Selectivity Estimation For Boolean Queries

Zhiyuan Chen *
Cornell University

Flip Korn
AT&T Labs-Research

Nick Koudas
AT&T Labs-Research

S. Muthukrishnan
AT&T Labs-Research

ABSTRACT

In a variety of applications ranging from optimizing queries on alphanumeric attributes to providing approximate counts of documents containing several query terms, there is an increasing need to quickly and reliably estimate the number of strings (tuples, documents, etc.) matching a Boolean query. Boolean queries in this context consist of substring predicates composed using Boolean operators. While there has been some work in estimating the selectivity of substring queries, the more general problem of estimating the selectivity of Boolean queries over substring predicates has not been studied.

Our approach is to extract selectivity estimates from relationships between the substring predicates of the Boolean query. However, storing the correlation between all possible predicates in order to provide an exact answer to such predicates is clearly infeasible, as there is a super-exponential number of possible combinations of these predicates. Instead, our novel idea is to capture correlations in a space-efficient but approximate manner. We employ a Monte Carlo technique called set hashing to succinctly represent the set of strings containing a given substring as a signature vector of hash values. Correlations among substring predicates can then be generated on-the-fly by operating on these signatures.

We formalize our approach and propose an algorithm for estimating the selectivity of any Boolean query using the signatures of its substring predicates. We then experimentally demonstrate the superiority of our approach over a straightforward approach based on the independence assumption wherein correlations are not explicitly captured.

1. INTRODUCTION

Boolean queries over substring predicates, that is, logical expressions composed of keywords connected by the Boolean operators `AND`, `OR`, and `NOT`, or recursively composed expressions thereof, are very common. They have been employed in Information Retrieval systems for decades [15]. Two examples of Boolean queries are `peanut AND butter`, which finds documents containing both the word `peanut` and the word `butter`, and `peanut AND NOT butter`, which finds documents containing `peanut` but not containing `butter`. Due to the proliferation of the Internet, Boolean queries have become ubiquitous, for example, in Web search engines, online digital libraries, *etc.* For example, 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 [16].

It is often useful to obtain a fast and accurate estimate of the fraction of documents matching a query. The *selectivity* of Boolean queries can be used by the system for query optimization in Information Retrieval systems to find the best ordering of keywords for filtering. Selectivity estimates may also be useful for the users to formulate more (or less) precise Boolean queries, that is, to *refine* queries. A notorious problem in Information Retrieval is that the number of documents matching a given query is often too large for a user to sift through. Unfortunately, ranking the documents based on relevance is not effective when the search terms are too general [4]. 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 [18; 17]. Selectivity estimation also has the potential to become a commonplace operation as network elements diversify, and thin clients such as palm-held devices redefine Web browsing to inherently rely on query refinement based on estimates.

Selectivity estimation techniques are well developed for numerical attributes, and equality or range queries over them [9]. Selectivity estimation for string attributes is a more recently studied topic. However, all such results concern only the substring selectivity problem, namely, determining the number of documents or tuples that contain a query substring (in one or more dimensions). A one-dimensional substring selectivity query is a Boolean query with merely the single substring predicate, and a multidimensional substring selectivity may be thought of as a Boolean query of a single clause comprising only conjunctions. To the best of our knowledge, selectivity estimation of more general Boolean queries has not been addressed thus far.

*This work was done during the author's visit at AT&T Labs.

In this paper, we initiate the formal study of selectivity estimation problem on Boolean substring predicates. 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. Our novel contribution in this paper is the approach we develop for capturing these correlations in a space-efficient, but approximate manner. We employ a Monte Carlo technique called *min-wise independent permutations* 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 using *set hashing* a technique we introduce, to perform 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 [16]. Our selectivity estimation algorithms are very efficient for such cases. A simple approach to the Boolean selectivity estimation problem would have been to assume independence between substring occurrences within a string, and thereby not attempt to capture any correlation amongst their occurrences. Our experimental results show that our approach significantly outperforms the independence-based approach.

The rest of the paper is organized as follows: In Section 2, we formalize the problem and define the two variants. In Section 3, we present our algorithm for capturing the correlations amongst substring predicates using set hashing, for the full suffix tree variant. In Section 4, we show how to modify that algorithm for the pruned suffix tree variant. In Section 5, we present experimental results. In Section 6, we conclude with some remarks and directions for future work.

2. PROBLEM DEFINITION

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 another string s to test if σ is a substring of s.

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

- Any substring predicate σ 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, *i.e.*, $(\sigma_{11} \vee \cdots \vee \sigma_{1\ell_1}) \wedge \cdots \wedge (\sigma_{k1} \vee \cdots \vee \sigma_{k\ell_k})$. 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, *i.e.*, $(\sigma_{11} \wedge \cdots \wedge \sigma_{1\ell_1}) \vee \cdots \vee (\sigma_{k1} \wedge \cdots \wedge \sigma_{k\ell_k})$.

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. Any practical selectivity estimation method should provide acceptable accuracy while also being significantly more efficient than solving the problem exactly, that is, obtaining the precise number of strings that satisfy q.

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.[1] 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.

It is unusual to consider selectivity estimation on a domain (be it numerical or string) using a linear amount of space. That is because most selectivity estimation problems studied thus far are rather trivial if 100% space is allowed. For example, the substring selectivity problem can be solved exactly if the entire suffix tree were available. (Note that this would require time proportional to the length of the query substring and independent of the size of S.) Hence, the problem in those cases becomes interesting *only* when the space allowed is significantly small, say 20% of the input size. There are, however, a few selectivity estimation problems where the problem is nontrivial *even* if 100% of the space is allowed. An example is the substring selectivity problem on multiple string attributes. Our Boolean query selectivity estimation problem is another example, since all known text indexing methods for finding documents that satisfy a Boolean query (using bit vectors, *etc.*) require either space polynomial in the input size or search time linear in the input size [7]. Hence, it is of interest to efficiently

[1] 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.

perform selectivity estimation using 100% space, which motivates our FST case.

3. PROPOSED SOLUTION USING A 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.

Our goal is to dispense with the crude and unrealistic independence assumption. To do this, we propose a set-oriented approach to capture correlations by estimating selectivity using set operations (intersection, union, and difference). In Section 3.1, we show that if we could afford to keep the set of string identifiers (SIDs) that contain every substring in the database, then the selectivity could be exactly obtained (without error) by manipulating these sets. In practice there are two problems with this approach described below.

First, the size of the SID sets could be on the order of the number of strings in the database, which is infeasible in limited space. To solve this problem, we present a technique in Section 3.2 to succinctly represent the set of strings containing a given substring as a fixed-length signature of this set; the selectivity of any Boolean query can be estimated by manipulating these signatures, as shown in Sections 3.3 and 3.4. Our approach is space-efficient, requiring only $O(m)$ space to capture, theoretically, up to 2^{2^m} Boolean relationships between substring predicates, where m is the number of substrings stored in the data structure. In Section 3.5, we present a general algorithm based on this approach.

The second problem is that in practice we cannot afford to store every substring, that is, space allowed may be significantly smaller than the size of the input. This is the PST variation which we address in Section 4 using pruning and parsing strategies. We proceed with the description of our approach.

3.1 Set-Oriented Approach

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 labeling 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. Figure 1 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 $q =$ `ab` $\wedge$ `12`. Then the nodes j and k in the tree are located such that $w_j =$ `ab` and $w_k =$ `12` and $|S_j \cap S_k| = |\{1,2,4\} \cap \{1,2,3\}| = |\{1,2\}| = 2$ gives the result size.

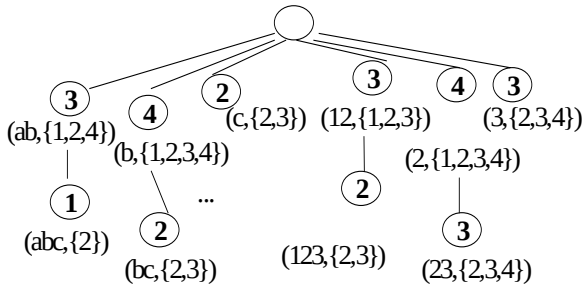


Figure 1: Set-oriented approach illustrated with a toy example with 4 strings {`ab12`, `abc123`, `bc123`, `ab23`}.

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 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.

Our key idea is to employ the Monte Carlo technique of *min-wise independent permutations* [2; 5]. 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 stored in much smaller space (typically 100 to 400 bytes) than the set itself. Second, the hash functions will serve to compute the result size of set operations over base sets. The catch, however, is that no hash functions exist that both use small space (significantly less than N) *and* provide exact count of the operations of any two sets.[2] Hence, we must settle for hash functions that allow some approximation in estimating the sizes for results of set operations. We use set hash signatures, as described in Section 3.2 for this purpose.

[2]This has been formalized and proved in the area of Communication Complexity [12].

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.

3.2 Estimating Set Resemblance

Min-wise independent permutations 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,

$$\rho = \frac{|A \bigcap B|}{|A \bigcup B|},$$

where, for a set S, the notation $|S|$ represents its cardinality. It was introduced by Cohen [5] and Broder et. al., [2]. This technique has been used for finding Web page duplicates [1], for data mining [6], and for estimating the size of transitive closure [5]

Suppose that "darts" are thrown randomly at the universe U. If two sets have high resemblance, then it is more likely that the first dart to hit one set will *simultaneously* hit the other; for low resemblance, the converse is true. Figure 2 illustrates this concept for two sets A and B.

To simulate this, we create signature vectors sig_A and sig_B. These signatures can be operated on directly to estimate ρ. The idea is to throw darts at universe U until an element of A is hit. When this occurs, the value of this element is recorded as a component value in the signature sig_A. This is repeated for each component. *Using the same dart throws*, we repeat the experiment now with B to generate sig_B. Finally, $\hat{\rho}$, an estimate for ρ, is determined from the number of respective signature vector components in sig_A and sig_B that match.

Following is a more detailed description of how we implement the set hash signature sig_A of A. For each signature vector component, we randomly permute the elements of the Universe (U) from which the sets are drawn and record the value of the first element of the permutation which is also an element of A.

Formally, let $U = \{1, \ldots, n\}$. If π is a permutation of U and $A \subseteq S$, we define $\min\{\pi(A)\} = \min\{\pi(x) | x \in A\}$. We choose $\pi_1, \ldots, \pi_\ell$, namely ℓ uniform, random permutations of U. For any set A, we define its signature vector as

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

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. Figure 3(a) and 3(b) illustrates this for sets A and B, respectively. Unfortunately, as reported in [2], 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 [1] and it is easy to generate a number of independent hash functions. Of course, each hash function h should be chosen so that the probability of collisions are low.

Our approach will be dependent on the *resemblance* measure defined as follows:

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

We can obtain $\hat{\rho_k}$, an estimate for ρ_k, using the formula

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

where ℓ is the number of hash functions we use in the definition of the signature vector for any set. It is not difficult to convince oneself of the following observation for any i:

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

Using this observation, it can be shown as in [5] that $\hat{\rho_k}$ is a good estimator for ρ_k for sufficiently large ℓ.

3.3 Extracting Result Sizes for Set Operations Using Set Hashing

Using the set signature described in the previous section, we will show how to estimate result sizes of set operations (intersection, union, and complements). More precisely, given sets A_i over Universe U, we wish to calculate sizes of the union, intersection or complements of the sets. Say the signature of each set A_i has been computed and we know the size of U, denoted $|U|$, as well as $|A_i|$ for each i. Our application scenario (of Boolean query selectivity estimation) will satisfy these assumptions.

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

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

That is, for each component, we choose the minimal value of the signatures of all the sets in that component. This is because, when computing each signature component of $A_1 \cup \cdots \cup A_k$, the minimum hash value over all elements in $A_1 \cup \cdots \cup A_k$, for a given hash function h, is precisely $\min\{\min h(a_1), \ldots, \min h(a_k)\}$, where $a_1 \in A_1, \cdots, a_k \in A_k$.

Estimating union without complements. Our procedure for estimating $|A_1 \cup \cdots \cup A_k|$ is as follows. Say A_j has the largest size of all A_i's.[3] We first calculate $sig_{A_1 \cup \cdots A_k}$ as described above. Using that, we obtain an estimate $\hat{\gamma}$ for

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

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

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

where we use $\hat{\gamma}$ for γ for estimation and $|A_j|$ is known.

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

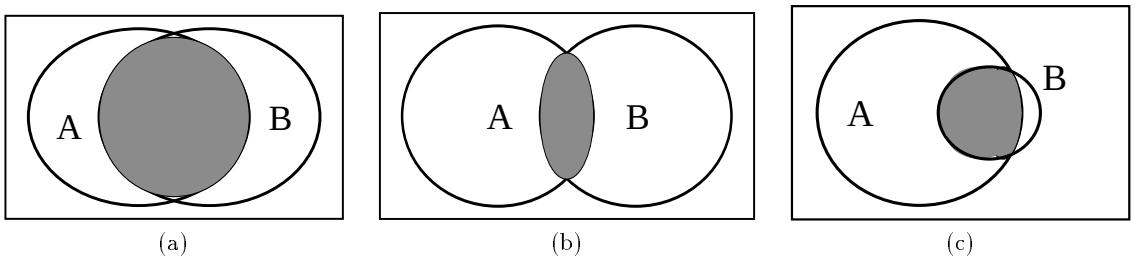


Figure 2: The main idea behind set resemblance: (a) high overlap, high resemblance; (b) low overlap, low resemblance; (c) high overlap, low resemblance.

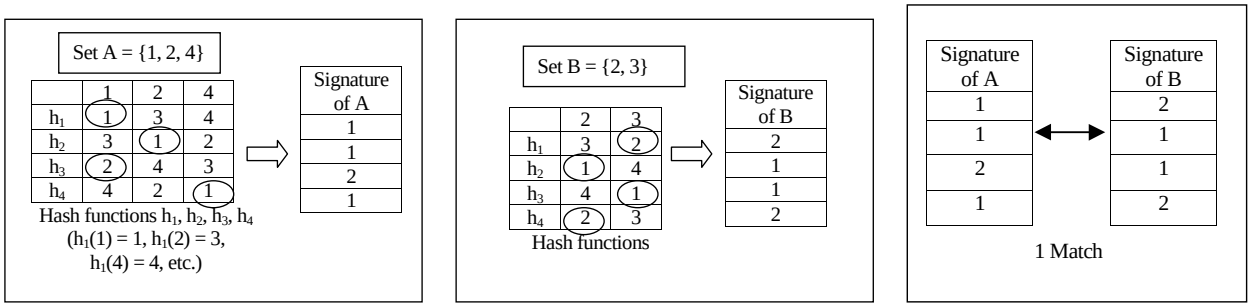


Figure 3: The generation and intersection of signatures of sets A and B.

Estimation intersection without complements. Our procedure for estimating $|A_1 \cap \cdots \cap A_k|$ is as follows:

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

We can estimate ρ_k as described in the previous section, γ as above, and $|A_j|$'s are known as we assumed.

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 above. We estimate $|A_1 \cup \cdots \cup A_k \cup \overline{A_{k+1}} \cup \cdots \overline{A_\ell}|$ as its complement $|\overline{A_1} \cap \cdots \cap \overline{A_k} \cap A_{k+1} \cap \cdots A_\ell|$, which we have already described how to estimate. □

That completes the description of how to estimate set operations of our interest; these will be used in the next section for Boolean query selectivity estimation.

3.4 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, $1 \le i \le n, 1 \le j \le k_i$ will be located in T. Let $sig_{\sigma_{ij}}$ be the signature at the node that σ_{ij} is located. From these signatures, we can derive the signature $sig_{(\sigma_{i1} \vee \cdots \vee \sigma_{ik_i})}$ for each disjunctive clause $(\sigma_{i1} \vee \cdots \vee \sigma_{ik_i})$ and then estimate the selectivity of q from their intersection size, which we can determine as described in the previous section. □

2. q **contains negations:** We convert q to the DNF expression $q' = (\sigma_{11} \wedge \cdots \wedge \sigma_{1l_1}) \vee \cdots \vee (\sigma_{m1} \wedge \cdots \wedge \sigma_{ml_m})$. 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 $y_i, z_j, 1 \le i \le r, 1 \le j \le t$ 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 3.3. □

3.5 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 FST T of the set S of strings. For each node v in T, we store the signature sig_{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 3.4. These result sizes can be estimated using the signatures of these sets via set hashing, as described in Section 3.3.

Let us consider the complexity of the algorithm. Construction of the suffix tree takes time $O(\sum_{s\in S}|s|)$, a classical result [13]; 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 3.3, in which case, the time taken is $2^{O(L)}$; this is the overall complexity of Step 2.

THEOREM 1. *The algorithm in Section 3.4 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. A query 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 3.3 very efficiently. Also, in practice, it suffices to use $\ell \leq 200$ bytes. For a more detailed experimental study of this algorithm, see Section 5.

4. PROPOSED SOLUTION USING A 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 [19; 10] to estimate the overall selectivity of the query.

The previous study in substring selectivity estimation [11; 19; 10; 8] 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 [10; 8]. The detail of MO parsing can be found in [10; 8]. The algorithm is as follows.

4.1 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 strategies in [19; 8]. With each node in the PST, we store the exact count of the number of string IDs of that substring (labeling 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 Figure 4.

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 3.4 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 3.4 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 than one term is present, add and subtract terms (as described in Section 3.4 Case 2) to derive the overall selectivity of q.

Figure 4: Estimating Selectivities of Boolean queries using a PST

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 3.4 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. $|q| = 1 - |23| + |abc \wedge 12 \wedge 23| = 1 - 0.75 + 0.26 = 0.51$.

5. EXPERIMENTAL RESULTS

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 the ID method, we follow the algorithm of figure 4 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. In the SH method, the nodes are also augmented with signature vectors.

5.1 Experimental Setup

Data Sets: Due to space limitations, we report only some of our experimental results. We give results from an AT&T data set of 130K strings (2.5MB) which we refer to as SERVICE; the strings contain brief English descriptions of service provided to AT&T customers.

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 chosen from the database, 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.

Error Measures: Our workload Q consists of 1000 queries according to each query template. Let S_q be the true selectivity of a query in Q, S'_q the estimate and N the number of strings in S. We use the average-absolute-relative-error to quantify the accuracy for positive queries:

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

We use the root-mean-square-error to quantify the accuracy of negative queries:

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

Both measures are standard in the substring selectivity estimation literature.

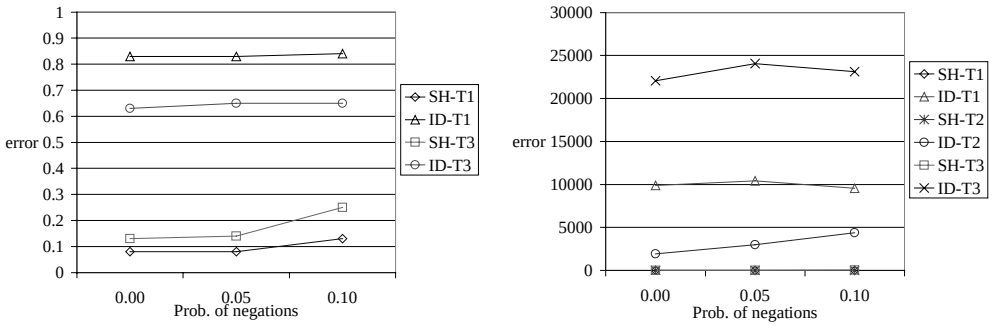
Implementation: All experiments were run on a 350 Mhz Pentium II PC. We implemented the SH and ID methods in C++. For the pruned suffix tree (PST) case, we used MO parsing in both methods. The set hash signature sizes were set at 50 with a hash space of 2^{17}.

5.2 Estimation Accuracy Using A FST

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.[4] Figure 5(a) presents 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 millisecond 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.

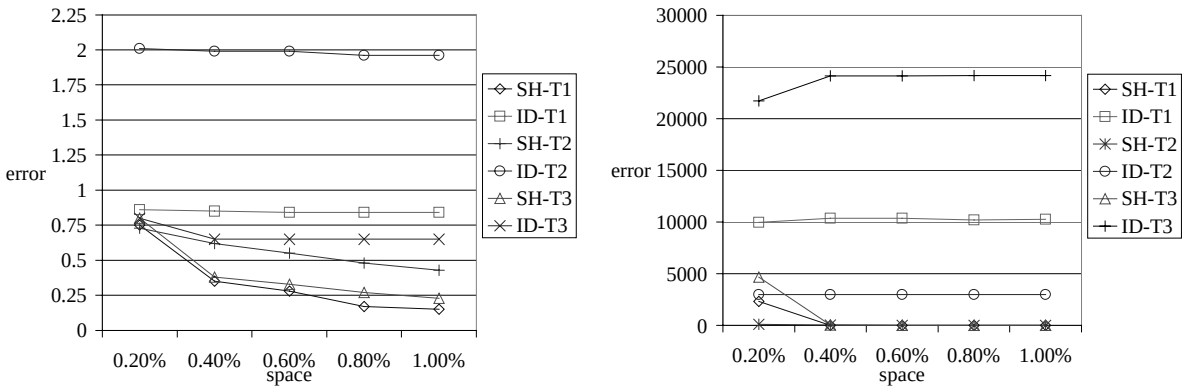
SH experiences a small increase in estimation error as the number of predicates per clause and the probability of negations increases, since, due to DNF conversion, more set hash signatures are involved in the estimation. As the probability of negation increases, ID constantly overestimates the true 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 Figure 5(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 Figure 5(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

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



(a) Queries generated by T1 and T3 (b) Accuracy for Negative Queries

Figure 5: Accuracy for positive and negative queries (FST)



(a) Positive Queries (b) Negative Queries

Figure 6: Accuracy for positive and negative queries (PST)

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.

Figure 5(b) presents the results of the same experiment for negative queries. 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. Note that the three curves for the SH methods are all overlapping at the bottom near 0.

5.3 Estimation Accuracy Using A PST

We pruned both suffix trees to the same amount of space and we report on the estimation accuracy of the methods. Figure 6 presents 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. Figure 6(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 consistently (by up to 6 times) as the number of predicates per clause increases and as the number of clauses in the query increases.

Figure 6(b) presents the results of the same experiment but for negative queries. 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.

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.

6. RELATED WORK

The problem of estimating substring selectivities is defined as follows: Given $p_1, \ldots p_k$ substring predicates and a relation R with k columns containing N k-dimensional tuples, we are interested in estimating the selectivity of $Q = p_1 \wedge \ldots \wedge p_k$, that is, the fraction of tuples from R that satisfy Q. Selectivity estimation tailored to alphanumeric strings has been the subject of recent work in which the problem of estimating the selectivity of both one- and two-dimensional substrings has been studied [11; 10; 19; 8].

In [11], the problem of substring selectivity estimation was

introduced. An approach based on pruned suffix trees was presented wherein queries are parsed via a greedy strategy into substrings retained in the pruned suffix tree, and the selectivities of these substrings are multiplied based on the independence assumption to derive selectivity estimates. In [10], the concept of conditioning based on maximal overlap parsing was introduced for improved estimation. Selectivity estimation of substrings over multiple attributes was first considered in [19], and was later improved upon in [8]. Although in principle the approaches of [19; 8] extend to multiple dimensions, only experiments with 2 dimensional string data were reported.

In [3], we addressed the multidimensional substring selectivity estimation problem, and initiated the use of the set hashing technique for selectivity estimation. There, set hashing was used to capture the co-occurrences of substrings across attributes. We showed experimentally that our approach is superior to the approaches of [19; 8] both in accuracy and scalability to the number of dimensions. The work presented in [3] essentially deals only with conjunctions of predicates. In this paper, the set hashing framework is considered in a more general context. We extend it to deal with additional logical connectives and in particular, we consider the problem of estimating the selectivity of Boolean queries containing conjunctions, disjunctions and negations. From a theoretical point of view, we expand the power of set hashing to its full potential.

Search methods for text retrieval systems have been a topic of interest for many years. These methods provide indexing capabilities to retrieve the actual strings satisfying a Boolean query (and thus the exact count). However, they either use super-linear space, or take time proportional to the number of strings in S for answering any query [14; 7]. In contrast, our approach can be tuned to any given space constraint and it provides good approximate answers in time dependent only on the query size, and not on the size of S; it takes less than a millisecond per query.

7. CONCLUSIONS

In this paper, we generalize the problem of substring selectivity estimation for Boolean predicates. Our novel idea is to capture correlations between Boolean query predicates in a space-efficient but approximate manner. We employ a Monte Carlo technique called set hashing to succinctly represent the set of strings containing a given substring predicate as a signature vector of hash values. Correlations among substring predicates can then be generated by operating on these signatures. We present an algorithm to estimate the selectivity of *any* Boolean query and experimentally demonstrate the superiority of our approach.

Several important issues are raised by this study. First, although we consider only one-dimensional strings in this paper for simplicity, our approach can be extended to multiple string dimensions in a straightforward manner; the effectiveness in multiple dimensions remains to be seen. Second, we do not consider even more general query algebras, such as Boolean queries over string 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, that is, matching the pattern $\sigma_1[< space > | < tab > | < newline >]^*\sigma_2$. Third, 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 [14].

Acknowledgments

We would like to thank Derek Jeter, Chuck Knoblauch, and Mariano Rivera.

8. REFERENCES

[1] A. Broder. On the Resemblance and Containment of Documents. *IEEE SEQUENCES '97*, pages 21–29, 1998.

[2] A. Broder, M. Charkar, A. Frieze, and M. Mitzenmacher. Minwise Independent Permutations. *Proceedings of STOC*, pages 327–336, 1998.

[3] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Approximating Cross-Counts: A Practical, Space Efficient Approach For Multidimensional Substring Selectivity Estimation. *AT&T Labs Technical Report*, Oct. 1999.

[4] C. Clarke and G. Cormack. Relevance Ranking For One To Three Term Queries. *Information Processing And Management*, page to appear, May 1999.

[5] E. Cohen. Size-Estimation Framework With Applications To Transitive Closure And Reachability. *Journal Of Comput. Syst. Sciences, 55*, pages 441–453, 1997.

[6] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. . Motwani, J. Ullman, and C. Yang. Finding Interesting Associations Without Support Prunning. *Proceedings of the 16th Annual IEEE Conference on Data Engineering (ICDE 2000)*, page to appear, Feb. 2000.

[7] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[8] H. V. Jagadish, O. Kapitskaia, R. Ng, and D. Srivastava. Multidimensional Substring Selectivity Estimation. *Proceedings of VLDB, Endiburgh, Scotland*, pages 287–398, Sept. 1999.

[9] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. *Proceedings of VLDB*, pages 275–286, Aug. 1998.

[10] H. V. Jagadish, R. Ng, and D. Srivastava. Substring Selectivity Estimation. *ACM Principles of Database Systems (PODS)*, pages 249–260, June 1999.

[11] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating Alphanumeric Selectivity In The Presense Of Wildcards. *Proceedings of SIGMOD, Montreal Canada*, pages 282–293, June 1996.

[12] L. Lovasz. Communication complexity: a survey. *Paths, Flows and VLSI Layout, B. Korte, L. Lovasz, H. Promel and A. Schrijver, Ed., Springer-Verlag*, 1990.

[13] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM Vol 23.*, pages 262–272, Dec. 1976.

[14] A. Salminen and F. W. Tompa. PAT Expressions: An Algebra For Text Search. *Acta Linguistica Hungarica, 41-4*, pages 177–306, May 1994.

[15] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

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

[17] E. Tanin, R. Beigel, and B. Shneiderman. Design and evaluation of incremental data structures and algorithms for dynamic query interfaces. In *Proceedings of IEEE InvoViz '97*, Phoenix, AZ, Oct. 1997.

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

[19] M. Wang, J. S. Vitter, and B. Iyer. Selectivity Estimation In The Presence Of Alphanumeric Correlations. *Proceedings of ICDE*, pages 169–180, 1997.