_k}$ as described above. Using this signature, we estimate

$$\gamma = \frac{|A_j|}{|A_1 \cup \cdots \cup A_k|}$$

using our method for estimating resemblance on S_{A_j} and $S_{A_1 \cup \cdots \cup A_k}$.

Step 2. We use the following formula:

$$|A_1 \cap A_2 \cap \cdots \cap A_k| = \rho_k |A_1 \cup \cdots \cup A_k| = \frac{\rho_k |A_j|}{\gamma}.$$

We can estimate ρ_k as described in the previous section, γ as in previous step, and $|A_j|$'s are maintained explicitly in our application. That completes the description of the overall method for estimating $|A_1 \cap \cdots \cap A_k|$.

3.1.4. Practical implementation of set hashing

Set hashing uses min-wise permutations to derive the intersection size estimate. Implementation of min-wise permutations requires generation of random permutations of a universe. Efficiently permuting the elements of the universe is impractical. In practice, for each signature vector component, we independently seed a hash function and generate the hash image $h(a)$ of each element $a \in A$; the minimum $h(a)$ is recorded in the signature. Unfortunately, as reported in [BCFM98], there is no tractable class of hash functions which guarantees equal likelihood for any element to be chosen as the minimum element of a permutation (*aka* min-wise independence); this property is needed in order to properly use hashing to simulate permutations. However, we use linear hash functions because they turn out to be good enough in practice [Bro98] and it is easy to generate a number of different independent hash functions. Of course, each hash function h should be chosen so that the probability of collisions are low. To reduce collisions, we design h to map elements into a range that is significantly larger than the domain. The hash function we used is $h(x)$: $((x + \texttt{seed}) \times y + z) \bmod s$, where y, z are large prime numbers, `seed` is an arbitrary random number, and s is the hash space size. When we vary the value of `seed`, we can generate different hash functions used to generate different signature components. See Section 4.1.5 for an experimental evaluation of how different parameters of hashing affect the accuracy of selectivity estimation based on set hashing.

3.1.5. High level description of our approach

Formally, the steps are as follows.

1. *Preprocessing.* We construct a PST for each dimension applying known pruning strategies in [JKNS99,WVI97]. With each node in the PST, we store the exact count of the number of row ID's of that substring (labelling the path from the root to that node). We also store the signature of the set of all tuples which contain that substring.
2. *Query parsing.* A query q is parsed on each dimension i independently into a number l_i of smaller subqueries $q_i(m)$, $0 \leqslant m \leqslant l_i$ that match nodes in the PST of ith dimension.

[6] This is only a technicality. Any one of the sets will do, but the largest gives the best accuracy.

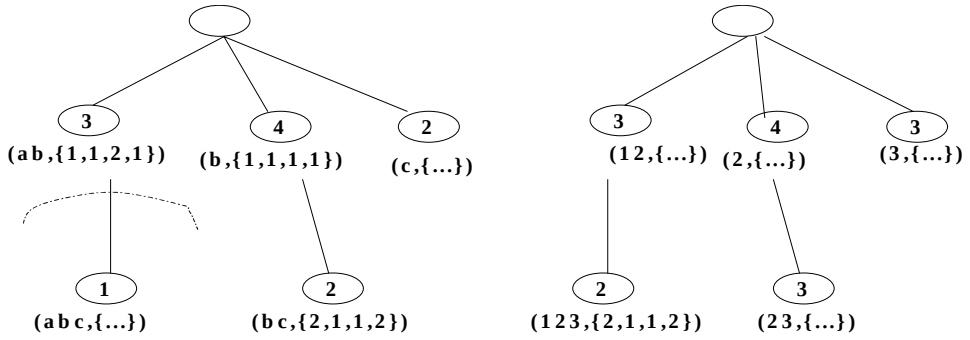


Fig. 7. Pruned structures for set hashing method on the toy data set. The signatures for some nodes are also shown.

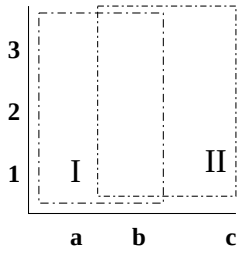


Fig. 8. Resulting parse of query (abc,123) on the 2-D example via set hashing with MO. The dashed boxes indicate approximate counts generated by set-hashes for the associated subquery.

3. *Cross-count lookup.* For each combination c_j of these subqueries from each dimension, the associated cross-count is generated using set hashing techniques from Section 3.1.3. For MO parsing, the cross-counts for overlaps of c_j's are also computed.
4. *Probabilistic estimation.* Using a probabilistic formula, we algebraically "combine" the c_j's to derive the selectivity estimate $Pr(q)$.

We have left out certain details, e.g., how to construct the PSTs without materializing the entire suffix tree in any dimension, detailed description of pruning techniques, etc. These details only involve adopting existing techniques from [JKNS99,WVI97].

Example. We now revisit the 2-D example from Section 2.2. Fig. 7 shows the two suffix trees generated by preprocessing, where the cut line indicates pruning. Suppose the query $q = (\mathtt{abc}, \mathtt{123})$. Then, MO parses abc into ab and bc, and 123 into 123, as illustrated in Fig. 8, where the dashed boxes indicate approximate counts. In the lookup step, the cross-counts for (ab, 123), (bc, 123), and (b, 123) are generated via set hashing. Suppose the exact cross-counts are 1/4, 1/2, and 1/2, and that the approximate values 0.27, 0.44, and 0.63 are generated, respectively. Then MO method estimates the selectivity as follows:

$$Pr\{(\mathtt{abc}, \mathtt{123})\} = \frac{Pr\{(\mathtt{ab}, \mathtt{123})\} \times Pr\{(\mathtt{bc}, \mathtt{123})\}}{Pr\{(\mathtt{b}, \mathtt{123})\}} = \frac{0.27 \times 0.44}{0.63} = 0.1886.$$

3.2. Proposed solution for Boolean queries using a full suffix tree (FST)

While there has been some recent work on substring selectivity estimation, the more general problem of selectivity estimation on Boolean substring predicates has not been studied. Of course, the special case of conjunctive queries on k keywords (i.e., $\sigma_1 \wedge \cdots \wedge \sigma_k$) can be mapped into k-dimensional substring queries (over k replicated attributes) and, hence, the selectivity can be estimated by any of the previous multidimensional substring selectivity estimation techniques. However, this special case approach is limited to queries with *exactly* k substring predicates, where k is known *a priori*; it is not clear how Boolean expressions over $(k+1)$ keywords can be handled by the same data structure built to handle queries containing k keywords (e.g., a k-D suffix tree). Furthermore, it is not clear how to extend this framework to handle disjunctions and negations.

A straightforward approach that enables the previous substring selectivity methods to be applied to Boolean queries is to assume independence between substring occurrences within the same string. (We shall henceforth refer to this approach as ID.) For example, $P(\sigma_1 \wedge \sigma_2)$ would be estimated as $P(\sigma_1) * P(\sigma_2)$, $P(\sigma_1 \vee \sigma_2)$ as $P(\sigma_1) + P(\sigma_2) - P(\sigma_1) * P(\sigma_2)$, and so forth. Unfortunately, the independence assumption rarely holds in real data sets, and its use in selectivity estimation contexts has repeatedly been shown to give pessimistic results [IC93,PKF00].

3.2.1. Sketch of our approach

We adopt the same set-oriented approach for conjunction queries. For exposition, let us first ignore any space constraints and assume that an unpruned suffix tree has been built from the collection of strings. We denote the string labelling the jth node as w_j. This is the string that spells the path from the root of the suffix tree to its jth node. We augment each node j with a *base set* S_j of the SIDs for each string containing w_j as a substring. The set of strings satisfying any Boolean query and its cardinality can then be computed via set operations on the base sets corresponding to the substring predicates of the query.

Consider the following example. Fig. 9 presents part of the suffix tree constructed from a toy data set, in which each substring node is augmented with its corresponding set of SIDs. Suppose the query $q = \texttt{ab} \wedge \texttt{12}$. Then the nodes j and k in the tree are located such that $w_j = \texttt{ab}$ and $w_k = \texttt{12}$ and $|S_j \cap S_k| = |\{1,2,4\} \cap \{1,2,3\}| = |\{1,2\}| = 2$ gives the result size.

Let N be the number of strings in the collection S; the input size is $O(\sum_{s \in S} |s|)$ which is also the number of nodes in the FST. If we could store the base sets associated with each substring, our problem would be solved—any Boolean query over substring predicates is a set operation on the

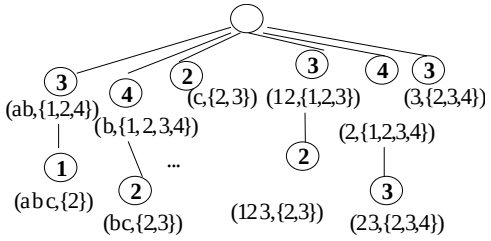


Fig. 9. Set-oriented approach illustrated with a toy example with 4 strings {ab12, abc123, bc123, ab23}.

base sets (and their complements). However, storing all the base sets requires space $\Theta((\sum_{s\in S}|s|)N)$, which is much larger than the linear space we are allowed in the FST variant. Furthermore, answering any query requires time $\Omega(N)$, which is undesirable.

We use the same set hash technique in our solution of the multidimensional substring selectivity problem. That is, we use a set hash signature to store the sets associated with each node, as in the case of multidimensional substring selectivity estimation. This approach based on set hashing uses only $O(\sum_{s\in S}|s|)$ space, and estimates the selectivity of any Boolean substring query by manipulating set hash signatures. Set operations using set hashing can be carried out as described in Section 3.1.3. In the case of boolean selectivity estimation, we also need to estimate the union and intersection size when complements are present. We describe how to do such estimations next.

Estimating union/intersection with complements. We first consider estimating the size of $|A_1\cap\cdots\cap A_k\cap\overline{A_{k+1}}\cap\cdots\overline{A_\ell}|$. We estimate it using set difference as $|A_1\cap\cdots\cap A_k|-|A_1\cap\cdots\cap A_k\cap A_{k+1}\cap\cdots A_\ell|$, where each of the two terms can be estimated as in Section 3.1.3. We estimate $|A_1\cup\cdots\cup A_k\cup\overline{A_{k+1}}\cup\cdots\overline{A_\ell}|$ as its complement (which will suffice) $|\overline{A_1}\cap\cdots\cap\overline{A_k}\cap A_{k+1}\cap\cdots A_\ell|$, which we have already described how to estimate.

3.2.2. Estimating the selectivity of Boolean queries

In this section, we show how to compute the selectivity of any general Boolean query q. Let T be the full suffix tree constructed on the collection of strings S. We wish to compute the selectivity of q. We consider the following two cases:

1. *q does not contain negations*: We convert q to the CNF expression $q'=(\sigma_{11}\vee\cdots\vee\sigma_{1k_1})\wedge\cdots\wedge(\sigma_{n1}\vee\cdots\vee\sigma_{nk_n})$. Each σ_{ij} in q, $1\leqslant i\leqslant n, 1\leqslant j\leqslant k_i$ will be located in T. Let $S_{\sigma_{ij}}$ be the signature at the node that σ_{ij} is located. From these signatures, we can derive the signature $S_{(\sigma_{i1}\vee\cdots\vee\sigma_{ik_i})}$ for each disjunctive clause $(\sigma_{i1}\vee\cdots\vee\sigma_{ik_i})$ as in Section 3.1.3, and then estimate the selectivity of q from their intersection size, which we can determine as described in Section 3.1.3.
2. *q contains negations*: We convert q to the DNF expression $q'=(\sigma_{11}\wedge\cdots\wedge\sigma_{1l_i})\vee\cdots\vee(\sigma_{m1}\wedge\cdots\wedge\sigma_{ml_m})$. In order to eliminate the negation operators, we use the set inclusion–exclusion formula to convert q' to an algebraic expression without negations that yields the same cardinality. Let $C_i=(C_1\vee\cdots\vee C_m)$ be a disjunction clause in q'. Then $|q'|=|C_1|+\cdots+|C_m|-(|C_1\cap C_2|+|C_1\cap C_3|\cdots)\cdots+(-1)^{m-1}|C_1\cap\cdots\cap C_m|$. After the application of this formula, q' contains only conjunctions. Let D be a conjunction expression in q'. We can rewrite D by keeping all negations in the end. Thus, $D=y_1\wedge y_2\wedge\cdots\wedge y_r\wedge\neg z_1\wedge\cdots\wedge\neg z_t$, where $y_i, z_j, 1\leqslant i\leqslant r,\ 1\leqslant j\leqslant t$ are substrings. Then the selectivity of D can be estimated as: $|D|=|y_1\wedge y_2\wedge\cdots\wedge y_r|-|y_1\wedge y_2\wedge\cdots\wedge y_r\wedge z_1\wedge\cdots\wedge z_t|$. Both terms can be estimated as described in Section 3.1.3.

3.2.3. Entire algorithm for the FST case

We can now summarize the entire algorithm for Boolean query estimation for the FST case.

1. *Preprocessing.* We construct the full suffix tree T of the set S of strings. For each node v in T, we store the signature S_{σ_v} for the set of all SIDs that contain σ_v as a substring. We also store the cardinality of this set in node v.

2. *Query processing*: Any Boolean query q gets compiled into an algebraic equation of result sizes of various set operations on the set of SIDs at nodes of T, as described in Section 3.2.2. These result sizes can be estimated using the signatures of these sets via set hashing, as described in Section 3.1.3.

Let us consider the complexity of the algorithm. Construction of the suffix tree takes time $O(\sum_{s\in S}|s|)$, a classical result [McC76]; its size, that is, the number of nodes in it, is also $O(\sum_{s\in S}|s|)$. We can calculate $|S_{\sigma_v}|$ for each v in $O(\sum_{s\in S}|s|)$ time altogether in a bottom-up traversal of T. It takes the reader a little thought to realize that the signature vectors too can be computed by a bottom-up traversal of T, combining the signature vectors of the children at each node. This takes $O(\ell(\sum_{s\in S}|s|))$ time where the signature for any set has ℓ components. Thus the overall running time of Step 1 is $O(\ell(\sum_{s\in S}|s|))$ which is also the space used in the data structure. In Step 2, say the query q has L predicates; the size of the CNF or DNF formula is at most $L2^L$. The time to estimate a CNF formula is thus $2^{O(L)}$. For a DNF formula, it is expensive to employ the inclusion–exclusion formula in its entirety; however, it would suffice to consider only the intersections of a constant number of a clauses in Section 3.1.3, in which case, the time taken is $2^{O(L)}$; this is the overall complexity of Step 2.

Theorem 3. *The algorithm in Section* 3.2.2 *for estimating Boolean query selectivity with set S in the FST case takes time and space* $O(\ell(\sum_{s_\in S}|s|))$ *for preprocessing*; *here* ℓ *is the size of a signature vector. Each query q on L predicates can be estimated in time* $2^{O(L)}$.

In practice, L is very small. Furthermore, any user-specified query can be expanded quite simply to remove nesting, if any, and the resultant queries are likely to be linear in the original query size. Any such nested-free Boolean query can be estimated using Section 3.1.3 very efficiently. Also,in practice, it suffices to use $\ell \leqslant 200$ bytes. For a more detailed experimental study of this algorithm, see Section 4.2.

3.3. Proposed solution for Boolean queries using a pruned suffix tree (PST)

This section presents our solution to the second variant of our problem, the pruned suffix tree (PST) case. What differentiates this variant with the FST case is that some substrings from the query may not be located in the suffix tree. Thus, we must rely on parsing the query into subqueries on substring predicates that can be located in the tree to reduce the problem to the FST case; the selectivity of these subqueries can then be algebraically combined via the previously proposed probabilistic formulae [JNS99,WVI97] to estimate the overall selectivity of the query.

The previous study in substring selectivity estimation [JKNS99,JNS99,KVI96,WVI97] presents two parsing techniques, greedy parsing and maximal overlap parsing (MO), to parse substring predicates to a set of substrings present in a PST. We generalize MO in this paper because it has been shown to be superior than greedy parsing [JKNS99,JNS99]. The detail of MO parsing can be found in [JKNS99,JNS99]. The algorithm is as follows.

The algorithm: The input is a Boolean substring query q and a pruned suffix tree T. We write q as a function on substring predicates, i.e., $q(\sigma_1, \ldots, \sigma_L)$, which contains substring predicates σ_i and logical connectives $\wedge, \vee, \neg$. The PST T is built on all strings in S applying known pruning

1. *Parse predicates into substrings located in T.* Each predicate σ_i is parsed independently into substrings $\sigma_i(1), \ldots, \sigma_i(j_i)$, that match nodes in the PST.
2. *Rewrite q to remove negations.* If q contains negations, apply the inclusion-exclusion formula from Section 3.2.2 case 2 to rewrite q as a set of terms $t_1, \ldots, t_n$, where each term is free of negations. Otherwise, return q as the only term t_1.
3. *Estimate the selectivity for each term.*
   For each term t_h
   - *Generate a set of subqueries.* We generate $t_h(\sigma_1(j_1), \ldots \sigma_L(j_L))$, that is, for each substring predicate σ_i in term t_h, we substitute the predicate with the corresponding parsed substring $\sigma_i(j_i)$ and form a subquery containing only predicates located in the PST.
   - *Estimate the selectivity for each subquery.* Since each subquery only contains substring predicates located in the PST, we can estimate the selectivity as described in Section 3.2.2 case 1.
   - *Algebraically combine the selectivities via probabilistic formula.* We apply a probabilistic formula to combine selectivities of the subqueries formed from t_h.
4. *Perform arithmetic on selectivities of terms.* If more that one terms are present, add and subtract terms (as described in Section 3.2.2 case 2) to derive the overall selectivity of q.

Fig. 10. Estimating selectivities of Boolean queries using a PST.

strategies in [JKNS99,WVI97]. With each node in the PST, we store the exact count of the number of string IDs of that substring (labelling the path from the root to that node). We also store the signature of the set of all string IDs which contain that substring. The output is an estimate of the selectivity of q. The algorithm is shown in Fig. 10.

Example. Assume $q = (abc \wedge 12) \vee \neg 23$. In step 1, MO parsing parses abc into ab and bc, and their overlap is b. Both 12 and 23 are located in T. In Step 2, since q contains negations, two terms $t_1 = 23$ and $t_2 = abc \wedge 12 \wedge 23$ are generated, where $|q| = 1 - |23| + |abc \wedge 12 \wedge 23|$. Now Step 3 estimates the selectivity for terms t_1 and t_2. The substring in t_1 happens to be stored in the PST, so we lookup the associated count in the node. To estimate the selectivity for t_2, we first generate subqueries by replacing each substring predicate in t_2 with a parsed substring or overlap. There are three subqueries: $t_{21} = t_2(ab, 12, 23) = ab \wedge 12 \wedge 23$, $t_{22} = t_2(bc, 12, 23) = bc \wedge 12 \wedge 23$, and $t_{23} = t_2(b, 12, 23) = b \wedge 12 \wedge 23$. Since each subquery only contains substring predicates located in T, we can use the technique described in Section 3.2.2 case 1, to derive their selectivities. Suppose the estimates are 0.24, 0.6, and 0.55. Then we combine them by conditioning on the subquery containing overlap of parsed substrings (t_{23}). That is,

$$P(t_2) = P(t_{21}) \times P(t_{22}|t_{21}) \simeq P(t_{21}) \times P(t_{22}|t_{23}) = P(t_{21})\frac{P(t_{22})}{P(t_{23})} = 0.24 * 0.6/0.55 = 0.26.$$

Finally, Step 4 combines the selectivities for terms to the selectivity of q. $|q| = 1 - |23| + |abc \wedge 12 \wedge 23| = 1 - 0.75 + 0.26 = 0.51$.

3.4. Multidimensional general Boolean queries

The extension for Boolean queries on multiple attributes is straightforward. We construct k suffix trees (either FST of PST), that is, one on each dimension; details on how to prune in the

PST case are given in [JKNS99,WVI97]. The suffix tree nodes contain the same auxiliary information as discussed in the previous sections. Query parsing is identical to the multi-dimensional conjunctive case. The same probabilistic estimation strategy used in Section 3.2 (for FST) and Section 3.3 (for PST) are employed.

4. Evaluation

In this section we present a detailed experimental evaluation of our proposals. We present the evaluation of our proposal for the multidimensional selectivity estimation with conjunctions in Section 4.1, and continue in Section 4.2 with the evaluation of our proposal for Boolean queries.

4.1. Experimental evaluation for multidimensional conjunctive queries

In this section we present the results of a thorough experimental evaluation of the proposed set hashing approach on conjunction queries. We start with an extensive investigation of the accuracy compared to the previous methods, with the focus on overall average estimation error over a variety of data sets and workload characteristics; we also look closely at the error distribution. Of the previous methods, every combination of data structures and parsing strategies were compared against: WVI with ID parsing (`WVI–ID`), WVI with MO parsing (`WVI–MO`), KD with ID parsing (GNO, also called KD–ID in this paper), and KD with MO parsing (`KD–MO`). We also tried two variations on the set hashing approach: SH with ID parsing (`SH–ID`) and SH with MO parsing (`SH–MO`). Section 4.1.1 describes the experimental setup. Section 4.1.2 compares the accuracy of the methods. Section 4.1.3 presents results for high dimensional data sets. Section 4.1.4 compares their runtime efficiency. Section 4.1.5 examines how to set parameters for the set hashing approach.

4.1.1. Experimental setup

We describe below the data sets used in our experiments, how queries were formed in the test workload, and how we evaluated the results. We also mention implementation details.

Data sets: The reported results are obtained from two AT&T data sets: 200 K events (6 MB) logged by an AT&T service with attributes `event_name` and `bundle`, which we refer to as EVENT; and 130K strings (5 MB) containing a brief English description of a service provided to customers, referred to as SERVICE, in which the field `billing` is replicated to create two attributes. We observed that there exists a higher correlation between the attributes of SERVICE compared to that of EVENT.

Queries: Following [JKNS99,JNS99,KVI96], we tested both *positive* (substring queries exist in the data set) and *negative* (substring queries are not present in the data set) 2-D substring queries. Positive queries were randomly sampled from within strings of the data set. The tuples from which these substrings come were chosen uniformly; then, for each attribute, the start position was chosen uniformly, and the length uniformly varied between 2 and 7 characters. Negative queries are generated similarly, except that each attribute comes from a different tuple such that the query does not find a full match. A workload Q consists of 1000 queries. We present experiments using four different workloads, consisting of queries of various selectivities. Workload U consists of

queries uniformly selected over a range of selectivities, L consists of queries with selectivity lower than 0.1%, M consists of queries with selectivity between 0.1% and 1% and H has queries with selectivity above 1%.

Error measures: Let S_q be the true selectivity of a query q in workload Q, S'_q the estimate, and N the number of strings in S. We use the average-absolute-relative error, a standard measure in selectivity estimation, to quantify the accuracy of positive queries:

$$E_{abs} = \frac{1}{N} \sum_{q \in Q} \frac{|S_q - S'_q|}{S_q}.$$

Following [JKNS99], we use the root-mean-squared error to quantify the accuracy of negative queries:

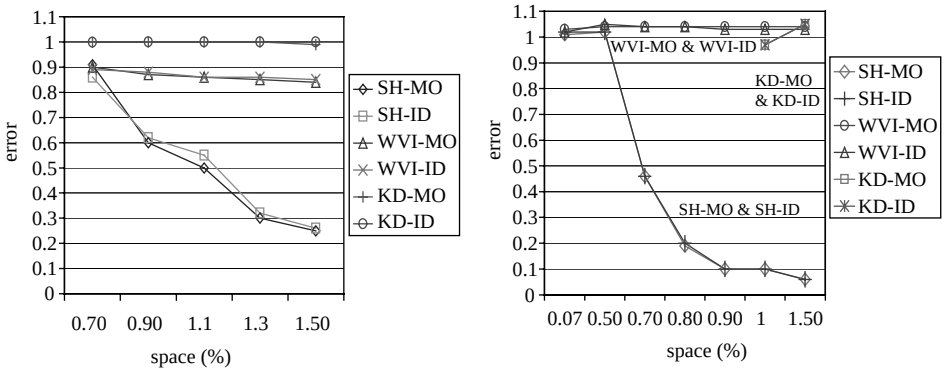
$$E_{std} = \sqrt{\frac{1}{N} \sum_{q \in Q} (S_q - S'_q)^2}.$$

Implementation: We implemented the SH-ID and SH-MO methods in C++ on a 350 MHz Pentium 2 PC with 128 MB memory. During the building phase, a pruning procedure is called whenever necessary to prevent the structure from exceeding the space constraint; therefore, the final suffix tree is an approximation of what would have been obtained by building the entire suffix tree and then pruning it. We used the same code from [JKNS99] for the KD–MO and KD–ID methods. To be generously fair to their implementation, we allow the entire KD-suffix tree to be built before pruning. We also implemented the version of WVI–ID which performed the best in [WVI97]; WVI–MO was implemented similarly, but with MO parsing.

Parameter setting: The length of the hash signatures as well as the hash space size are two very important parameters in determining the accuracy of the set hashing technique. We explain in detail how to choose these parameters in Section 4.1.5. We use the values determined by the observations of Section 4.1.5 for our experiments in this section. The hash space size is kept constant at 19 bits in all experiments.

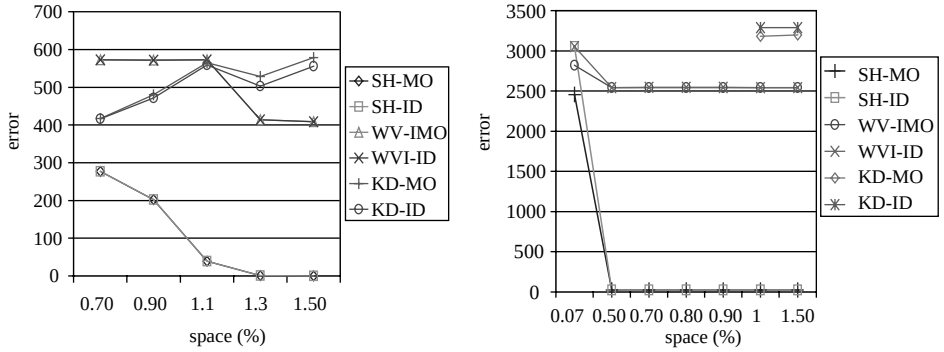
4.1.2. Accuracy

We measured the average error of positive and negative queries. For the positive queries, we investigated the effects of different workload selectivities. We also looked at the distribution of errors over ranges. Our finding is that methods based on set hashing are substantially more accurate (up to an order of magnitude) over a variety of workloads for positive queries; the accuracy is even better for negative queries. Moreover, the distribution of errors is considerably more favorable for the set hashing methods in that very few queries give bad estimates, which is not the case for the other methods. For set hashing methods, the number of cross-counts that can be generated increases quadratically, bringing about a rapid drop in the error; this is in contrast to the linear growth in the number of cross-counts explicitly stored by previous methods. It is interesting to note that, for all methods, there is a relatively small difference between MO and ID parsing in small space. Essentially, MO parsing is very similar to ID parsing, since lots of strings are pruned away and very few overlaps are identified in the substrings during query parsing. MO parsing becomes effective (for all methods) when large amounts of space are available, but is not practical at small space. Moreover, the difference of employing the set hashing approach has



(a) EVENT (b) SERVICE

Fig. 11. Accuracy of positive queries using workload U for the two data sets, E_{abs} versus space: (a) Event; (b) Service.



(a) EVENT (b) SERVICE

Fig. 12. Accuracy of negative queries for the two data sets, E_{std} versus space: (a) Event; (b) Service.

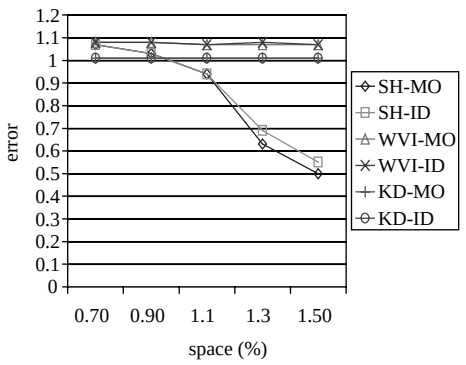
much more significant bearing on the accuracy than the parsing strategy or probabilistic formula employed. Below we give more details.

Positive queries. The average error for set-hashing methods was smaller by a factor of 4 than the four competing methods, for the EVENT data set; for SERVICE, the improvement was a factor of 10. Fig. 11 plots E_{abs} versus space for these data sets.[7]

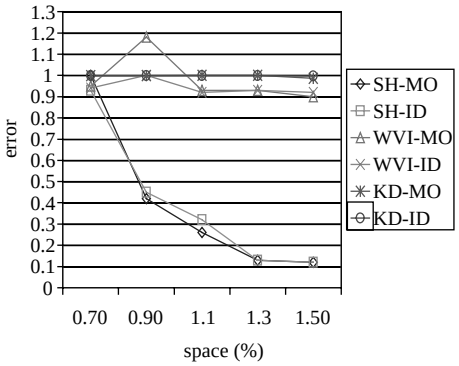
Negative queries. Fig. 12 plots root-mean-squared error versus space for negative queries, for both data sets. Set hashing methods are astoundingly more accurate than the other methods on both data sets up to 3 orders of magnitude.

Workload. In our investigation of the effect of workload, all methods achieved substantially better accuracy for higher query selectivities; the set hashing methods are better by a factor of up to 10. We tested three workloads for which the selectivity was low, medium, and high, respectively. For brevity, we only report results from the EVENT data set. Fig. 13(a)–(c) presents

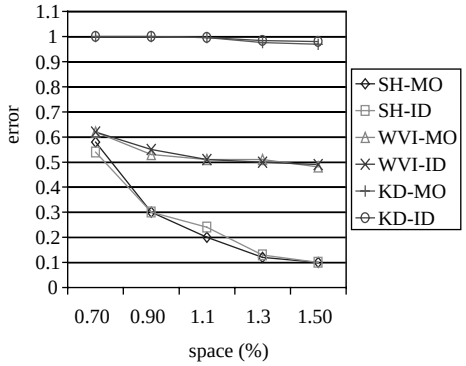
[7] The code available to us for the implementation of the KD method always keeps top level nodes, whose size is already 1% of the SERVICE data set; thus we, cannot show accuracy results for space less than 1% space for this data set.



(a) Workload L



(b) Workload M



(c) Workload H

methods	*low*	*medium*	*high*
SH-MO	0.1	0.12	0.5
SH-ID	0.1	0.12	0.55
WVI-MO	0.48	0.9	1.07
WVI-ID	0.49	0.92	1.07
KD-MO	0.97	1	1.01
KD-ID	0.98	1	1.01

(d) table summary

Fig. 13. Accuracy (measured with average-absolute-relative error) for the EVENT data set, using workloads consisting of queries with various selectivities: (a) Workload L; (b) Workload M; (c) Workload H; (d) table summary.

E_{abs} versus space for all methods, using workloads L, M, and H; (d) summarizes these results at 1.5% space overhead. We verified that, indeed, queries with low selectivity tend to be instantiated deep down in the unpruned suffix tree, and thus are pruned away; this in turn requires cruder estimations based on many subqueries. Note that WVI improves more than KD does for workload H. We believe that it is due to the space overhead of the k-D suffix tree, as the KD method devotes more space to instantiate the tree structure compared to WVI methods, thus storing fewer exact counts.

Distribution. To gain an understanding of the distribution of the error, we performed the following experiment. We created an equiwidth histogram on the x-axis consisting of 10 buckets ranging between 0 and 100% error, and two extra buckets representing error between 100% and 200%, and above 200%. For the results reported in Fig. 13(d), we examined the number of queries at the various error ranges, for each workload. The results are shown in Fig. 14. Since MO and ID parsing techniques gave similar error, we limit our discussion to the MO parsing technique.

It is interesting to note that, for all methods, there are few queries in the middle error buckets (between 30% and 80%). This shows that the estimates are hit-or-miss. Queries with small

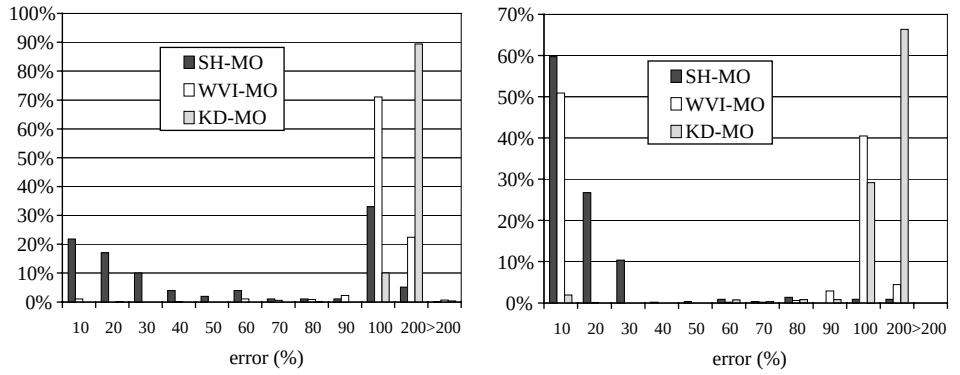


Fig. 14. Fraction of queries allocated to various error classes for the EVENT data set, using 1.5% space for various workloads: (a) Workload L; (b) Workload H.

selectivities, such as those existing in workload L are located very deeply in the unpruned suffix tree. Thus, with very high probability they are not present in the PST. Selectivity for those queries rely more heavily on estimates. Set hashing, being able to derive accurate cross count estimates, does a good job in keeping a rather large fraction of the queries in L (almost 50%) below the 40% error range. Queries with large selectivity, since they are present high in the unpruned suffix tree, require few estimated counts. This explains the improvement in WVI accuracy in Fig. 14(b). Notice that almost 99% of queries in H are below the 30% error range, using set hashing.

4.1.3. Accuracy in higher dimensions

We have shown that set hashing can be generalized to perform selectivity estimation for dimensions higher than two. Our approach enables a larger number of cross-counts to be generated in linear space, and thus one would expect the superiority of set hashing methods to become more pronounced with increasing dimension. However, there is a trade-off here: in higher dimensions, the number of signature intersections needed to generate a cross-count increases, which in turn leads to increased error in approximating each cross-count.

To assess the accuracy of set hashing with increasing dimension, we conducted the following experiment. We used two-, three- and four-dimensional data sets, extracted from a large AT&T warehouse of billing data. Each data set contains 200,000 tuples and each attribute takes up roughly 3 MB of space. We used the workload U and recorded the average-absolute-relative error as a function of space. The results are presented in Fig. 15. We only compare with a high dimensional generalization of WVI–MO, since, of the existing methods, it was the most accurate in our two-dimensional experiments.

The results show that, as dimension increases and space remains limited, the set hashing approach degrades in accuracy significantly more gracefully than existing methods. Fig. 15(a)–(c) shows the trend of how WVI–MO and SH–MO perform relative to each other as the dimension increases from 2 to 4. In all the plots, the SH-MO accuracy improves steadily with increasing space, with the error dropping to 10% in 2-D and to 25% in 4-D, at 1.5% space. In contrast, the WVI–MO accuracy degrades rapidly with increasing dimension. At 1.5% space, the 2-D error

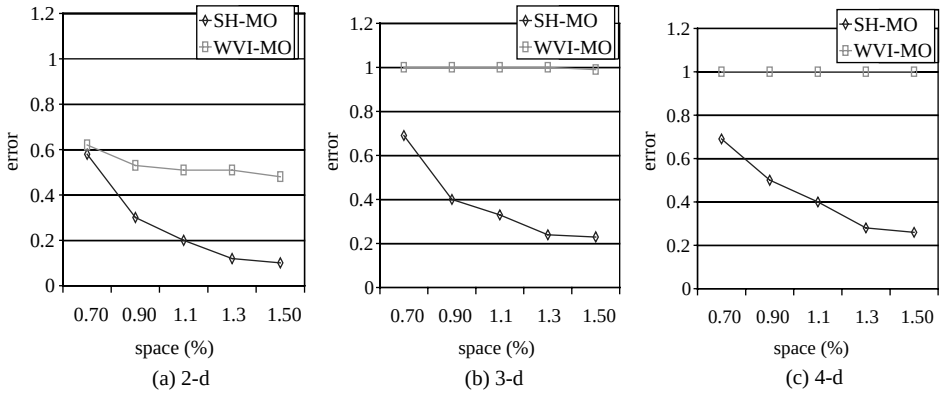


Fig. 15. E_{abs} for WVI–MO and SH–MO as the dimension increases from 2-D to 4-D: (a) 2-D; (b) 3-D; (c) 4-D.

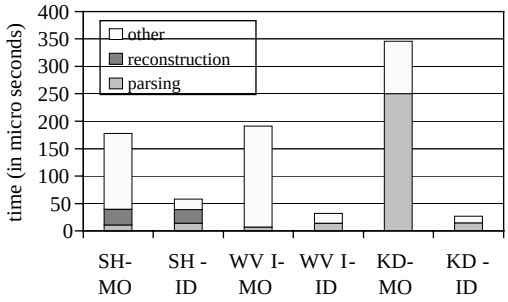


Fig. 16. Average time to perform the estimation for set hashing, WVI and KD-tree methods.

drops to less than 50%; in both 3-D and 4-D there is almost no drop in the error. The reason is that most queries are parsed into many small subqueries with little overlap. Therefore, WVI–MO effectively reduces to GNO in high dimensions.

4.1.4. Runtime performance

We compared the average runtime performance of all the methods. Although the time for these methods is well below a millisecond (and potentially negligible), we present a breakdown of the time required to perform the estimation in Fig. 16. The label *Parsing* shows the time the methods (for each parsing strategy) require to parse the query string. The set hashing methods require extra time to approximate cross-counts (labeled *reconstruction* in Fig. 16). The label *other* represents the time in deriving selectivity estimate using formula. Set hashing, WVI and `KD–ID` methods take approximately 5 μs for parsing; in contrast, `KD–MO` takes 250 μs because it requires additional searching in the two-dimensional structure in order to locate the two-dimensional maximum overlaps. It turns out that the overhead to approximate cross-counts is almost negligible, as the overall estimation time of set hashing methods is better than existing methods that use MO parsing, requiring a total of less than 200 μs. All the methods we studied required under 350 μs overall runtime.

The construction time of set hashing methods is within 5 min for all experiments we run. In contrast, KD–MO and KD–ID methods take much longer time (about 3 h) because these two methods need to collect 2-D information (2-D strings and 2-D cross counts) while set hashing methods only keep 1-D information.

4.1.5. Tuning set hashing parameters

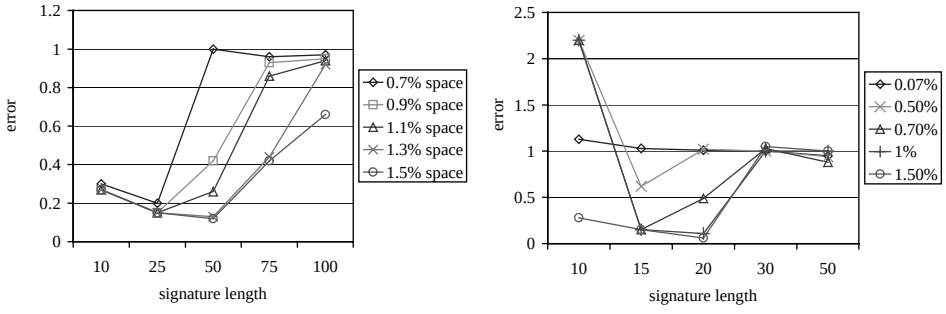
There are two important parameters that can affect multidimensional selectivity estimates based on set hashing: the signature length and the hash space size. In this section, we study the effects of these parameters and conclude with heuristics on setting appropriate values for them.

Signature length. As stated in Theorem 1, the accuracy of set hashing intersection estimates improves with increasing signature length; as a result, the overall estimation accuracy based on the set hashing approach will improve. However, at fixed space, increasing the length of the signatures will also limit the number of suffix tree nodes that can be kept, and thus yield higher error.

Fig. 17(a) presents E_{abs} for SH–MO on the EVENT data set with signature length ranging between 10 and 100; each curve corresponds to a fixed amount of space. The knee of the curves appears at a length of roughly 50. Fig. 17(b) presents the E_{abs} for the SERVICE data set, with signature length varying between 10 and 50. Here the knee of the curves appears at a length of roughly 20. In conclusion, the signature length should be between 20 and 100, with lower correlated data sets requiring longer signatures.

Size of hash space. Clearly a small hash space consumes less space but, to be effective, the size of the hash space should be large enough so as to avoid collisions. In order to identify an optimal hash space size, we performed the following experiment. We fix the total number of suffix tree nodes and compare the accuracy of SH–MO using three different hash space sizes: 2 bytes (16 bits), $\lceil \log N \rceil + 1$ (where N is the number of tuples in the table) bits, and 4 bytes (32 bits).

Fig. 18 shows E_{abs} vs. space for the EVENT data set. The three curves represent the three different hash space sizes we tried. The space on x-axis is the space used by using $\lceil \log N \rceil + 1$ bits. The accuracy of $\lceil \log N \rceil + 1 = 19$ bits and 32 bits is very close. However, the accuracy of 16 bits is much worse. This is intuitive because there are 200,000 tuples in this data set and 16 bits are not enough to represent that many tuples, resulting in many collisions. Once the hash space is sufficiently larger than the domain, the chances of collision become so small that any value above



(a) EVENT (b) SERVICE

Fig. 17. Varying the set hash signature length for fixed amount of space: (a) Event; (b) Service.

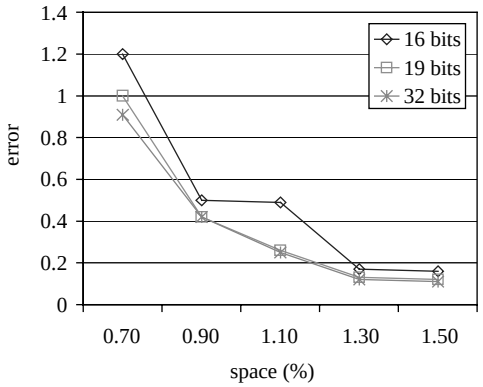


Fig. 18. Effects of varying the hash space size on the accuracy of the set hashing method.

$\lceil \log N \rceil + 1 = 19$ bits gives very good results. Similar trends were observed for the other data sets used in this study. In conclusion, it suffices to use $\lceil \log N \rceil + 1$ bits, where N is the total number of tuples in the table.

4.2. Experimental results for Boolean queries

In order to assess the benefits of our proposed estimation framework for Boolean queries, we performed an experimental evaluation of the proposed method based on set hashing (SH) compared to an approach that assumes independence between the selectivities of the substring predicates in the Boolean query (ID). For both estimation methods we keep the count of substrings associated with each suffix tree node (in both pruned and unpruned suffix tree cases we consider). More specifically, for ID, in the case of the full suffix tree, the selectivity of each clause is estimated via inclusion–exclusion, with the selectivity of a negated predicate $Pr\{\neg P\} = 1 - Pr\{P\}$, and these selectivities are multiplied together. In the case of pruned suffix tree, for ID, we follow the algorithm of Fig. 10 to parse the query into subqueries and make the independence assumption for each subquery. For both methods, we build a count-suffix tree on a set of strings and we report experimental results for the following two cases: (a) the suffix tree is fully materialized; and (b) the suffix tree is pruned to satisfy a specified space constraint. For SH, the nodes are also augmented with signature vectors.

4.2.1. Experimental setup

We describe below the data sets used in our experiments, how queries were formed in the test workload, and how we evaluated the results. We also mention implementation details.

Data sets: The reported results are obtained from the SERVICE and EVENT data sets described in Section 4.1.1. Here the attribute in SERVICE was not replicate, that is, SERVICE contains one attribute while EVENT contains two.

Queries: We tested the accuracy of both positive queries (i.e., matching at least one string in the database) and negative queries (i.e., no matches). We varied the number of substring predicates as well as the number of clauses in the queries in order to evaluate the impact of these parameters on selectivity estimation. Our Boolean queries are derived from the following

templates: $T_1 = (A \vee B) \wedge (C \vee D)$, $T_2 = (A \vee B) \wedge (C \vee D) \wedge (E \vee F) \wedge (H \vee I)$ and $T_3 = (A \vee B \vee C \vee D) \wedge (E \vee F \vee H \vee I)$ where A, B, C, D, E, F, H, I are substring predicates uniformly extracted from attributes in the databases (they can belong to different attributes), with length uniformly chosen between 2 and 7 characters. These templates were chosen because they cover a variety of common queries. Each predicate is preceded with a negation with certain probability in order to investigate the accuracy of our technique in the presence of negations; the negation probability was varied in our experiments. Our workload Q consists of 1000 queries according to each query template.

Error measures: We use the same metrics as defined in Section 4.1.1.

Implementation: For the pruned suffix tree (PST) case, we used MO parsing in both methods. The set hash signature sizes were set at 50 with a hash space of 2^{17}.

4.2.2. Estimation accuracy using a full suffix tree

We allowed the suffix trees employed by the competing methods to store all substrings occurring in the database, that is, they were fully materialized and unpruned.[8] Fig. 19(a) presents the results for the SERVICE data set. The figure shows the accuracy of the competing methods as a function of the probability of negations appearing in predicates; the curves represent workloads of positive queries from different query template classes. The runtime was well below 1 ms for both methods, on all queries. In these experiments, SH was significantly more accurate than ID, almost 10 times better when the number of predicates per clause is two and 5 times better when there are four predicates per clause.

SH experiences a small increase in estimation error as the number of predicates per clause and the probability of negations increases, since, due to DNF conversion, more set hash signatures are involved in the estimation. As the probability of negation increases, ID constantly overestimates the true selectivity of each clause, and tends to underestimate the true selectivity due to multiplication of individual estimations. As a result the overall estimation error could have varying trends depending on the relative error terms introduced by over and under estimation. In Fig. 19(a) the curve appears flat since the contribution of over and under estimation seem to cancel out. As the number of predicates per clause increases, the gap between the estimation accuracy of SH and ID decreases. Having more predicates per clause forces the disjunctions to become less correlated; thus the accuracy of the independent estimation method increases. The effect of increasing the number of clauses in the query (T2) is not shown in Fig. 19(a) because ID incurs error which is out of the scale of the figure (above 2.5). As the number of clauses increases, the overall selectivity is expected to decrease; since ID makes the independence assumption (which not is true in this case) its accuracy gets penalized by large factors. In contrast, SH experiences a small decrease in accuracy in this case, since it is able to preserve correlations better. Fig. 19(b) presents the results of the same experiment for negative queries on the SERVICE data set. SH in all cases offers very accurate estimation, in contrast with ID. The estimation benefits of SH are evident, with SH outperforming ID by many orders of magnitude.

Fig. 21 shows similar results for the EVENT data set on a FST; the y-axis is in logarithmic scale because ID method returns large errors. Clearly, the errors for both methods on the

[8] Note that the suffix tree for SH will consume a constant factor more space than that for ID since the tree nodes for SH are augmented with signature vectors as well as counts.

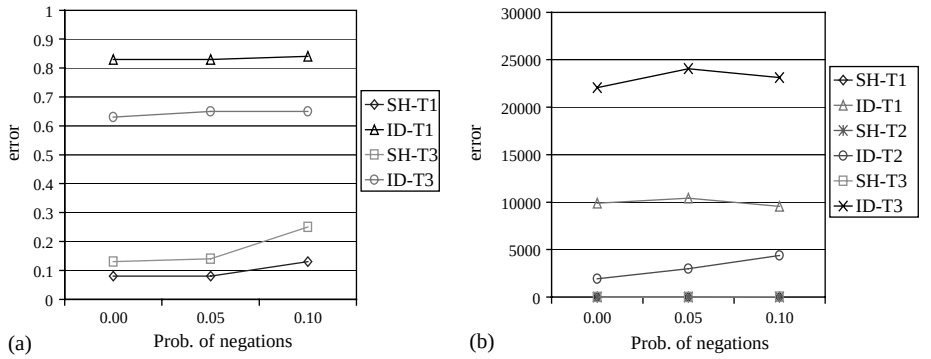


Fig. 19. Accuracy for positive and negative queries on the SERVICE data set(FST): (a) Queries generated by T1 and T3; (b) Accuracy for negative queries.

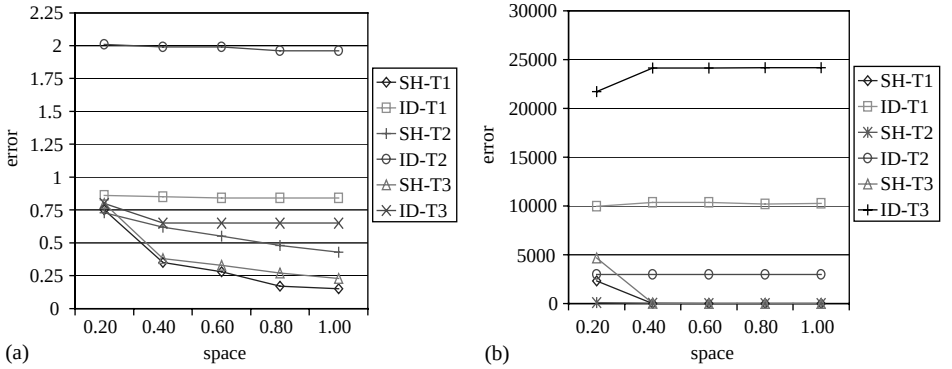


Fig. 20. Accuracy for positive and negative queries on the SERVICE data set (PST): (a) Positive queries; (b) Negative queries.

two-dimensional EVENT data are higher than that on SERVICE, which has only one attribute. This is not surprising, given that the correlations between substrings within the same attribute as well as across attributes are needed. In fact, the error gap between each respective SH and ID is substantially greater over this data set compared with SERVICE. Thus, it appears that SH scales more gracefully (in accuracy) with increasing dimension compared to ID.

4.2.3. *Estimation accuracy using a pruned suffix tree*

We pruned both suffix trees to the same amount of space and we report on the estimation accuracy of the methods. Fig. 20 presents the results for the EVENT data set. The figure shows the accuracy of the methods, when we vary the space allowed to the suffix tree from 0.2% (5 KB) to 1% (25 KB) of the data set size. Fig. 20(a) presents curves for each method, for positive queries and each query template. In all cases, the probability of negations being present in a predicate is 0.05. At very small amounts of space, both methods yield large errors because there is a storage overhead before enough information is in the PST to be useful. As the space increases, the overall trends in estimation error of both methods are similar with the unpruned case. SH outperforms ID

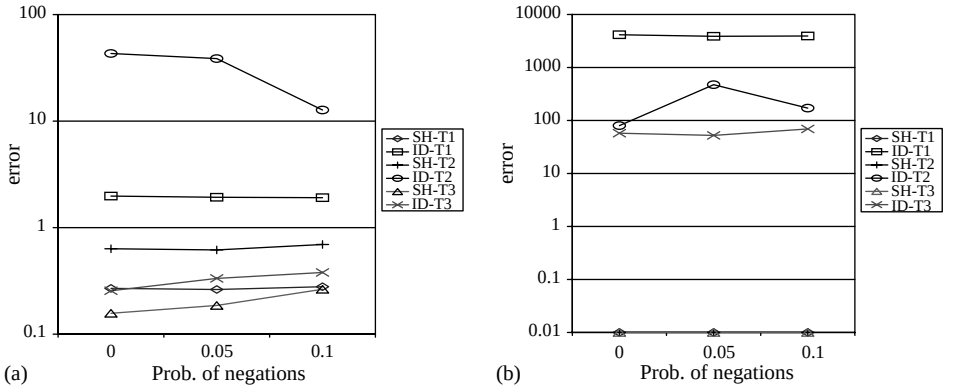


Fig. 21. Accuracy for positive and negative queries on the EVENT data set(FST): (a) Queries generated by T1 and T3; (b) Accuracy for negative queries.

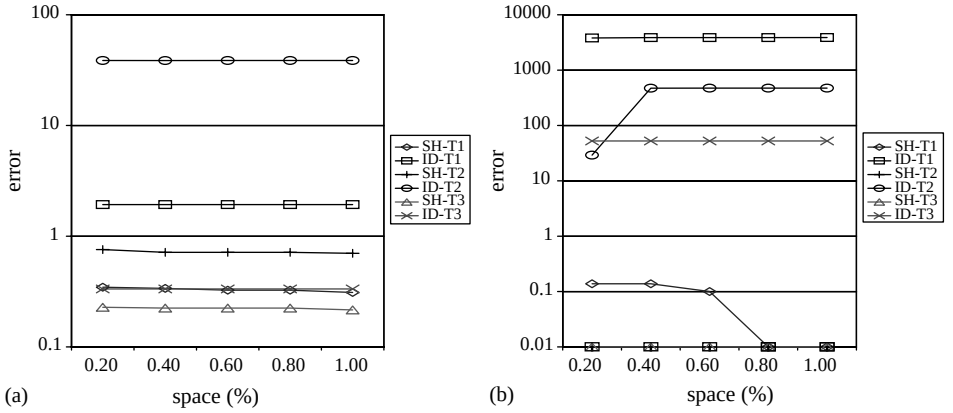


Fig. 22. Accuracy for positive and negative queries on the EVENT data set(PST): (a) Positive queries; (b) Negative queries.

consistently (by up to 6 times) as the number of predicates per clause increases and as the number of clauses in the query increases.

Fig. 20(b) presents the results of the same experiment but for negative queries on the SERVICE data set. As space increases, the overall trends become similar to those of the unpruned case. `SH` offers excellent accuracy outperforming `ID` by several orders of magnitude for all query templates.

Fig. 22(a) and (b) show similar the results for the EVENT data set. The y-axis is in logarithmic scale because `ID` returns large errors.

Surprisingly, our experimental results show similar accuracy for both the FST and PST cases. This seems to imply that MO parsing does not contribute much to the error, and that PST is just as good as FST for obtaining accurate selectivity. However, in other experiments, not included here due to lack of space, this was not the case (Figs. 20–22).

5. Concluding remarks

In emerging applications, database systems increasingly handle textual data and process both single- and multi attribute substring queries with Boolean predicates. We have presented solutions for estimating the selectivity of such queries.

Previous multidimensional substring selectivity estimation methods explicitly store *cross-counts* for the most frequently co-occurring substrings. In this paper, we have presented a novel approach for multidimensional substring selectivity estimation in which we do not explicitly store cross-counts, but rather generate them on-the-fly based on a technique called *set hashing*. Thus, using only linear storage, a large number of cross-counts can be generated as needed, including Boolean combinations of the substring occurrences. Although the cross-counts generated are not exact, but rather approximate, our experiments show that our overall approach is significantly more accurate than known methods in two dimensions; the improvement is even more impressive in higher dimensions. For general Boolean queries, the set hashing based approach significantly outperforms alternatives.

Several important issues are raised by this study. First, we do not consider more general query algebras, such as Boolean queries over substring predicates with regular expressions, though they may be of interest. For example, one may seek documents in which the words (or substrings) σ_1 and σ_2 are separated by white space (i.e., matching the pattern $\sigma_1[\langle space\rangle|\langle tab\rangle|\langle newline\rangle]^*\sigma_2$). Second, it would be worthwhile to extend this framework to allow positional constraints between predicates (e.g., the keywords appear near each other in the document), as is employed in many current systems [ST94].

At the theoretical core, we have used set hashing to estimate the cardinality of various Venn combinations of sets. Although this does not give us provably good estimates for certain combinations (for example, intersections of two or more sets), no hash functions exist with this property given the small space constraint in our problem.[9] However, our experiments show very accurate performance. As new theoretical tools emerge for estimating cardinality of Venn combinations of sets, one can revisit the multidimensional and Boolean substring selectivity estimation problem to see if they provide significantly improved estimates in practice.

Acknowledgments

We thank Divesh Srivastava for thought-provoking discussions, and for supplying code and data sets for our experiments.

References

[BYRN99] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison–Wesley, Reading, MA, 1999.

[9] This has been formalized and proved in the area of Communication Complexity [Lov90].

[Bro98] A. Broder, On the resemblance and containment of documents, IEEE Compression and Complexity of Sequences, Positano, Salerno, Italy, 1997, pp. 21–29.

[BCFM98] A. Broder, M. Charikar, A. Frieze, M. Mitzenmacher, Minwise independent permutations, Proceedings of STOC, Dallas, TX, 1998, pp. 327–336.

[CCT00] C. Clarke, G. Cormack, E.A. Tudhope, Relevance ranking for one to three term queries, Inf. Process. Manage. 36 (2) (2000) 291–311.

[Coh97] E. Cohen, Size-estimation framework with applications to transitive closure and reachability, J. Comput. System Sci. 55 (1997) 441–453.

[CDF+00] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, C. Yang, Finding interesting associations without support prunning, Proceedings of the 16th Annual IEEE Conference on Data Engineering (ICDE 2000), San Diego, CA, February 2000, pp. 489–499.

[CKKM00] Z. Chen, F. Korn, N. Koudas, S. Muthukrishnan, Selectivity Estimation for Boolean Queries, ACM Principles of Database Systems (PODS), May 2000, Dallas, TX, 216–225.

[DD97] C.J. Date, H. Darwen, A Guide to the SQL Standard, Addison–Wesley, New York, 1997.

[HHW97] J. Hellerstein, P. Haas, H. Wang, Online Aggregation, Proceedings of SIGMOD '97, Tucson, AZ, June 1997.

[IC93] Y. Ioannidis, S. Christodoulakis, Optimal histograms for limiting worst-case error propagation in the size of join results, ACM Trans. Database Systems 18 (4) (1993) 709–748.

[JKNS99] H.V. Jagadish, O. Kapitskaia, R. Ng, D. Srivastava, Multidimensional substring selectivity estimation, Proceedings of VLDB, Edinburgh, Scotland, September 1999, pp. 287–398.

[JKS00] H.V. Jagadish, N. Koudas, D. Srivastava, On effective multidimensional indexing for strings, Proceedings of ACM SIGMOD, Dallas, TX, May 2000, pp. 403–414.

[JNS99] H.V. Jagadish, R. Ng, D. Srivastava, Substring selectivity estimation, ACM Principles of Database Systems (PODS), Philadelphia, PA, June 1999, pp. 249–260.

[KT00] M. Kobayashi, K. Takeda, Information retrieval on the web: selected topics, IBM research, Tokyo Research Laboratory, IBM, 2000.

[KVI96] P. Krishnan, J.S. Vitter, B. Iyer, Estimating alphanumeric selectivity in the presence of wildcards, Proceedings of SIGMOD, Montreal Canada, June 1996, pp. 282–293.

[Lov90] L. Lovasz, Communication complexity: a survey, in: B. Korte, L. Lovasz, H. Promel, A. Schrijver (Eds.), Paths, Flows and VLSI Layout, Springer, Verlag, Berlin, 1990.

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

[MMSZ98] Y. Matias, S. Muthukrishnan, S. Cenk Sahinalp, J. Ziv, Augmenting suffix trees, with applications, in: ESA '98, Sixth Annual European Symposium, August 1998.

[MD98] M. Muralikrishna, D.J. DeWitt, Equi-depth histograms for estimating selectivity factors for multidimensional queries, Proceedings of ACM SIGMOD, Chicago, IL, June 1988, pp. 28–36.

[MPS99] S. Muthukrishnan, V. Poosala, T. Suel, Partitioning two dimensional arrays: algorithms, complexity and applications, Proceedings of the International Conference on Database Theory, Jerusalem, Israel, 1999.

[PKF00] B.-U. Pagel, F. Korn, C. Faloutsos, Deflating the dimensionality curse using multiple fractal dimensions, in: Proceedings of ICDE 2000, San Diego, CA, February 2000, pp. 589–598.

[PI97] V. Poosala, Y. Ioannidis, Selectivity estimation without the attribute value independence assumption, Proceedings of VLDB, Athens, Greece, August 1997, pp. 486–495.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, E. Shekita, Improved histograms for selectivity estimation of range predicates, Proceedings of ACM SIGMOD, Montreal, Canada, June 1996, pp. 294–305.

[ST94] A. Salminen, F.W. Tompa, PAT expressions: an algebra for text search, Acta Ling. Hung. 41 (4) (1994) 177–306.

[SM83] G. Salton, M.J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill, New York, 1983.

[SHM98] C. Silverstein, M. Henzinger, H. Marais, Analysis of a very large altavista query log, Technical note #1998-014, Digital SRC, October 1998.

[TBS97] E. Tanin, R. Beigel, B. Shneiderman, Design and evaluation of incremental data structures and algorithms for dynamic query interfaces, in: Proceedings of IEEE InvoViz '97, Phoenix, AZ, October 1997.

[VWSG97] B. Vélez, R. Weiss, M. Sheldon, D. Gifford, Fast and effective query refinement, in: ACM SIGIR'97, Philadelphia, PA, July 1997.

[WVI97] M. Wang, J.S. Vitter, B. Iyer, Selectivity estimation in the presence of alphanumeric correlations, Proceedings of ICDE, Birmingham, UK, 1997, pp. 169–180.