Extending Q-Grams to Estimate Selectivity of String Matching with Low Edit Distance *

Hongrae Lee
Univ. of British Columbia
xguy@cs.ubc.ca

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

Kyuseok Shim
Seoul National Univ.
shim@ee.snu.ac.kr

ABSTRACT

There are many emerging database applications that require accurate selectivity estimation of approximate string matching queries. Edit distance is one of the most commonly used string similarity measures. In this paper, we study the problem of estimating selectivity of string matching with low edit distance. Our framework is based on extending q-grams with wildcards. Based on the concepts of replacement semi-lattice, string hierarchy and a combinatorial analysis, we develop the formulas for selectivity estimation and provide the algorithm *BasicEQ*. We next develop the algorithm *OptEQ* by enhancing *BasicEQ* with two novel improvements. Finally we show a comprehensive set of experiments using three benchmarks comparing *OptEQ* with the state-of-the-art method *SEPIA*. Our experimental results show that *OptEQ* delivers more accurate selectivity estimations.

1. INTRODUCTION

With the widespread use of the internet, text-based data sources have become ubiquitous. The demand for effective support of approximate string queries becomes ever increasing. For example, if a keyword is misspelled in a web search, candidate corrections are suggested. Furthermore, many database applications such as data integration or data cleaning require effective handling of approximate string matching and joins.

In query optimization, the estimation of selectivity for selection predicates or joins is necessary to produce an optimized query execution plan. For example, a query with two conjunctive selection predicates, one of which may be a LIKE predicate, may result in dramatically different processing time depending on the order the two selections are executed. Processing the more selective predicate first is more advantageous if the costs of checking a record for both predicates are the same. Thus, accurate estimation of the selectivity of approximate string matching is essential for many database applications.

To handle approximate string matching, various string similarity measures, such as the edit distance, hamming distance and Jaccard distance distance have been considered [3]. As one of the most widely accepted measures in text retrieval [3, 15], the edit distance between two strings s_1 and s_2, denoted as $ed(s_1, s_2)$, is the minimum number of edit operations of single characters that are needed to transform s_1 to s_2. The problem statement of the paper is as follows: *Given a query string s_q and a bag of strings DB, estimate the size of the answer set $\{s \mid ed(s_q, s) \leq \Delta$ and $s \in DB\}$*, where Δ is the edit distance threshold.

In [6], Chaudhuri et al. proposed the Short Identifying Substring (SIS) assumption stating that: "*a query string s usually has a 'short' substring s' such that if an attribute value contains s', then the attribute value almost always contains s as well.*" Thus, for estimating the frequency of s, it is desirable to estimate instead the frequency of s'. They show that the SIS assumption appears to hold for long strings in many real life data sets. Specifically, if one defines s' to be a short identifying substring when its selectivity is within 5% of the selectivity of s, then there exist s' with length ≤ 7 for over 90% of strings s of longer lengths. Thus, in this paper, we focus our attention on queries s_q with length shorter than 40. Given such queries s_q, we restrict our attention to low edit distance, more specifically $\Delta = 1, 2$ or 3. For instance, if $s_q =$ "sheffey", the following seemingly unrelated strings are within an edit distance of 3: 'hefte', 'yeffet', 'elfey', etc. Thus, a fast solution for $\Delta \leq 3$ can be valuable for many database applications. As a preview, we make the following contributions.

- We propose in Section 3 the concept of extended q-grams, which can contain the wildcard symbol ?, representing any single character from the alphabet. Based on the concepts of replacement semi-lattice, string hierarchy and a combinatorial analysis, we develop in Section 3 the formulas for determining the selectivity when only replacements or deletions are allowed. The replacement formula is particularly valuable as it forms the basis for later optimizations.
- We propose in Section 4 an algorithm called *BasicEQ* for selectivity estimation when all edit operations are allowed. We then develop in Section 5 novel techniques to scale up *BasicEQ* and present the algorithm *OptEQ*. The first technique is done by approximat-

*This research is sponsored by funding from Canadian NSERC and Genome Canada. It is also supported by the Ministry of Information and Communication, Korea, under the College Information Technology Research Center Support Program, grant number IITA-2006-C1090-0603-0031.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

ing with a completion of the appropriate replacement semi-lattice. The second one is done by avoiding unnecessary operations by grouping semi-lattices.

- We show a comprehensive set of experiments in Section 6. We compare OptEQ with $SEPIA$ using three benchmarks of varying difficulty levels. Even with less memory, our selectivity estimations are more accurate when $\Delta \leq 3$.

2. RELATED WORK

Jin and Li studied the problem of estimating string selectivity with edit distance [15]. Their technique $SEPIA$ clusters similar strings, selects a pivot for each cluster, and captures the edit distance distribution with histograms. Given a query, it visits each cluster and estimates the number of strings in the cluster that are within the edit distance threshold using the the histograms. In comparison, the algorithms proposed here are based on extended q-grams, which are less dependent on data size. A comprehensive comparison with $SEPIA$ is given in Section 6.

Krishnan et al. first proposed using suffix trees for substring selectivity estimation [17]. The KVI estimate, which assumes independence of substrings, was proposed. Based on the Markov assumption and the concept of maximally overlapping substrings, Jagadish et al. proposed the MO estimate [14]. Chaudhuri et al. observed that MO often under-estimates [6]. Based on the SIS assumption, they proposed the CRT algorithm, which uses q-gram tables and regression trees. All of these estimates do not deal with edit distance.

Approximate string matching has widely been studied across various fields including computational biology [12], signal processing [9], text retrieval [3, 21], and data cleaning [22]. Jin et al. proposed MAT trees to process fuzzy predicates on strings [16]. Chaudhuri et al. developed an index structure and a fuzzy matching algorithm to effectively filter incoming tuples [7]. Estimating the cardinality of approximate string predicates is essential in all these tasks.

Burkhardt et al. used q-grams to match patterns in DNA sequences [5]. Gravano et al. applied q-grams and proposed positional q-grams to build approximate string join capabilities [11]. Burkhardt and Karkkianen proposed one-gapped q-gram filters to efficiently filter out strings using edit distance for query processing (but not for selectivity estimation) [4]. The q-gram is made by skipping some positions and is similar to a special case of our extended q-grams.

Apart from the edit distance, Jaccard distance [8] and Jaro-Winkler distance [23] were proposed for text retrieval and 'record linkage' respectively. The framework proposed here is designed specifically for edit and hamming distance. Extensions to handle Jaccard and Jaro-Winkler distances are beyond the scope of this paper.

3. EXTENDED Q-GRAMS

3.1 Extending Q-grams with Wildcards

Let Σ be a finite alphabet with size denoted as $|\Sigma|$. The bag DB consists of strings drawn from $\Sigma^\star$. To mark the beginning and the end of a string, we use the symbols # and \$, which are assumed not to be in Σ. For a string s, we prefix s with # and suffix with \$. For example, "beau" is transformed into "#beau\$" before processing. Throughout this paper, we use double quotes for strings, but do not use any quotation marks for substrings. Whenever there is no confusion, we omit # and \$ for clarity.

Any string of length $q(>0)$ in $\Sigma \cup \{\#, \$\}$ is called a *q-gram*. An *N-gram table* over DB consists of the frequencies of all q-grams for $1 \leq q \leq N$ [6]. If an entry s in the table contains both # and \$, the entry gives the frequency of the strings in DB that are exactly s. Otherwise, the entry gives the frequency of the strings that contain s as a substring.

For edit distance, edit operations considered are deletion (D), insertion (I) and replacement (R). Given a query string s_q, we use $Ans(s_q, iDjImR)$ to denote the set of strings s' such that s_q can be converted to s' with at most $i(\geq 0)$ D(eletion) operations, $j(\geq 0)$ I(nsertion) operations and $m(\geq 0)$ R(eplacement) operations *applied in any order*. For example, $Ans(\text{"}abcd\text{"}, 2D1R)$ denotes the set of strings that can be converted from "abcd" with at most two deletions and one replacement applied in any order. The notation $2D0I1R$ is simplified to just $2D1R$. We also use the notation $Ans(s_q, k)$ to denote the set of strings s' such that s_q can be converted to s' with at most k operations when deletion, insertion and replacement are allowed, i.e.,

$$Ans(s_q, k) = \bigcup_{i+j+m=k \text{ and } i,j,m \geq 0} Ans(s, iDjImR).$$

For instance, $Ans(\text{"}abcd\text{"}, 1R)$ consists of the strings of one of the following forms: "?bcd", "a?cd", "ab?d" or "abc?", where the wildcard symbol ? denotes a single character from Σ. To estimate the frequency for string matching with edit distance, we extend q-grams with the wildcard symbol. Any string of length $q > 0$ in $\Sigma \cup \{\#, \$, ?\}$ is called an *extended q-gram*. We generalize an N-gram table by keeping extended q-grams, and call it an *extended N-gram table*.

For instance, a 3-gram table for the string "beau" contains 1-gram (i.e., for b, e, a, u individually), 2-grams (i.e., #b, be, ea, au, u\$), and 3-grams (i.e., #be, bea, eau, au\$). In an extended 3-gram table, the additional 2-grams are ?b, ?e, ?a, ?u, b?, e?, a?, u?, ??, #? and ?\$. The additional 3-grams include (non-exhaustively) ?ea, #?e, etc. The entry ?ea gives the frequency of strings that include a substring of length 3 ending in ea. The entry #?e gives the frequency of strings that begin with the form ?e.

3.2 Replacement Semi-lattice

We introduce below a *replacement semi-lattice* and show how this structure can be used to derive a formula for estimating frequencies when only replacements are allowed. We will also show how this formula can be exploited by the optimized algorithm OptEQ in Section 5.

Consider the set $Ans(\text{"}abcd\text{"}, 2R)$, which consists of strings of one of the following forms: "ab??", "a?c?", "?bc?", "a??d", "?b?d" and "??cd". Hereafter, we refer to these as the *base strings* of $Ans(\text{"}abcd\text{"}, 2R)$. Formally, a base string for a query string s_q and the edit distance threshold k is any string b such that $ed(b, s_q) = k$ and insertion/replacement modelled by '?'. Intuitively, base strings represent possible forms of strings satisfying the query.

Let S_1 consists of strings of the form "ab??", and S_2 of the form "a?c?", and so on, then $Ans(\text{"}abcd\text{"}, 2R) = S_1 \cup \ldots \cup S_6$. Thus, the cardinality of $Ans(\text{"}abcd\text{"}, 2R)$ can be computed using the inclusion-exclusion principle:

$$|S_1 \cup S_2 \cup \ldots \cup S_n| = \Sigma|S_i| - \Sigma|S_i \cap S_j| + \Sigma|S_i \cap S_j \cap S_k| - \ldots + (-1)^{n-1}|S_1 \cap S_2 \cap \ldots \cap S_n|$$

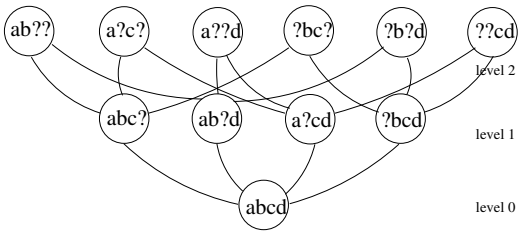


Figure 1: String Semi-lattice for $Ans(\text{“}abcd\text{”}, 2R)$

The problem is that this formula requires a number of terms which is exponential with respect to n. Fortunately, as shown below, the structure of character replacements helps to collapse the formula significantly.

The structure of character replacements is captured in a replacement semi-lattice. Figure 1 shows the semi-lattice for $Ans(\text{“}abcd\text{”}, 2R)$. A node in the lattice represents a base string or any string that is a result of intersection among base strings. The intersection is defined as follows; (i) the intersection of ? and? gives ?; (ii) the intersection of ? and a character c gives c; and (iii) the intersection of two characters c_1, c_2 is c_1 if $c_1 = c_2$, and empty if $c_1 \neq c_2$. For example, intersection between “ab??” and “a?c?” gives “abc?”. Union operation is defined similarly. We define the *level* of a node to be the number of wildcard symbols in the node. Each of the aforementioned sets $S_1, \ldots, S_6$ is represented as a level-2 node. An edge in the semi-lattice represents an inclusion relationship between nodes in adjacent levels. For example, there in an edge between the level-2 node “ab??” and the level-1 node “abc?” and it indicates that the set S_7 consisting of strings of the form “abc?” is a subset of S_1 corresponding to “ab??”. Note that it is also a subset of S_2 and S_4. Six nodes at the top row(i.e., level-2 nodes here) in the lattice are base strings, and they give rise to four level-1 nodes, which correspond to $Ans(\text{“}abcd''\text{”}, 1R)$. In turn, they are parents of the single level-0 node “abcd”, which is the bottom element of the semi-lattice. As shown later, we exploit the property of a semi-lattice that the intersection between any two nodes results in a node at a lower level, decreasing the occurrence of wildcards by at least one.

3.3 An Example Replacement Formula

Continuing the example of $Ans(\text{“}abcd\text{”}, 2R)$, there are 6 level-2 nodes, thus there are $\binom{6}{2} = 15$ ways of choosing two nodes for intersections between two nodes. We call these 2-intersections. An ℓ-intersection is any intersection of ℓ base strings. Among the fifteen 2-intersections, twelve correspond to level-1 nodes (e.g., “ab??” ∩ “a?c?” = “abc?”), but three correspond to the level-0 node of “abcd” (e.g., “ab??” ∩ “??cd” = “abcd”).

Let us continue with the 3-intersections of the six level-2 nodes. Some of the 3-intersections correspond to level-1 nodes. For example, the 3-intersection of “ab??”, “a?c?” and “?bc?” gives “abc?”. Among the $\binom{6}{3} = 20$ 3-intersections, four result in level-1 nodes, with the remaining sixteen resulting in the level-0 node. The table shown in Figure 2 also summarizes the numbers for 4-, 5- and 6-intersections. In the table, the row for level-1 indicates that twelve 2-intersections and four 3-intersections of base wild strings result in a level-1 node. One important observation is that given the highly regular structure of the semi-lattice, each

	2-inter.	3-inter.	4-inter.	5-inter.	6-inter.
# at level-1	12	4	0	0	0
# at level-0	3	16	15	6	1
Total	15= $\binom{6}{2}$	20= $\binom{6}{3}$	15= $\binom{6}{4}$	6	1

Figure 2: Number of resulting intersections for $Ans(\text{“}abcd\text{”}, 2R)$

level-1 node appears as results in exactly the same number of times. That is, among the twelve 2-intersections over four level-1 nodes, each level-1 node is the result of a 2-intersection three (= 12/4) times. Similarly, among the four 3-intersections over four level-1 nodes, each level-1 node is the result of a 3-intersection once.

We can now calculate the size of $Ans(\text{“}abcd\text{”}, 2R)$. Given a string s, we abuse notation by using $|s|$ to denote the size of the set $\{t | t \in DB$ and t is a string obtained by replacing all wildcards, if any, in s with characters in $\Sigma\}$. By using the inclusion-exclusion principle and the numbers given in Figure 2 to simplify the calculation, we have:

$$
\begin{aligned}
&|\text{“}ab??\text{”} \cup \text{“}a?c?\text{”} \cup \text{“}a??d\text{”} \cup \text{“}?bc?\text{”} \cup \text{“}?b?d\text{”} \cup \text{“}??cd\text{”}| \\
&= |\text{“}ab??\text{”}| + |\text{“}a?c?\text{”}| + \ldots + |\text{“}??cd\text{”}| \\
&+(-3+1)(|\text{“}abc?\text{”}| + |\text{“}ab?d\text{”}| + |\text{“}a?cd\text{”}| + |\text{“}?bcd\text{”}|) \\
&+(-3+16-15+6-1)|\text{“}abcd\text{”}| \\
&= F_2 + (-3+1)F_1 + (-3+16-15+6-1)F_0
\end{aligned}
$$

where F_i denotes the sum of the frequencies of *all* the level-i nodes. Thus, with the analysis on the replacement semi-lattice, the formula is significantly simplified.

3.4 The General Replacement Formula

Below we generalize this analysis to give the cardinality of $Ans(s_q, kR)$, where the length of s_q is ℓ. There are $\binom{\ell}{k}$ base strings, i.e., the number of strings with exactly k characters replaced by ? in s_q. Let $B_{\ell,k,r}$ denote the number of r-intersections ($2 \leq r \leq \binom{\ell}{k}$) among the $\binom{\ell}{k}$ base strings. It is easy to see that:

$$B_{\ell,k,r} = \binom{\binom{\ell}{k}}{r} \quad (1)$$

Among these r-intersections, let $D_{\ell,k,r}$ denote the number of those that give s_q, i.e., there is no wildcard contained in the r-intersection. For our example of $Ans(\text{“}abcd\text{”}, 2R)$, we have $\ell = 4$ and $k = 2$. Thus, there are $B_{4,2,2} = 15$ 2-intersections, $B_{4,2,3} = 20$ 3-intersections and so on (cf: the last row in Figure 2). Then $D_{4,2,2}$, $D_{4,2,3}$ and so on correspond to the second last row of the table.

Let us take a closer look into $D_{4,2,2}$ which is the number of 2-intersections giving s_q. Since performing intersections of base strings always decreases the number of wildcards in the result string, the intersections for $B_{4,2,2}$ can only contain strings with zero or one wildcard position. Thus, $D_{4,2,2}$ is equal to subtracting from the total number of 2-intersections, which is given by $B_{4,2,2}$, the number of 2-intersections with 1 wildcard position in the intersection.

Let s_1 and s_2 be the two non-identical base strings forming such a 2-intersection. Let the wildcard positions of s_1 be $p_{1,1}$ and $p_{1,2}$ and those of s_2 be $p_{2,1}$ and $p_{2,2}$. The two base strings must agree on exactly one wildcard position. Thus, without loss of generality, we can assume that $p_{1,1} = p_{2,1}$

and $p_{1,2} \neq p_{2,2}$. There are $\binom{4}{1}$ combinations for choosing the common wildcard position $p_{1,1}$. For the remaining positions, there must be 1 wildcard for s_1 and 1 wildcard for s_2 in different positions since $p_{1,2} \neq p_{2,2}$. This is exactly as if we were counting the number of 2-intersections between s'_1 and s'_2 which were strings of length 3 with 1 wildcard character in different positions in each. This is $B_{3,1,2}$, which automatically guarantees that $p_{1,2} \neq p_{2,2}$ because the base strings always have different wildcard positions (i.e. $p_{1,2} \neq p_{2,2}$). Thus, in sum, $D_{4,2,2} = B_{4,2,2} - \binom{4}{1} * B_{3,1,2}$. By using Eqn. (1) for the B terms, $D_{4,2,2} = 15 - 4 * 3 = 3$ (cf: the 2-intersection column in Figure 2). By a similar analysis, $D_{4,2,3}$ is given by $B_{4,2,3} - \binom{4}{1} * B_{3,1,3}$, which gives $D_{4,2,3} = 20 - 4 * 1 = 16$ (cf: the 3-intersection column). We have the following proposition for the general case. A proof is included in the full version of the paper.

PROPOSITION 3.1: *The general equation of $D_{\ell,k,r}$ is:*

$$D_{\ell,k,r} = \sum_{i=0}^{i<k} (-1)^i * \binom{\ell}{i} * B_{\ell-i,k-i,r} \quad (2)$$

Recall that $D_{\ell,k,r}$ gives the number of r-intersections with no wildcard contained in the intersection, i.e., level-0. We can apply the same reasoning for a node in level-i ($1 \leq i \leq k$). A node in level-i has exactly i wildcards. There are $\binom{\ell}{i}$ possibilities to pick these i wildcard positions. For the remaining positions, there must not be any wildcard in the r-intersection. Thus, $\binom{\ell}{i} * D_{\ell-i,k-i,r}$ counts the total number of r-intersections resulting in a node in level-i. Recall that due to the regular structure of the semi-lattice, each node in this level occurs the same number of times as the result of r-intersections. Because there are $\binom{\ell}{i}$ nodes in level-i, dividing $\binom{\ell}{i} * D_{\ell-i,k-i,r}$ by $\binom{\ell}{i}$ gives the number of r-intersections resulting in a specific node in level-i, which is $D_{\ell-i,k-i,r}$. This explains the intuition behind the following proposition. Recall that F_i gives the sum of the frequencies of *all* the level-i nodes. The alternating sign comes from the inclusion-exclusion principle.

PROPOSITION 3.2.: *The cardinality of $Ans(s_q, kR)$ is given by:*

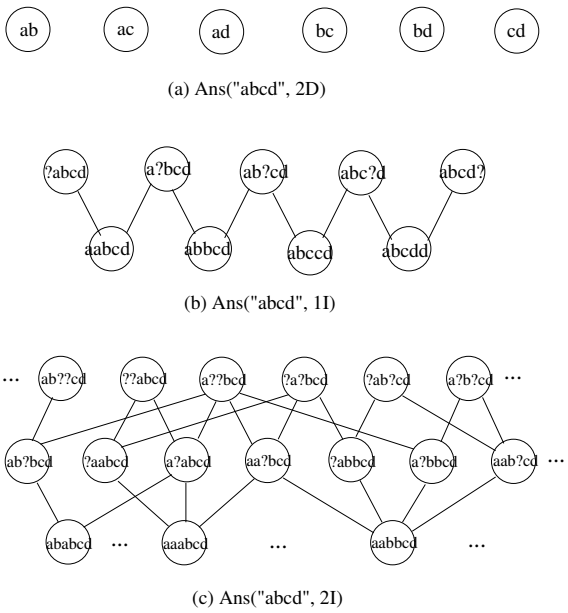
$$|Ans(s_q, kR)| = F_k + \sum_{i=0}^{k-1} F_i * \sum_{r=2}^{\binom{l}{k}} (-1)^{r+1} D_{l-i,k-i,r} \quad (3)$$

As a side benefit, this formula covers the situation when the hamming distance is used for similarity matching. For small edit distance threshold k and l within, say, 40, the magnitude of these terms is manageable computationally. In our implementation, we use Java's BigInteger package for high precision.

3.5 The General Deletion Formula

Deletion operations do not insert any wildcard. This makes the string hierarchy and computation quite trivial. We use the example $Ans(\text{“}abcd\text{”}, 2D)$ as an illustration. Figure 3(a) gives the string hierarchy. There are six base strings, as the strings in $Ans(\text{“}abcd\text{”}, 2D)$ must be one of “ab”, “ac”, “ad”, “bc”, “bd” and “cd”. Notice that as no wildcards are involved, any r-intersection ($r \geq 2$) gives the empty set. Thus, the hierarchy becomes a trivial single-level hierarchy.

$$\begin{aligned}
&|Ans(\text{“}abcd\text{”}, 2D)| \\
&= |\text{“}ab\text{”} \cup \text{“}ac\text{”} \cup \text{“}ad\text{”} \cup \text{“}bc\text{”} \cup \text{“}bd\text{”} \cup \text{“}cd\text{”}| \\
&= |\text{“}ab\text{”}| + |\text{“}ac\text{”}| + |\text{“}ad\text{”}| + |\text{“}bc\text{”}| + |\text{“}bd\text{”}| + |\text{“}cd\text{”}| \\
&= F_0
\end{aligned}$$



(a) Ans("abcd", 2D)

(b) Ans("abcd", 1I)

(c) Ans("abcd", 2I)

Figure 3: Various representative string hierarchies

where F_0 gives the sum of the frequencies of all the level-0 nodes; This result extends to the general case: $|Ans(s_q, kD)|$ is equal to F_0.

3.6 The Formula for One Insertion

Next we turn to insertion operations. We begin with the example $Ans(\text{“}abcd\text{”}, 1I)$. The string hierarchy is shown in Figure 3(b). There are five base strings, representing all the possible positions for the insertion. Note that only the 2-intersections between two adjacent base strings are non-empty.

$$\begin{aligned}
&|Ans(\text{“}abcd\text{”}, 1I)| \\
&= |\text{“}?abcd\text{”} \cup \text{“}a?bcd\text{”} \cup \text{“}ab?cd\text{”} \cup \text{“}abc?d\text{”} \cup \text{“}abcd?\text{”}| \\
&= |\text{“}?abcd\text{”}| + |\text{“}a?bcd\text{”}| + |\text{“}ab?cd\text{”}| + |\text{“}abc?d\text{”}| + |\text{“}abcd?\text{”}| \\
&-(|\text{“}aabcd\text{”}| + |\text{“}abbcd\text{”}| + |\text{“}abccd\text{”}| + |\text{“}abcdd\text{”}|) \\
&= F_1 - F_0
\end{aligned}$$

This extends to the general case: the size of $Ans(s_q, 1I)$ is given by $F_1 - F_0$.

The analysis for $Ans(s_q, kI)$ for $k > 1$ can be complex. Figure 3(c) shows the string hierarchy for $Ans(\text{“}abcd\text{”}, 2I)$. The structure of the hierarchy is less uniform than that of a replacement semi-lattice. This motivates the machinery to be presented next. Specifically, we tackle the general case where:

$$Ans(s_q, k) = \bigcup_{i+j+m=k \text{ and } i,j,m \geq 0} Ans(s, iDjImR).$$

We develop the basic algorithm *BasicEQ* to estimate the size of $Ans(s_q, k)$.

4. BASIC ALGORITHM FOR SIZE ESTIMATION

4.1 Procedure BasicEQ and Generating Nodes of the String Hierarchy

To estimate the size of $Ans(s_q, k)$, the strategy is to partition $Ans(s_q, k)$ by the length of the strings in $Ans(s_q, k)$. Specifically, if the length of s_q is l, then the strings in $Ans(s_q, k)$ vary in length from $(l-k)$ to $(l+k)$. Thus, $Ans(s_q, k)$ can be partitioned – according to the length of strings – into $(2k+1)$

Input: query s_q of length l, edit distance threshold k
Output: estimated frequency c

1: $c = 0$
2: **for** $len = l - k$ to $l + k$ **do**
3: find all combinations (i, j, m) for $Ans(s_q, iDjImR)$ such that $i + j + m = k$ and $l - i + j = len$
4: **if** $Ans(s_q, iDjImR)$ is one of the special cases **then**
5: get c' from the corresponding algorithm
6: **else**
7: $c' = BasicEQ(s_q, k, len)$
8: **end if**
9: $c = c + c'$
10: **end for**
11: **return** c

Figure 4: A Skeleton for Estimating Frequency

non-overlapping subsets. The size of each of these $(2k + 1)$ partitions is then estimated. Figure 4 shows a skeleton of this algorithm.

To illustrate, consider $Ans(\text{“}abcde\text{”}, 2)$. This answer set can be partitioned into five non-overlapping subsets consisting of strings of length from 3 to 7. Answer strings of length 3 are all contained in $Ans(\text{“}abcde\text{”}, 2D)$. This being a pure case, can be handled directly by the formula given in Section 3.5. Similarly, answers of length 7 are all contained in $Ans(\text{“}abcde\text{”}, 2I)$. This is also a pure case and a formula for determining its size will be given later in this section.

Line (7) of Figure 4 deals with all the general cases when the formulas in the previous section cannot be used directly. For instance, for $Ans(\text{“}abcde\text{”}, 2)$, answer strings of length 5 are contained in $Ans(\text{“}abcde\text{”}, 2R) \cup Ans(\text{“}abcde\text{”}, 1D1I)$. While we have a formula for $Ans(\text{“}abcde\text{”}, 2R)$, we do not have a formula for $Ans(\text{“}abcde\text{”}, 1D1I)$. Even if we had both formulas, estimating the sizes of the two sets separately and adding the two estimates would give a large error because the two answer sets overlap significantly. Thus, we have to resort to a *combined* treatment of the two sets, i.e., operating from a single combined string hierarchy. The set of base strings is first computed based on either two replacements, or one deletion and one insertion. During the computation, all redundant base strings are removed. For example, $Ans(\text{“}abcde\text{”}, 2R)$ produces a base string "abc??" while $Ans(\text{“}abcde\text{”}, 1D1I)$ generates a base string "abcd?". However, "abc??" contains all strings represented by "abcd?". Thus, we delete "abcd?" in the base strings for the combined string hierarchy. From then on, the procedure *BasicEQ* generates the string hierarchy and compute the estimate.

Procedure BasicEQ: A skeleton of Procedure *BasicEQ* is presented in Figure 5. Given a query s_q and edit distance threshold k, it gives a frequency estimate of all the answer strings of a specific length *len*. Line (2) of the procedure generates all the base strings for *len*. The set $baseNodes$ is the set of base strings that remain after removing redundant base strings as above. Then the bulk of the computation is to generate the nodes of the string hierarchy.

A naive approach is to perform this generation by considering all r-intersections $(2 \le r \le |baseNodes|)$ of the base strings in $baseNodes$. Obviously, due to the exponential nature, this is impractical. Instead, the procedure generates the r-intersections in a level-wise fashion. The algorithmic structure follows the well-known Apriori algorithm in [2].

Input: query s_q of length l, edit distance threshold k
length *len* of answer strings
Output: estimated frequency c

1: find all combinations (i, j, m) for $Ans(s_q, iDjImR)$ such that $i + j + m = k$ and $l - i + j = len$
2: generate the set $baseNodes$ of *non-redundant* base strings for the combinations above
3: initialize $newNodes$ as $baseNodes$, $totalNodes$ as ϕ, c as 0
4: **repeat**
5: $tmpNodes = \phi$
6: **for all** $u \in newNodes$ and $b \in baseNodes$ **do**
7: **if** $u \cap b \neq \emptyset$ **then**
8: $tmpNodes = tmpNodes \cup \{u \cap b\}$
9: **end if**
10: **end for**
11: $newNodes = tmpNodes - totalNodes$
12: $c = c + PartitionEstimate(newNodes)$
13: $totalNodes = totalNodes \cup newNodes$
14: **until** $newNodes$ does not contain any u with a wildcard
15: **return** c

Figure 5: A Skeleton for Procedure BasicEQ

The procedure first generates 2-intersections, such as $u \cap v$, $v \cap w$ and $u \cap w$ and so on. An r-intersection is non-empty only if all of its $(r-1)$-intersections among base strings appearing in the r-intersection are non-empty. This explains why it is sufficient to consider only new nodes in line (6).

However, the apriori strategy alone is not sufficient, and the following optimization is critical. *In order for a new intersection $(u \cap b)$ to be non-empty, u must contain at least one wildcard.* If there is no wildcard in u, an additional intersection to it is either itself or the empty string. As each iteration of the repeat-until loop of the procedure decreases the number of wildcards by at least one and the edit distance threshold k is not large, the procedure halts with a small number of iterations.

4.2 Node Partitioning

Line (12) of *BasicEQ* attempts to estimate the frequency of every newly generated node in the hierarchy. Recall from the previous section that the frequency contribution of a node q in the hierarchy is a combination of $C_q * |q|$, where *coefficient* C_q denotes the number of times q appears in all the intersections of base strings, and $|q|$ denotes the frequency of string q occurring in the database DB. For instance, recall from the example $Ans(\text{“}abcd\text{”}, 2R)$ in Section 3.3 that if q is the node "abcd", C_q is (-3+16-15+6-1) = 3. The same example also illustrates that many nodes q have the same coefficient C_q. For the example $Ans(\text{“}abcd\text{”}, 2R)$, all the nodes in the same level of the semi-lattice have exactly the same coefficient (e.g., the coefficient for F_1 is -3+1 = -2).

The same principle applies to a general string hierarchy. The goal of *PartitionEstimate* is to group the nodes of the hierarchy into partitions, so that *every node q in a partition has the same coefficient C_q*. In that case, it is sufficient to compute the coefficient of *all* the nodes in the partition only once. Below we give a sufficient condition for partitioning.

Given a string hierarchy H and a node q in H, the *local semi-lattice* of q is defined to be the sub-hierarchy of H that includes only nodes that are ancestors of q and q itself. By definition, q is the bottom element, or the *root*, of the

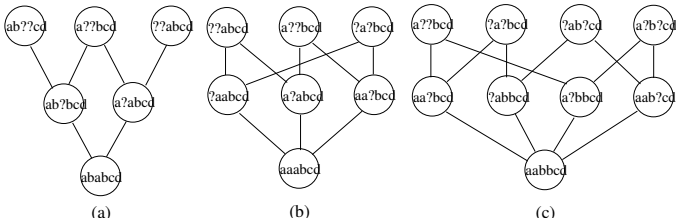


Figure 6: Examples of Local Semi-lattice

semi-lattice. For instance, if H is the semi-lattice shown in Figure 1, q_1 is the node "abcd" and q_2 is "abc?", then the local semi-lattice of q_1 is the entire semi-lattice H, and the local semi-lattice of q_2 is the sub-hierarchy with "abc?" as the bottom node and the 3 parents being "ab??", "a?c?" and "?bc?". We have the following proposition stating that local semi-lattices can form the basis of node partitioning.

PROPOSITION 4.1. *For nodes q_1, q_2 in string hierarchy H, if the local semi-lattices rooted at q_1, q_2 are isomorphic, then $C_{q_1} = C_{q_2}$.*

The above proposition can be proved by induction on the depth of the nodes in the semi-lattices. The converse of the proposition is not true, namely $C_{q_1} = C_{q_2}$ does not imply isomorphic local semi-lattices.

4.3 An Example and a Formula for Two Insertions

Let us consider the example of $Ans(\text{"}abcd\text{"}, 2I)$. Recall that Figure 3(c) shows the string hierarchy. As examples of local semi-lattices, Figure 6 shows the local semi-lattices of "ababcd", "aaabcd" and "aabbcd". To illustrate how *BasicEQ* operates, recall that every 2-intersection node is generated in its first iteration. Thus, all the level-0 and level-1 nodes in Figure 6(a) are produced in the first iteration (e.g. "ababcd"="aa??cd" ∩ "??abcd"). However, "aaabcd" in Figure 6(b) is generated in the second iteration because no 2-intersection can generate that node.

To illustrate node partitioning, all the level-0 nodes of the hierarchy shown in Figure 3(c) can be grouped into the three partitions as shown in Figure 6. The first partition, shown in Figure 6(a), consists of the nodes with a local semi-lattice isomorphic to that of "ababcd". (Below we refer to this partition as $P_{0,1}$.) Every node in this partition appears once in 2-intersections and once in 3-intersections of base strings with alternating sign, giving rise to the coefficient of -1+1=0. For example, among all possible 2-intersections of base strings in Figure 3(c), only "ab??cd" ∩ "??abcd" results in "ababcd".

The second partition illustrated in Figure 6(b) consists of the nodes with a local semi-lattice isomorphic to that of "aaabcd". (Below we refer to this partition as $P_{0,2}$.) Every node q in this partition does not appear in its local semi-lattice as a 2-intersection, but appears once as a 3-intersection, giving a coefficient C_q of 1.

Finally, the third partition presented in Figure 6(c) consists of nodes with a local semi-lattice isomorphic to that of "aabbcd" (Below we refer to this partition as $P_{0,3}$.). With a similar argument, the coefficient for each node in this partition becomes $-2 + 4 - 1 = 1$.

It is not hard to verify that $|Ans(\text{"}abcd\text{"}, 2I)|$ is given by $F_2 - F_1 + 0 * F_{0,1} + 1 * F_{0,2} + 1 * F_{0,3} = F_2 - F_1 + F_{0,2} + F_{0,3}$,

Input: $newNodes$, a set of newly formed nodes
Output: estimated frequency c

1: **for all** node $q \in newNodes$ **do**
2: form the set Par_q of all the parents of q
3: form the multiset PID_q of the partition id of parents in Par_q
4: form the set B_q of all the base strings which are ancestors of q
5: **end for**
6: given $newNodes = \{q_1, \ldots, q_t\}$ for some $t > 0$ put $q_i, q_j (1 \le i, j \le t)$ in one partition if the multisets PID_{q_i} and PID_{q_j} are identical and $|B_{q_i}| = |B_{q_j}|$
7: update the global partition table and c = 0
8: **for all** partition P formed in the previous step **do**
9: $C_P = ComputeCoefficient(P)$
10: $c = c + C_P * \sum_{q \in P} EstimateFreq(q)$
11: **end for**
12: **return** c

Figure 7: A Skeleton of Procedure PartitionEstimate

where F_i give the sum of frequencies of all the nodes in level-i for $i = 1, 2$, and $F_{0,1}, F_{0,2}, F_{0,3}$ denote the sums of the frequencies in partitions $P_{0,1}, P_{0,2}$ and $P_{0,3}$. This result in fact generalizes to arbitrary length s_q as the following:

PROPOSITION 4.2. *The cardinality of $Ans(s_q, 2I)$ is given by $F_2 - F_1 + F_{0,2} + F_{0,3}$.*

4.4 Procedure PartitionEstimate

In order to form partitions to use Proposition 4.1, an isomorphic test on the structure of the local semi-lattices is required. A brute-force approach is computationally expensive. Fortunately, in our framework, this test can be integrated into the level-wise computation of *BasicEQ*. As newly created nodes are computed in each iteration of *BasicEQ*, all these nodes are passed to *PartitionEstimate*, of which a skeleton is presented in Figure 7. Each node generated by *BasicEQ* is assigned a partition id. For bookkeeping purposes, there is a partition table which maps a node to its partition id. There is a second table that maps a partition id to the set of nodes contained in the partition. In our experimentation, there are typically fewer than 40 partitions; thus, lookups can be done very quickly.

When *PartitionEstimate* starts, all the base strings are in one partition because each local semi-lattice consists of just itself. Then inductively, to check whether the local semi-lattices of two nodes q_1 q_2 have the same structure, it checks whether the partition ids of these two *multisets* of parents and the number of base strings that are ancestors are identical. These are necessary but not sufficient conditions for isomorphism of two lattices. In Section 6, the empirical results will show that the two conditions are rather effective heuristics.

It is possible that both a node and its parents are generated, put into $newNodes$, and fed as input in the same invocation of $PartitionEstimate$. However, as ancestors cannot be generated later than their descendants, level-wise processing in the first for-loop in Figure 7 insures that the PID of parents are available when it processes a node. This detail is omitted in Figure 7.

Input: a partition P
Output: estimated frequency C_P

```
1: take any node q in partition P and set compBase consisting of all the base strings which are ancestors of q
2: initialize rInter = {{b}|b ∈ compBase}; r = 2; C_P = 0
3: while rInter ≠ φ and r ≤ |compBase| do
4:    tmpInter = φ, C'_P = 0
5:    for all base ∈ compBase and inter ∈ rInter and base ∉ inter do
6:       tmpInter = tmpInter ∪ {inter ∪ {base}}
7:       if (base ⋂ intersect(inter)) = q then
8:          C'_P = C'_P + 1
9:       end if
10:   end for
11:   C_P = C_P + (−1)^{r+1} C'_P
12:   rInter = tmpInter; r = r + 1
13: end while
14: return C_P
```

Figure 8: A Skeleton of Procedure ComputeCoefficient

Once the partitions are formed, then *PartitionEstimate* invokes *ComputeCoefficient* in line (9) to compute the coefficient C_p for each newly formed partition P, a procedure to be detailed in the next subsection. Once C_P is computed, line (10) multiplies this coefficient to the frequency of each node in the partition P. While every node in a partition P has the same coefficient C_P, each may have its own frequency of occurrence in the database DB. Estimating these frequencies will be discussed at the end of this section.

4.5 Procedure ComputeCoefficient

Given a partition P of nodes computed by *PartitionEstimate*, the procedure *ComputeCoefficient* shown in Figure 8 computes the coefficient C_P, the number of times a node $q \in P$ appears in all the r-intersections of the base strings. Essentially, it starts with the set *compBase* of base strings that are ancestors of q in the hierarchy. Then the algorithm iterates to find r-intersections, starting from $r = 2$. The set $rInter$ consists of all non-empty r-intersections. In other words, a set *inter* in $rInter$ represents a particular non-empty r-intersection. The for-loop starting in line (5) constructs $(r + 1)$-intersections by testing one base string *base* at a time from *compBase*. To perform the intersection step in line (7), $intersect(inter)$ computes the intersection of all elements in *inter*. However, we can store the intersection in previous step and just return it here. If this new intersection is equivalent to q itself, the coefficient C_q is incremented.

Let us illustrate how *ComputeCoefficient* works. Assume that a partition P with a node of "ababcd" in Figure 6(a) and q is selected as "ababcd" at the beginning of the execution. Then, we get *compBase* = {"ab??cd", "a??bcd", "??abcd"}. At the first iteration of the while-loop in line (3), *tmpIter* becomes { {"ab??cd", "a??bcd"}, {"ab??cd", "??abcd"}, {"a??bcd", "??abcd"'}} after exiting the for-loop. Furthermore, the if-statement in line (7) finds that only "ab??cd"' ∩ "??abcd" results in q. Thus, C'_P becomes 1 and C_P is -1. Since we set *rIter* to *tmpIter* and *rIter* is not empty, we perform the next iteration of the while-loop. Now, there is only one intersection {"ab??cd", "a??bcd", "??abcd"} that is the only element in *tmpIter* and the result of the intersection is q. Thus, C'_P becomes 1 which is added to C_P by line (11) and C_P becomes zero. The value of C_P does not be changed later on and finally C_P becomes zero.

One reason why this algorithm is slow is that we need to maintain every intersection generated even though the current r-intersection does not result in q. To alleviate this problem, we develop an approximation method later that avoids the maintenance of all r-intersections generated.

4.6 Procedure EstimateFreq

For the material presented so far, the discussion is mainly based on strings (e.g., nodes in string hierarchy). It is only in line (10) of *PartitionEstimate* that requires the use of the N-gram table. The task is that for a given string q, the frequency that this string appears in the database DB is returned. For an extended string q, if the extended N-gram table kept by the system contains an entry for q (e.g., when $|q| \leq N$), then the frequency $|q|$ is immediately returned. Otherwise, $|q|$ needs to be estimated using estimation algorithm as MO [14]. The experimental evaluation section will compare the accuracy when *EstimateFreq* implements MO, as well as other variants below.

- If s is obtained from t by replacing at least one character in t with ?, then by definition $|s| \geq |t|$ (e.g. $|$"*abc*?"$| \geq |$"*abcd*"$|$). However, MO estimation may violate this condition, i.e., $MO(s)$ may be smaller than $MO(t)$. In that case, the MO estimate of s is reset to the maximum MO estimate of all such t's. We call this the MAX estimate.
- Among all the substrings q' of q that are kept in the N-gram table, let q'_{min} be the substring with the smallest frequency. By definition, $|q| \leq |q'_{min}|$. In other words, this is an over-estimate of q. It has been well documented that the MO-estimate of q is often an underestimate, particularly when q is long [6]. Thus, one estimate is to take the geometric mean of $|q'_{min}|$ and the MO-estimate of q. We call this MO+ estimate.
- We can combine the MAX estimate and the MO+ estimate to give another estimate referred to as MM. That is, it gives the geometric mean of $|q'_{min}|$ and the MAX estimate of q.

Section 6 will compare empirically the effectiveness of these instances of *EstimateFreq*.

5. OPTIMIZED ALGORITHM OPTEQ

BasicEQ is efficient whenever the general formulas presented in Section 3 can be applied. For other situations, *BasicEQ* is sufficient for small problems. However, it is clear that it does not scale up with respect to query length l and Δ when the formulas cannot be used. The complexity of *BasicEQ* is exponential with respect to l. In this section, we propose the optimized algorithm *OptEQ* which extends *BasicEQ* with two enhancements. We show experimentally in the next section how *OptEQ* can handle long queries efficiently.

5.1 Approximating Coefficients with a Replacement Semi-lattice

When the set *compBase* is large (e.g., query s_q is long or k is large), computing the exact value of C_q as in *ComputeCoefficient* is prohibitive. We next present a strategy to approximate C_q by avoiding the generation of all r-intersections.

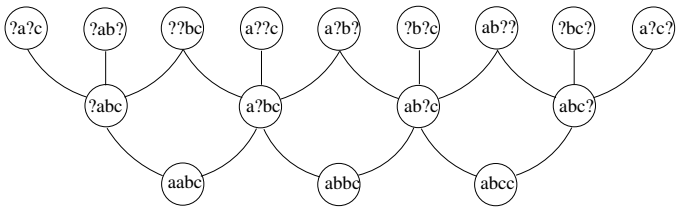


Figure 9: String Hierarchy of $Ans(\text{“}abc\text{”}, 1I1R)$

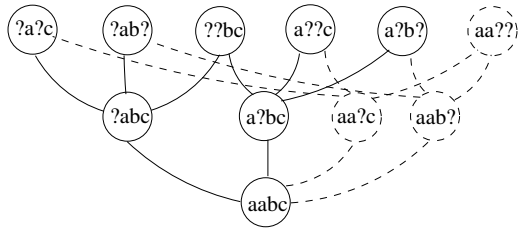


Figure 10: Completion of a Replacement Semi-lattice of $Ans(\text{“}aabc\text{”}, 2R)$

Let us revisit the example of $Ans(\text{“}abcde\text{”}, 2)$. As the computation iterates over the length of the answer strings, one iteration deals with the strings of length five contained in $Ans(\text{“}abcde\text{”}, 1D1I) \cup Ans(\text{“}abcde\text{”}, 2R)$. The next iteration deals with those of length six in $Ans(\text{“}abcde\text{”}, 1I1R)$. The string hierarchies in both cases are rather complex and thus the approximation strategy is applied.

For ease of presentation, we use $Ans(\text{“}abc\text{”}, 1I1R)$ as an example to illustrate the strategy. To generate all the base strings, we can first consider the position of one insertion, followed by one replacement. Specifically, we can group the base strings into four and the 4 groups are as follows:

- (inserting into first position) “??bc”, “?a?c” and “?ab?”
- (inserting into second position) “??bc”, “a??c” and “a?b?”
- (inserting into third position) “?b?c”, “a??c” and “ab??”
- (inserting into fourth position) “?bc?”, “a?c?” and “ab??”

Note that the strings in DB belonging to “?abc” in the first group above is also included in other base strings. Thus, “?abc” is pruned in the first group. Similarly, “a?bc”, “ab?c” and “abc?” are removed from their group’s base strings.

With the base strings of $Ans(\text{“}abc\text{”}, 1I1R)$, we can build the string hierarchy presented in Figure 9. In the string hierarchy, the three base strings in the first group are organized in a sub-semi-lattice rooted at “?abc”. Similarly, the three base strings in the second group form a sub-semi-lattice rooted at “a?bc”. Note that “??bc” is shared between two groups and thus these two groups correspond to a general situation when two semi-lattices *overlap* on some base strings. To compute the size of $Ans(\text{“}abc\text{”}, 1I1R)$, eventually the coefficient of every node in the string hierarchy in Figure 9 is needed.

Assume that we want to compute the coefficient of “aabc” in Figure 9. Instead of applying *ComputeCoefficient* directly on “aabc”, the approximation strategy *completes the semi-lattice* of “aabc” with two replacements. This completed replacement semi-lattice is shown in Figure 10 where the two sub-semi-lattices rooted at “?abc” and “a?bc” are shown in solid lines, while the other parts, which do not appear in the string hierarchy of Figure 9 and thus are not required for

	2-inter.	3-inter.	4-inter.	5-inter.	6-inter.
C_r for bottom node	-3	16	-15	6	-1
estimated C_r for “aabc”	$\binom{5}{2}/\binom{6}{2}*-3=-2$	$\binom{5}{3}/\binom{6}{3}*16=8$	$\binom{5}{4}/\binom{6}{4}*-15=-5$	$\binom{5}{5}/\binom{6}{5}*6=1$	$\binom{5}{6}=0$
exact C_r for “aabc”	-2	8	-5	1	0

Figure 11: Approximating Coefficients for “aabc”

$Ans(\text{“}abc\text{”}, 1I1R)$, are shown in dotted lines. The beauty of the replacement semi-lattice rooted at “aabc” is that Eqn. (3) immediately gives a formula for C_{aabc}, *if the entire replacement semi-lattice appears (i.e. are involved)* in the sub-semi-lattice rooted at “aabc” in Figure 9. Specifically, recall from the discussion in Section 3.3 that the root of the semi-lattice is the level-0 node. Thus, the relevant coefficient is the one for F_0, which is:

$$C_{root} = \sum_{r=2}^{\binom{\ell}{k}} (-1)^{r+1} D_{\ell,k,r}.$$

When parts of the replacement semi-lattice are not involved, the approximation strategy *assumes that the number of r-intersections leading to a root node is proportional to the number of its base strings that are involved.* This assumption arises from the highly uniform structure of a replacement semi-lattice. Thus, each term in the summation is scaled relative to the number of the base strings involved. Specifically, let B the size of *compBase*, which is the set of base strings that are involved. Then, C_{root} is estimated by:

$$C'_{root} = \sum_{r=2}^{\binom{\ell}{k}} (-1)^{r+1} D_{\ell,k,r} * \frac{\binom{|B|}{r}}{\binom{\binom{\ell}{k}}{r}} \quad (4)$$

That is, in a completed replacement semi-lattice, there are $\binom{\ell}{k}$ base strings. But as there are only B base strings in the completed replacement semi-lattice that are involved, every term is scaled proportionally.

Let us return to the example of “aabc”. Recall from Figure 10 that there are five base strings being involved for the query of $Ans(\text{“}abc\text{”}, 1I1R)$, as supposed to six (i.e., “aa??” being the only base string not involved) in the completed replacement semi-lattice. Thus, in this example, $|B| = 5$ and $\binom{\ell}{k} = \binom{4}{2} = 6$. The table shown in Figure 11 applies the approximation strategy to C_{aabc}. The second row of the table is identical (modulo the sign) to the third row of the table in Figure 2, as it gives the coefficients C_r for the root node. The third row applies the proportionality factor to each term. The last row of the table shows the true coefficients, which turn out to be identical to the approximated ones! In general this is not always true. For example, the division in Eqn. (4) does not always give an integer value.

5.2 Fast Intersection Tests by Grouping

Another way to optimize *BasicEQ* is to optimize the for-loop in line (6) of Figure 5. Specifically, for every possible pair of a base string $b \in baseNode$ and $u \in newNodes$, we perform the intersection. However, as seen in the previous example shown in Figure 10, base strings may naturally group into a collection of semi-lattices. In the case of approximating by completion of a replacement semi-lattice, the individual semi-lattices are all parts of a larger, completed semi-lattice. In a more general setting, this condition may not hold for all the semi-lattices. Nevertheless, some of the semi-lattices may overlap. Below we explore further the notion of grouping and show how group-wise operations can be exploited.

Input: query s_q of length ℓ, edit distance threshold k
length *len* of answer strings
Output: estimated frequency c

1: Identical to procedure *BasicEQ* except using grouping in line 4.1 and invoking *ComputeCoefficient*(q) by PartitionEstimate:
2: **if** q is the root of a replacement semi-lattice **then**
3: apply the formula in Eqn. (3)
4: **else if** *compBase* of q exceeds a certain size **then**
5: apply the approximation strategy based on Eqn. (4)
6: **else if** *ComputeCoefficient* is to be applied **then**
7: use grouping to speed up and computation of coefficients as discussed in Section 5.2
8: **end if**

Figure 12: A Skeleton of Procedure OptEQ

Note that a replacement operation does not cause any character shifting in the result, while an insertion or a deletion does. Thus, we organize all the base strings for a query of the form $Ans(s_q, jIjD1R)$ by considering the position of insertion and deletions only.

To illustrate the power of grouping, let us consider the example of $Ans(\text{“}abcd\text{”}, 1D1I1R)$, as part of a query with the edit distance $k = 3$. We group all the base strings for $Ans(\text{“}abcd\text{”}, 1D1I1R)$ by first considering the deletion and insertion. The following array shows all the possible combinations for one deletion and one insertion in any order.

$$G = \begin{pmatrix} \text{“?}abc\text{”} & \text{“}a\text{?}bc\text{”} & \text{“}ab\text{?}c\text{”} & \text{“}abc\text{?”} \\ \text{“?}abd\text{”} & \text{“}a\text{?}bd\text{”} & \text{“}ab\text{?}d\text{”} & \text{“}abd\text{?”} \\ \text{“?}acd\text{”} & \text{“}a\text{?}cd\text{”} & \text{“}ac\text{?}d\text{”} & \text{“}acd\text{?”} \\ \text{“?}bcd\text{”} & \text{“}b\text{?}cd\text{”} & \text{“}bc\text{?}d\text{”} & \text{“}bcd\text{?”} \end{pmatrix}$$

We can view the first row as deleting the character d and inserting a character at various positions. Similarly, we can view the first column as inserting a character at the first position and deleting a character at various positions. Then the replacement can be applied to *each* of the 16 elements of matrix G. For instance, applying one replacement to “?abc” gives “??bc”, “?a?c” and “?ab?”. *Note that these three strings form a replacement semi-lattice rooted at “?abc”.* Indeed, the same phenomenon applies to every element of G. In other words, the base strings of $Ans(\text{“}abcd\text{”}, 1D1I1R)$ are split into sixteen groups, or more specifically, semi-lattices with the roots being the elements of G.

The first benefit of grouping in this manner can be exemplified by noting that $G(1,1) =$ “?abc” and $G(4,1) =$ “?bcd” generate empty intersection. More importantly, any one of the base strings within the group of “?abc” (namely, “??bc”, “?a?c” and “?ab?”) and any one of the base strings within the group of “?bcd” (namely, “??cd”, “?b?d?” and “?bc?”) always produce an empty intersection. This is because regardless of which pair, there is at least one position where there are no wildcards and the characters do not match. This motivates the following proposition.

PROPOSITION 5.1. *Let g_1, g_2 be two elements in matrix G. Let $grpDist(g_1, g_2)$ be defined as the number of non-matching positions between g_1, g_2. Let s_1 be any node within the r_1-replacement semi-lattice rooted at g_1, where r_1 is defined by the query. Similarly, let s_2 be any node within the r_2-replacement semi-lattice rooted at g_2. Then $grpDist(g_1, g_2) > r_1 + r_2$ implies $s_1 \cap s_2 = \phi$.*

For the running example of $Ans(\text{“}abcd\text{”}, 1D1I1R)$, if g_1, g_2 are “?abc” and “?bcd” respectively, then $r_1 = r_2 = 1$ because of the $1R$ specification in the query. However, the group distance $grpDist(g_1, g_2) = 3$ because there are 3 non-matching positions. The proposition then allows us to immediately conclude that for *any* pair of s_1 from the 1-replacement semi-lattice rooted at g_1 and s_2 from the 1-replacement semi-lattice rooted at g_2, their intersection must be empty. Thus, grouping and group distance defined via the roots of two groups provide a fast negative test for groups of base strings. This speeds up line (6) of *BasicEQ*.

Another benefit of grouping is that each replacement semi-lattice allows the coefficient to be computed directly by the formula given in Eqn. (3), thus avoiding the use of *ComputeCoefficient*. Specifically, the coefficient of the root of a group is precisely $C_{root} = \sum_{r=2}^{\binom{\ell}{k}} (-1)^{r+1} D_{\ell,k,r}$ and the coefficient of an intermediate level-i node in the group is $C_q = \sum_{r=2}^{\binom{\ell}{k}} (-1)^{r+1} D_{\ell-i,k-i,r}$, which is the coefficient of F_i in Eqn. (3). However, we have to be careful in defining ℓ and k before the above formulas can be used. In the “?abc” group for our example, every node in the replacement semi-lattice has a wildcard at (at least) the first position. This position should not be considered in the replacement semi-lattice. *Effective string length* is defined as the length of the root after excluding any wildcard in the root which is shared in the group. Similarly, *effective number of wildcards* is the number of wildcards excluding the wildcards common to all within the group. In applying the formulas above, ℓ and k are set to the effective string length and effective number of wildcards. For example, in Figure 9, the sub-semi-lattice rooted at “?abc” has 3 base strings. Its effective string length is 3 instead of 4 and its effective number wildcards is 1 instead of 2. Thus, we compute the coefficient of “?abc” using Eqn. (3) by setting $\ell = 3$ and $k = 1$

Procedure *OptEQ* is an optimized version of *BasicEQ* that incorporates approximation and grouping into the size estimation. The skeleton shown in Figure 12 highlights the key differences. *OptEQ* is scalable to deal with size estimation of queries like $Ans(\text{“}abcd\text{”}, 1D1I1R)$ and more complex combinations. In general, as query s_q becomes longer, the G matrix shown earlier becomes larger. However, there are more and more groups with large group distance and fast negative tests provide scalability. Similarly, for the general situation of $Ans(s_q, iDjImR)$, if $(i+j)$ is large, again the G matrix becomes larger. Yet, like before, grouping helps significantly. On the other hand, if m is large (while k remains constant), the replacement semi-lattice becomes dominant. In that case, *OptEQ* either applies Eqn. (3) if possible, or Eqn. (4) if approximation is needed.

6. EMPIRICAL EVALUATION

6.1 Implementation Highlights

The proposed algorithms *BasicEQ* and *OptEQ* were developed using Java 1.5. These algorithms were applied to different settings of N-gram tables parameterized by the triplet (N_B, N_E, PT):

- all q-grams *without wildcards* for $|q| \leq N_B$ are kept;
- all q-grams *with wildcards* for $|q| \leq N_E$ are kept; but
- all the entries kept must have a count $>$ the prune threshold PT.

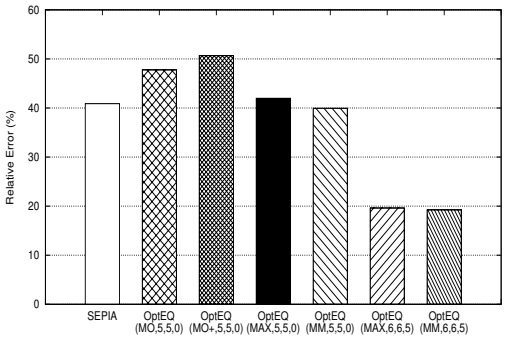


Figure 13: Actress Last Name, $\Delta = [1, 3]$

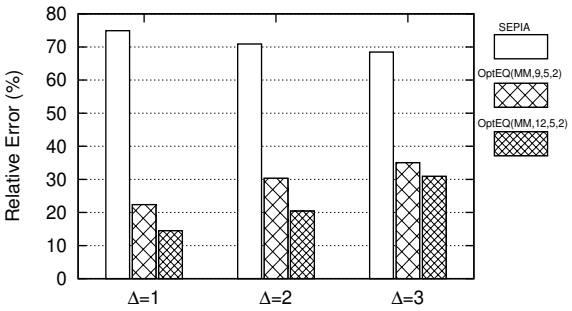


Figure 14: DBLP Authors, $\Delta = 1, 2, 3$

$OptEQ$(method,N_B,N_E, PT) denotes the variants when the method used for $EstimateFreq$ was one of: MO, MO+, MAX and MM, and using the N_B-gram table and the extended N_E-gram table with prune threshold PT. For instance, $OptEQ$(MM,9,6,1) gives the version of $OptEQ$ using MM for frequency estimation on the 9-gram table and the extended 6-gram table, where only entries of count ≥ 2 are kept. Below we will evaluate the tradeoffs among N_B, N_E and PT, with respect to accuracy and memory size.

In our implementation, the (extended) q-gram tables are optimized in size in two ways. First, in hash buckets of tables, the string corresponding to a q-gram entry is hashed again and stored as a byte key not as a full string key. Thus, there may be collisions introduced by hashing, representing a tradeoff between accuracy and size. The results reported below include the errors arisen due to collisions. Second, for each entry, the count is initially restricted to a short integer. When the count exceeds the range of a short integer, the count is maintained separately in an overflow table.

The discussion so far on the computation of coefficients shows how they can be computed - for once. In our implementation, these coefficients are pre-computed and stored in a coefficient table. This is possible because the coefficients depend only on the length and the Δ value of the query. In "query time", once the actual frequencies of the required nodes are determined, the selectivity can be estimated very efficiently using the pre-computed coefficients. Note that this table remains unchanged from one data set to another.

6.2 Experimental Setup

We conducted a series of experiments using several data sets. The results shown here are based on three benchmarks.

- (Short strings) The Actresses last name data set consists 392,132 last names of actresses downloaded from www.imdb.com. The maximum and average length are 16 and 6.3.
- (Medium-length strings) The DBLP authors data set consists of 699,198 authors' full names from DBLP [18]. The maximum and average length are 43 and 14.3.
- (Long strings) The DBLP titles data set consists of 53,365 titles collected from DBLP. The maximum and average length are 40 and 29.6.

$SEPIA$ was downloaded from the FLAMINGO project homepage [10] and was written in C++. We tuned $SEPIA$ to maximize its performance. The results reported here are based on applying the error correction step included in the software, and on using 2,000 clusters, a sampling ratio of 5% and the CLOSE_RAND method to populate the PPD table.

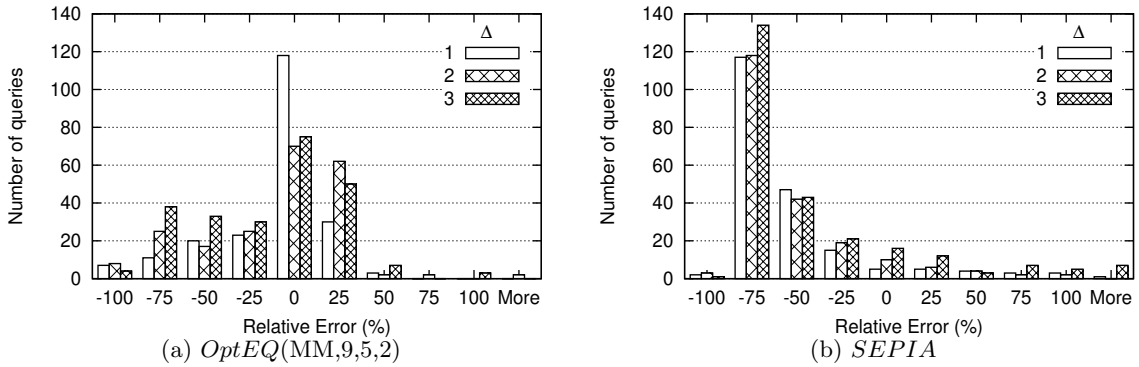
Accuracy is measured by *average relative error*, defined as $|est - true|/true$, where est and $true$ are the estimated and true frequencies respectively. To avoid very low frequency queries from skewing the average relative error figure, we exclude queries whose true frequencies are below 3. We also exclude best and worst 3 queries from the analysis. The experiments were run on an Intel P4 3GHz desktop PC with 1 GB memory and 40GB disk space running GNU/Linux with kernel 2.6. For the memory size figures cited below, the figure for $SEPIA$ is based on the file size written on disk which includes the PPD table and the frequency table. For $OptEQ$, the size figure is based on multiplying the number of entries in the q-gram and extended q-gram tables with the number of bytes per entry.

6.3 Actresses Last Names

Figure 13 compares the average relative error between $SEPIA$ and variants of $OptEQ$ for the Actresses data set. A total of 300 queries are randomly selected, of which 272 have true frequencies exceeding 3. The average relative error is obtained based on the 272 queries. The edit distance threshold Δ is selected among 1, 2 and 3 with equal likelihood. The first column shows that the average relative error for $SEPIA$ is around 40%. The next four columns show the version of $OptEQ$ when the different frequency estimation methods are used, $N_B = N_E = 5$, and $PT = 0$. Compared with MO and MO+, MAX and MM were consistently superior. In the experiments to follow, we only show the results of MM.

When $OptEQ(MAX, 6, 6, 5)$ or $OptEQ(MM, 6, 6, 5)$ are used, the last two columns show that the average relative errors are reduced in half from 40% to about 20%. This shows the effectiveness of increasing N_B and N_E in reducing relative error. The reason for setting the prune threshold of PT to 5 is to keep the memory utilization of $OptEQ$ to be more or less the same as that used by $SEPIA$ (3.3MB vs 3.6MB). Thus, using less memory, $OptEQ(MM, 6, 6, 5)$ reduced the average relative error by half.

In terms of running time, the "build" time to construct the extended N-gram tables took about 5 minutes. The average query processing time for $OptEQ$ was about 13 msec using the pre-computed coefficient table (or 1.2 seconds if $OptEQ$ were to compute every coefficient on-the-fly). The build time of $SEPIA$ to construct the clusters and the histograms took around 60 minutes. The query processing time for $SEPIA$ was about 8 msec. We do not include the results for $BasicEQ$ because the execution time was very slow when the query exceeded a length of 10. For instance, for a query of length 11, $BasicEQ$ took 74 seconds.



(a) $OptEQ$(MM,9,5,2) (b) $SEPIA$

Figure 15: Error Distribution: $OptEQ(MM, 9, 5, 2)$ vs $SEPIA$

If we only keep simple q-grams, not extended q-grams, the viable approach would be to enumerate all possible strings within the threshold and sum up their frequencies. We do not present detailed results, but the performance was unacceptable. For example, even for the simple query s_q = "*blank*", there are more than 4 million possible strings within edit distance threshold of 3. It took more than 20 seconds just to estimate the frequency of each string and sum them up. It took only 7 msec for $OptEQ$ to process the same query which is three orders of magnitude faster. This clearly shows the utility of the proposed extended q-grams.

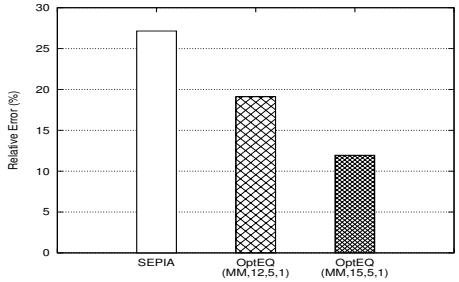


Figure 16: DBLP Titles, Δ in [1, 3]

6.4 DBLP Authors

This set of experiments used queries of an average length more than double than those used in the Actresses data set. Out of the 900 randomly selected queries, 674 have true frequencies exceeding 3, and are more or less equally distributed among $\Delta = 1, 2, 3$. Figure 14 gives the average relative errors, with Δ separated into 1, 2 and 3 respectively. $SEPIA$ consistently hovers around 70% in relative error. The average relative error given by $OptEQ(MM, 12, 5, 2)$ increases from around 15% for $\Delta = 1$ to about 30% for $\Delta = 3$. The memory used by $OptEQ(MM, 12, 5, 2)$ and $SEPIA$ were 13.7MB and 14MB respectively. Again, with similar or less memory, $OptEQ$ delivered significantly more accurate estimations. Furthermore, we reduce the memory consumption by using $OptEQ(MM, 9, 5, 2)$, which occupies only 9.2MB. Yet the average relative error is still significantly lower than that of $SEPIA$. Note that we increased N_B not N_E from the Actresses data set. N_E= 5 or 6 generally give good accuracy with reasonable space. However, increasing N_E beyond 6 is not recommended as it incurs combinatorial increase in the space. N_B offers a tunable alternative, and we used higher N_B for the authors data set as strings are generally longer.

While a single value of average relative error could be misleading, Figure 15 shows the distribution of errors with respect to Δ. The x-axis divides the error range into ranges of 25% width, e.g., ranges [-100%,-75%), [-75%,-50%), etc. The y-axis shows the numbers of queries within the error ranges. For $\Delta = 1, 2, 3$, $OptEQ(MM, 9, 5, 2)$ gives a more balanced distribution. In contrast, $SEPIA$ suffers from rather serious under-estimation problem here.

6.5 DBLP Titles

Compared with the other two data sets, the DBLP titles data set contains the longest strings. Out of the 600 randomly generated queries, only 31 have true frequencies exceeding 3. Among them 16, 8 and 7 have $\Delta = 1, 2, 3$ respectively. In particular, for $\Delta = 3$, 5 out of 7 queries have a length 15 or higher. To reduce the processing time for $OptEQ$, we employed sampling and limited the maximum number of groups and base strings in each group to 15 and 10 respectively for any query of length ≥ 10 and $\Delta = 3$. Figure 16 gives the average relative errors. The average relative error of $SEPIA$ is about 27%, while occupying 13MB of memory. In contrast, $OptEQ(MM, 12, 5, 1)$, occupying a space of 4.3MB, gives an average relative error of about 20%. If more space is allowed, $OptEQ(MM, 15, 5, 1)$ with a space of 5.4MB, gives an average relative error of 12%. The average query time is 76 msec.

Although our main focus lies in low edit distance thresholds(i.e. $\Delta = 1, 2, 3$), we briefly present results on higher thresholds, Δ= 4 and 5. To handle higher thresholds, *sampling* approach is essential. We randomly sample 10 base strings for each string hierarchy. We exclude too short sampled queries (length ≤ 5) as they are meaningless in the increased thresholds. The average relative error given by $OptEQ(MM, 12, 5, 5)$ is around 60%. The error is increased significantly as all the base strings are not considered due to sampling. However, it is still better than the average relative error of SEPIA which was 150%. It occupied 1.5MB and the running time was 10 msec. SEPIA occupied 10MB and it took 30 msec to estimate.

The maximum edit distance threshold we support is limited by the maximum number of wildcard in a q-gram, which in turn is bounded by N_E. Based on the discussion in section 6.4 regarding N_E, we do not recommend $OptEQ$ for the thresholds of 6 or higher.

6.6 Space vs Accuracy

The three parameters N_B, N_E and PT provide a very tunable setting. As shown in Figure 16, an increase in N_B leads

to increase in space consumed but a decrease in relative error. The same can be said for N_E as shown in Figure 13. However, incrementing N_E by 1 requires a lot more space than incrementing N_B by 1. In our experimentation, we found that there is a significant reduction in error when N_E is raised from 4 to 5, whereas the reduction in error becomes much smaller when N_E exceeds 7.

When a large N_E is used, one way to keep memory consumption in check is to apply a larger prune threshold PT. In general, an increase in PT leads to a reduction in space but an increase in relative error. Consider $OptEQ(MM, 15, 5, 1)$ in Figure 16. The following table shows the impact of PT on accuracy and size.

	avg. rel. error	size (MB)
$OptEQ(MM, 15, 5, 0)$	11.8%	19.4
$OptEQ(MM, 15, 5, 1)$	12.0%	5.4
$OptEQ(MM, 15, 5, 4)$	14.7%	1.9

As compared with $SEPIA$ which uses 13MB of memory and gives an average relative error of 27%, the last two variants shown in the above table are better alternatives.

7. CONCLUSIONS

OptEQ uses the concept of extended q-grams, and exploits the analysis of a replacement semi-lattice. The formula for determining the number of answer strings in a replacement semi-lattice can be used to approximate coefficients of partitions. Moreover, *OptEQ* groups semi-lattices and applies group-wise operations to provide scalability. For all the three benchmarks, *OptEQ* can deliver selectivity estimations more accurate than $SEPIA$ does. *OptEQ* is capable of exploiting available disk space to give higher precision. Among the variants experimented with, *OptEQ* with the MM estimate or the MAX estimate appears to give good accuracy in most cases.

In ongoing work, we explore further the relationship between disk space utilization and estimation accuracy given by *OptEQ*. We also plan to extend the current framework to deal with higher edit distance threshold. While this extension may not be too essential for database applications, the extension is essential for applications such as DNA sequence matching. At the present time, when the edit distance threshold is larger than 5, $SEPIA$ can be more effective than *OptEQ*. Finally, as q-grams keeps frequencies of substrings, we plan to augment *OptEQ* to give selectivity estimation for substring matching with edit distance.

8. REFERENCES

[1] A. Aboulnaga, A. R. Almeldeen and J. F. Naughton. Estimating the selectivity of XML path expressions for Internet scale applications. *Proc. VLDB*, pp. 591-600, 2001.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *Proc. VLDB*, pp. 487–499, 1994.

[3] R. Baeza-Yates and B. A. Ribeiro-Neto. Modern Information Retrieval. *ACM Press/Addison-Wesley*, 1999.

[4] S. Burkhardt and J. Karkkianen. One-gapped q-Gram Filters for Levenshtein Distance, Springer, LNCS, pp. 225-234, 2002.

[5] S. Burkhardt et al. Q-gram based database searching using a suffix array (QUASAR), *Proc. RECOMB*, pp. 77-83, 1999.

[6] S. Chaudhuri, V. Ganti and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. *Proc. ICDE*, pp. 227- 238, 2004.

[7] S. Chaudhuri et al. Robust and Efficient Fuzzy Match for Online Data Cleaning. *Proc. SIGMOD*, pp. 313-324, 2003.

[8] W. Cohen, P. Ravikmar and and S. Fienberg. A Comparison of String Metrics for Matching Names and Records. *Proc. of KDD Workshop on Data Cleaning*, pp. 73-78, 2003.

[9] R. Dixon and T. Martin. Automatic Speech and Speaker Recognition. *IEEE Press*, 1979.

[10] Flamingo Project, http://www.ics.uci.edu/ flamingo/.

[11] L. Gravano et al. Approximate string joins in a database (almost) for free. *Proc. VLDB*, pp. 491-500, 2001.

[12] D. Gusfield. Algorithms on Strings, Trees and Sequences. *Cambridge Univ. Press*, 1997.

[13] B. Hore et al. Indexing Text Data under Space Constraints. *Proc. CIKM*, pp. 198–207, 2004.

[14] H. V. Jagadish, R. T. Ng and D. Srivastava. Substring Selectivity Estimation, *Proc. PODS*, pp. 249-260, 1999.

[15] L. Jin and C. Li. Selectivity Estimation for Fuzzy String Predicates in Large Data Sets , *Proc. VLDB*, pp. 397-408, 2005.

[16] L. Jin et al. Indexing Mixed Types for Approximate Retrieval, *Proc. VLDB*, pp. 793-804, 2005.

[17] P. Krishnan, J. S. Vitter and B. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. *Proc. SIGMOD*, pp. 282-293, 1996.

[18] M. Ley. *DBLP*, http://www.fnformatick.uni-tier.de/ ley/db.

[19] L. Lim et al. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. *Proc. VLDB*, pp. 442-453, 2002.

[20] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Survey*, 33, pp. 31-88, 2001.

[21] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *Journal of Experimental Algorithmics*, volume 5, article 4, 2000.

[22] S. Sarawagi and A. Bhamidipaty. Interactive Deduplication Using Active Learning. *Proc. VLDB*, pp. 269-278, 2002.

[23] Winkler, W. E. The State of Record Linkage and Current Research Problems. Statistics of Income Division, Internal Revenue Service Publication R99/04.