Cost-Based Variable-Length-Gram Selection for String Collections to Support Approximate Queries Efficiently

Xiaochun Yang
School of Information Science and Engineering
Northeastern University
Shenyang, China
yangxc@mail.neu.edu.cn

Bin Wang
School of Information Science and Engineering
Northeastern University
Shenyang, China
binwang@mail.neu.edu.cn

Chen Li
Department of Computer Science
University of California, Irvine
CA, USA
chenli@ics.uci.edu

ABSTRACT

Approximate queries on a collection of strings are important in many applications such as record linkage, spell checking, and Web search, where inconsistencies and errors exist in data as well as queries. Several existing algorithms use the concept of "grams," which are substrings of strings used as signatures for the strings to build index structures. A recently proposed technique, called VGRAM, improves the performance of these algorithms by using a carefully chosen dictionary of variable-length grams based on their frequencies in the string collection. Since an index structure using fixed-length grams can be viewed as a special case of VGRAM, a fundamental problem arises naturally: what is the relationship between the gram dictionary and the performance of queries? We study this problem in this paper. We propose a dynamic programming algorithm for computing a tight lower bound on the number of common grams shared by two similar strings in order to improve query performance. We analyze how a gram dictionary affects the index structure of the string collection and ultimately the performance of queries. We also propose an algorithm for automatically computing a dictionary of high-quality grams for a workload of queries. Our experiments on real data sets show the improvement on query performance achieved by these techniques. To our best knowledge, this study is the first cost-based quantitative approach to deciding good grams for approximate string queries.

Categories and Subject Descriptors

H.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Content Analysis and Indexing

General Terms

Algorithms, Performance

Keywords

Approximate string query, gram selection, VGRAM

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. INTRODUCTION

This paper studies the following problem: given a collection of textual strings, such as person names, paper titles, telephone numbers, and company addresses, how to efficiently find those that are similar to a given query string? The problem arises naturally in many applications. To name a few, in record linkage [14], we often need to find from a table those records that are similar to a given query string that could represent the same real-world entity, even though they have slightly different representations, such as `giuliani` versus `guliani`. In spellchecking, for each word that is not in a predefined lexicon, we need to recommend a few good words by searching within the lexicon those similar to the possibly mistyped word. In Web search, the implementation of the "Did you mean" feature by many search engines can benefit from the capability of finding keywords that are similar to a keyword in a search query.

Having a high performance of answering such queries is critical to these applications, especially for the cases where we want to answer queries interactively [9], or where the queries are submitted to a server by many clients. For the server case, there is a big difference between a response time of $5ms$ for a query and a response time of $20ms$, since the former means a throughput of 200 queries per second, while the latter means only 50 queries per second.

There are a variety of functions to measure the similarity between strings, including edit distance (a.k.a. Levenshtein distance), Jaccard similarity, and cosine similarity. Several algorithms have focused on approximate string queries using the edit distance function (e.g., [18, 22]), mainly due to its applicability in many scenarios. The idea of "grams" has been widely used in these algorithms. A *q-gram* of a string is a substring of length q that can be used as a signature for the string. For instance, the 3-grams of the string `giuliani` are `giu`, `iul`, `uli`, `lia`, `ian`, and `ani`. These algorithms rely on the following observation: if the edit distance between strings are within a threshold k, then they should share a certain number of common grams, and this lower bound is related to the gram-length q and the threshold k. For instance, the edit distance between `giuliani` and `guliani` is 1, and they share 4 common 3-grams. Using this observation, we can decompose each string in the given collection to grams, and build an inverted list for each gram, which includes the id of the strings in which this gram appears. Fig. 1 shows an example collection of 6 strings, and the corresponding inverted lists of their 2-grams. For a query, we can also generate its grams, and locate the corresponding

inverted lists. We then search on the lists to identify those string ids that have enough occurrences on these lists.

id	string
1	bingo
2	bioinng
3	bitingin
4	biting
5	boing
6	going

(a) Strings.

gram		string ids
bi	→	1, 2, 3, 4
bo	→	5
gi	→	3
go	→	1, 6
in	→	1, 2, 3, 3, 4, 5, 6
io	→	2
it	→	3, 4
ng	→	1, 2, 3, 4, 5, 6
nn	→	2
oi	→	2, 5, 6
ti	→	3, 4

(b) Inverted lists of string ids (default gram dictionary $\mathcal{D}_0$).

Figure 1: Strings and their inverted lists of 2-grams.

The gram length q can greatly affect the performance of these algorithms. Earlier algorithms mainly use a "try-and-see" approach. But there is a dilemma in choosing the gram length q: If we increase q, there could be fewer strings sharing a gram, resulting in shorter lists, and less time to access them. On the other hand, it may also reduce the lower bound on the number of common grams shared by similar strings, causing more false positives after accessing the lists.

Recently we proposed a new technique, called VGRAM [18], to improve the performance of these algorithms. Its main idea is to judiciously choose a *dictionary* of high-quality grams of variable lengths from the string collection based on gram frequencies. An important observation is that two similar strings should still share certain number of common grams, and the new lower bound can be computed efficiently. At a high level, VGRAM can be viewed as an additional index structure associated with the collection of strings. Experiments in [18] showed that adopting this technique in existing algorithms can reduce not only their index size, but also their query-answering time.

The following is an interesting observation: *An inverted-list index based on grams of fixed-length q can be viewed as a special* VGRAM *index structure, in which the gram dictionary only includes grams of length q.* The choice of the gram dictionary greatly affects the performance of existing algorithms. Based on this observation, several fundamental problems arise naturally: what is the fundamental relationship between the gram dictionary and the performance of queries on the string collection? If this relationship is understood, how to compute a good gram dictionary automatically? In this paper we study these issues, and make the following contributions.

- Since the lower bound on the number of common grams between similar strings affects the performance of algorithms, it is important to make this bound as tight as possible. [18] presented a simple way to compute a lower bound. In Section 3 we develop a dynamic programming algorithm that can compute a tighter lower bound.
- We formally analyze how adding a new gram to an existing gram dictionary can affect the index structure of the string collection, thus the performance of queries (Section 4). We will show that these analyses are technically very challenging and interesting.
- We develop an efficient algorithm that can automatically find a high-quality gram dictionary for the string collection (Section 5). Notice in [18] we also developed a heuristic-based algorithm for generating a gram dictionary, which requires several manually-tuned parameters. Our new algorithm does not require some of the parameters, and is cost-based. To our best knowledge, this study is the first cost-based quantitative approach to deciding good grams for approximate string queries.
- We have conducted experiments on real data sets to evaluate the proposed techniques, and show that they can indeed generate good gram dictionaries to improve performance of existing algorithms (Section 6).

2. BACKGROUND

2.1 Approximate String Queries

The *edit distance* (or Levenshtein distance) between two strings s_1 and s_2 is the minimum number of single-character edit operations (insertion, deletion, and substitution) that are needed to transform s_1 to s_2. For example, the edit distance between `masachusatts` and `massachusetts` is 2. In particular, we can transform the first string to the second by inserting a character `s` and substituting the last character `a` by character `e`. We denote the edit distance between s_1 and s_2 as $ed(s_1, s_2)$. We focus on approximate selection queries on collections of strings using this function. Let S be a collection of strings (possibly with duplicates). An *approximate selection query* on S includes a string Q and a threshold k, and its answer includes all the strings $s \in S$ such that $ed(Q, s) \leq k$. Such a query is denoted by $\sigma(Q, k)$.

2.2 Gram-Based Indices and Algorithms

Let s be a string of characters . We use "$|s|$" to denote the length of the string, "$s[i]$" to denote the i-th character of s (starting from 1), and "$s[i, j]$" to denote the substring from its i-th character to its j-th character.

Let q be a positive integer. A *positional q-gram* of s is a pair (i, g), where g is the substring of length q starting at the i-th character of s, i.e., $g = s[i, i + q - 1]$. The set of *positional q-grams* of s, denoted by $G(s, q)$, is obtained by sliding a window of length q over the characters of s. For example, suppose $q = 2$, and $s =$ `bitingin`, then $G(s, q) =$ {(1,`bi`), (2,`it`), (3,`ti`), (4, `in`), (5, `ng`), (6, `gi`), (7, `in`)}. For simplicity, in our notations we omit positional information, which is assumed implicitly to be attached to each gram.

Several algorithms [17, 22] are developed to answer approximate string queries based on inverted-list structures of q-grams. In the index, for each gram g of the strings in the collection S, we have an inverted list of the ids of the strings that include this gram. If a gram appears in a string multiple times (with different positions), the string id will appear multiple times on the inverted list of this gram, with the different positions. Fig. 1(b) shows such an index structure of 2-grams on six strings. To answer an approximate query $\sigma(Q, k)$ on S, these algorithms use various filtering techniques to prune strings. Some filters are using the following fact. If $ed(s_1, s_2) \leq k$, then the lengths $|s_1|$ and $|s_2|$ should differ by at most k, and the strings should share at least the following number of common q-grams:

$$max\{|s_1|, |s_2|\} + 1 - (k + 1) \cdot q. \quad (1)$$

In particular, a necessary condition for a string s in S to be in the answer to the query is that s should share the

following number of common grams with Q:

$$|Q| + 1 - (k+1) \cdot q. \quad (2)$$

Using the inverted-list index, we can find the set of string ids that satisfy this necessary lower-bound condition. Among these candidate strings, we remove the false positives by computing their real edit distances to the query string. If the lower bound in Equation 2 is zero or negative, then we need to scan the entire collection to find answers. This scan could become more efficient if we can apply additional filters on the strings. For instance, the length filter can help us limit our search to a subset of strings.

As an example, consider the following approximate selection query $\sigma(\texttt{bingon}, 1)$ on the six strings indexed in Fig. 1(b). We generate 2-grams from the query: {bi, in, ng, go, on}. For each of them, we access the corresponding inverted list, and find the string ids that share $|Q| + 1 - (k+1) \cdot q = 3$ common grams with the query string, which are ids 1, 2, 3, 4, and 6. Finally, we compute their real edit distances to the query string, and identify the (only) answer to the query, which is string bingo with id 1.

2.3 VGRAM

In [18] we proposed VGRAM to improve the performance of these algorithms by using a predefined *dictionary* of variable-length grams, denoted by $\mathcal{D}$. It uses a parameter, called q_{min}, to indicate that by default, all the q_{min}-grams are in the gram dictionary $\mathcal{D}$. For a string s, we generate a set of grams for s using the gram dictionary $\mathcal{D}$, denoted by $VG(s, \mathcal{D})$, or just $VG(s)$ when the dictionary $\mathcal{D}$ is clear from the context. The main idea is that we still use a sliding window to go through the characters of the string from the left to the right, while the window size can vary at different character positions, depending on the grams in $\mathcal{D}$.

We use an example to illustrate this process. Let $\mathcal{D}_0$ denote the implicit fixed-length-gram dictionary in Fig. 1(b), and $\mathcal{D}_1$ be the dictionary by adding a new gram ing to $\mathcal{D}_0$. Fig. 2 illustrates how we decompose the string bingon to grams using the new dictionary $\mathcal{D}_1$. At each character position i, we find the longest gram in the dictionary that matches a substring starting at the position i. (This step can be done efficiently when the gram dictionary is stored as a trie.) For instance, for the second position of character i in the string, we generate a gram ing, since it is the longest gram in the dictionary that matches a substring starting from this position. We keep the gram only if its corresponding substring is not subsumed by the substring of any earlier gram. For this reason, we do not keep the gram ng, since its corresponding substring has been subsumed by the substring for the gram ing. If no gram in the dictionary matches a substring starting at this position, we generate a q_{min}-gram for this position. We produce the gram on for this reason. The final set of grams is {bi, ing, go, on}.

b i n g o n

Figure 2: Decomposing string bingon to variable-length grams using the gram dictionary $\mathcal{D}_1 = \mathcal{D}_0 \cup \{\texttt{ing}\}$.

After decomposing each string in S to grams using dictionary $\mathcal{D}$, we construct the corresponding inverted lists. For the six strings in our running example, Fig. 3(a) shows the inverted lists of string ids using the dictionary $\mathcal{D}_1$. For comparison purposes, we use Fig. 3(b) shows the inverted lists using another gram dictionary $\mathcal{D}_2$, which is obtained by adding a new 3-gram bin to $\mathcal{D}_1$.

gram		string ids
bi	→	1, 2, 3, 4
bo	→	5
gi	→	3
go	→	1, 6
in	→	2, 3
ing	→	1, 3, 4, 5, 6
io	→	2
it	→	3, 4
ng	→	2
nn	→	2
oi	→	2, 5, 6
ti	→	3, 4

(a) Use dictionary $\mathcal{D}_1$.

gram		string ids
bi	→	2, 3, 4
bin	→	1
bo	→	5
gi	→	3
go	→	1, 6
in	→	2, 3
ing	→	1, 3, 4, 5, 6
io	→	2
it	→	3, 4
ng	→	2
nn	→	2
oi	→	2, 5, 6
ti	→	3, 4

(b) Use dictionary $\mathcal{D}_2$.

Figure 3: Inverted lists using dictionaries of variable-length grams. Each "gram" column includes the grams in the corresponding dictionary. Each bold gram is a newly added gram in the dictionary.

The process of answering a query is the same as that of fix-length grams. The only difference is that we could use a different lower bound. (See Section 3.2 for details.) For instance, consider the same approximate query $\sigma(\texttt{bingon}, 1)$. We first generate positional grams {bi, ing, go, on}. We can show that a string in the answer to this query should share at least 2 common grams with the query string. We use the new lists with the new bound to find the answer. Compared to the lists of fixed-length grams, now we can access shorter inverted lists. This example shows the main advantages of VGRAM on reducing the index size and query time.

3. TIGHTENING LOWER BOUNDS ON NUMBER OF COMMON GRAMS

A lower bound on the number of common grams shared by two similar strings affects the performance of a query in two ways. First, it affects how efficiently we can access the inverted lists of the grams in the query. Second, it decides how many strings become candidate answers to the query after accessing these lists. Therefore, it is critical to make this lower bound as tight as possible. In [18] we gave a simple way to compute this lower bound in the VGRAM technique. In this section, we develop a dynamic programming algorithm for computing a tighter lower bound.

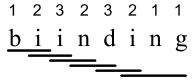
3.1 Effect of Edit Operations on Grams

Let $\mathcal{D}$ be a gram dictionary in VGRAM, using which we decompose strings to grams. We first see how edit operations on a string s affect its grams in $VG(s, \mathcal{D})$. Consider the i-th character $s[i]$. If there is a deletion operation on this character, we want to know how many grams in $VG(s, \mathcal{D})$ could be affected by this deletion, i.e., they may no longer be generated from the new string after this deletion operation. In [18] we showed how to efficiently compute an upper bound on this number by using two tries of the grams in the dictionary $\mathcal{D}$. Let $B[i]$ be the computed upper bound. For simplicity, deletions and substitutions on this character and insertions immediately before and after this character are all called "edit operations at the i-th positions." The following proposition shows that this $B[i]$ value is indeed an

upper bound on the number of destroyed grams due to all possible edit operations at this position.

PROPOSITION 3.1. *Let string s' be obtained by doing edit operations at the i-th position of a string s. The number of grams in $VG(s, \mathcal{D})$ that will no longer exist in $VG(s', \mathcal{D})$ is at most $B[i]$.*

We call $\langle B[1], B[2], \ldots, B[|s|]\rangle$ the *Position-Gram-Bound Vector* (or "PGB Vector") of the string s. For example, consider a string `biinding` and the gram dictionary shown in Fig. 3(a). The grams generated from this string are shown in Fig. 4(a), which also shows the PGB vector for the string. For instance, the value 3 on the character `d` means that any number of edit operations at this position can destroy at most 3 grams of this string. Fig. 4(b) shows the grams that could be affected by edit operations at each position.



(a) Grams and PGB vector.

Position	Affected grams
b	bi
i	bi, ii
i	bi, ii, in
n	in, nd
d	in, nd, di
i	di, ing
n	ing
g	ing

(b) Affected grams.

Figure 4: Position-gram-bound (PGB) vector.

3.2 Tightening Lower Bounds Using Dynamic Programming

From this PGB vector, we can compute a "number-of-affected-gram vector" ("NAG vector" for short) for the string. The k-th value in this vector, denoted by $NAG(s, k, \mathcal{D})$ (or just $NAG(s, k)$ when the dictionary $\mathcal{D}$ is clear from the context), indicates that if there are k edit operations on this string, then there are at most this number of grams that will no longer exist after these edit operations. It has been shown in [18] that if two strings satisfy $ed(s_1, s_2) \leq k$, then they should share at least the following number of grams (compared to the bound in Equation 1):

$$max\big(|\,VG(s_1)| - NAG(s_1, k), |\,VG(s_2)| - NAG(s_2, k)\big). \quad (3)$$

In particular, for a given query $\sigma(Q, k)$, the following is a lower bound on the number of common grams between a string s and Q when $ed(s, Q) \leq k$ (compared to the bound in Equation 2):

$$\mathcal{B}(Q, k) = |\,VG(Q)| - NAG(Q, k). \quad (4)$$

When answering a query, we often use the bound in Equation 4 mainly because it is only dependent upon the query, not a string in the collection. For each candidate string satisfying this bound, we could further do some pruning using the possibly tighter bound in Equation 3.

Based on this analysis, the values in NAG vectors affect the lower bound on the number of common grams between similar strings, and ideally we want these bounds to be as tight as possible. One way to compute the $NAG(s, k)$ value, as proposed in [18], is to take the summation of the k largest values in the PGB vector of string s. We call it the "k-max algorithm." For example, consider the query string `biinding` in Fig. 4(a). Assume $k = 2$. To compute NAG("`biinding`",2), we could take the summation of the 2 largest values in the PGB vector, which is $3 + 3 = 6$. The bound given by this pessimistic approach could be loose, since in some cases, those largest numbers might represent overlapping grams. In the running example, the two positions (3 and 5) with the largest bound value are close to each other. Fig. 4(b) shows that for the edit operations on these two positions, their sets of affected grams overlap, and both share the gram `in`. The total number of affected grams is 5 (instead of 6), which are `bi`, `ii`, `in`, `nd`, and `di`. In fact, a tighter value for NAG("`biinding`", 2) is 5.

We develop a dynamic programming algorithm for computing tighter-bound values for the NAG vector.

Subproblems: We create subproblems as follows. Let $0 \leq i \leq k$ and $0 \leq j \leq |s|$ be two integers. Let $P(i, j)$ be an upper bound on the number of grams in $VG(s, \mathcal{D})$ that can be affected by i edit operations that are at a position *no greater than* j. Our final goal is to compute a value for $P(k, |s|)$.

Initialization: For each $0 \leq i \leq k$, we have $P(i, 0) = 0$. For each $0 \leq j \leq |s|$, we have $P(0, j) = 0$.

Recurrence Function: Consider the subproblem of computing a value for the entry $P(i, j)$, where $i > 0$ and $j > 0$. We have two options.

- Option (1): We do not have an edit operation at position j. In this case, we can set $P(i, j)$ to be $P(i, j-1)$, since all the j edit operations occur before or at position $j-1$.
- Option (2): We have (possibly multiple) edit operations at position j. These operations could affect at most $B[j]$ grams of s, where $B[j]$ is the j-th value in the PGB vector of this string. For all the grams of s that start from a position before j that cannot be affected by these edit operations at the j-th position, let $R(j)$ be their largest starting position. (This number can be easily computed when we compute the PGB vector for this string.) Therefore, $P(i, j)$ can be set as the summation of $P(i-1, R(j))$ and $B[j]$, assuming in the worst case we have $i - 1$ edit operations on positions before or at j.

For these two cases, we can assign their maximal value to the entry $P(i, j)$. The following formula summarizes the recurrence function:

$$P(i, j) = \max \begin{cases} P(i, j-1), & \text{(no operation at } j\text{)} \\ P(i-1, R(j)) + B[j]. & \text{(operations at } j\text{)} \end{cases}$$

Using the analysis above, we can initialize a matrix of size $(k+1) \times (|s|+1)$. We set the values in the first row and the first column to be 0. We use the recurrence function to compute the value of each entry, starting from the top-left entry, until we reach the right-bottom entry. The rightmost column will give us an NAG vector for the string. For example, consider the query $s =$ `biinding` and the gram dictionary shown in Fig. 3(a). Fig. 5 shows the matrix to calculate an NAG vector for this string. The dotted line shows the steps to compute the value at the right-bottom entry. The last column gives us an NAG vector, which is $\langle 0, 3, 5\rangle$.

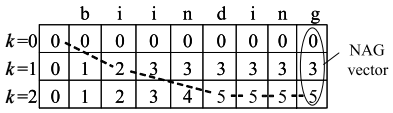


Figure 5: Matrix in dynamic programming.

4. ADDING A GRAM TO A DICTIONARY

To decide a good gram dictionary for a string collection S to answer approximate queries on S efficiently, we need to understand the fundamental relationship between the gram dictionary and query performance. In this section, we study the following problem: how does the addition of a new gram g to an existing gram dictionary $\mathcal{D}$ of S affect the index structure of S, thus affect the performance of queries using the index? At a high level, the new gram will have the following effects. (1) The inverted-list index structure of S will have one more inverted list for this gram g, which will "suck" some string ids from the inverted lists of some grams related to g. For instance, if $g =$ `abc`, then the new list of g will take some string ids from the list of the gram `ab` and some from the list of the gram `bc`. (2) For an approximate string query, it could generate a different set of grams, and their corresponding inverted lists could become shorter. In addition, the new gram can also affect the NAG vector for the query string. Thus it could result in a different set of candidate strings for the query using the possibly new lower bound on the number of common grams between similar strings. Next we will analyze the details of these effects.

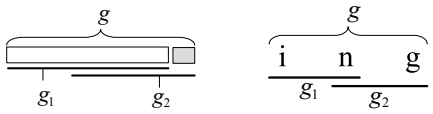
4.1 Effects on Inverted Lists

We first study how adding a gram affects the inverted lists on S. We introduce the following concepts.

Definition 1. Let g_i be a gram in $\mathcal{D}$. The *complete list of* g_i, denoted by $C(g_i)$, includes all the ids of the strings in S that include this gram. The *local list of* g_i *with respect to a gram dictionary* $\mathcal{D}$, denoted by $L(g_i, \mathcal{D})$, includes all the ids of the strings whose decomposed grams using a gram dictionary $\mathcal{D}$ include the gram g_i.

The lists shown in earlier figures are all local lists. When using fixed-length grams, the complete and local lists for a gram are always the same. For instance, the lists in Fig. 1(b) are both the complete and local lists for the corresponding grams. In general, $L(g_i, \mathcal{D})$ is always contained in $C(g_i)$. However, if a string id appears on the complete list $C(g_i)$, it might not appear on the local list $L(g_i, \mathcal{D})$, since the string might not generate this gram g_i due to the fact that this gram is subsumed by another gram of the string. We will see that these lists can help us analyze the effects of adding a gram to an existing dictionary, and quantify the performance improvement on queries. Thus we want to incrementally maintain these lists after adding a gram. In the rest of this paper, we refer the local list of a gram as "the list of a gram," unless specified otherwise. Notice that the set of complete lists of grams is a temporary index structure used in the process of generating a gram dictionary. After that, the final index structure does not keep these complete lists.

Let g_1 be an existing gram in $\mathcal{D}$, and g be a new gram obtained by appending a character to the end of g_1. Let g_2 be the longest suffix of g that exists in $\mathcal{D}$. Fig. 6(a) illustrates these grams, and Fig. 6(b) shows an example, where $g_1 =$ `in`, $g =$ `ing`, and $g_2 =$ `ng`. Let the new gram dictionary $\mathcal{D}' = \mathcal{D} \cup \{g\}$. We next show how to obtain the complete and local lists for this new gram, how the new gram affects the local lists of the existing grams. Notice that complete lists of grams never change, and this new gram g does not affect the local lists of the existing grams except those of g_1 and g_2.



(a) Extend g_1 to g. (b) An example.

Figure 6: Extending a gram to a new gram.

The complete list $C(g)$ can be obtained by scanning the string ids on the complete list $C(g_1)$. For each occurrence of a string s in $C(g_1)$, consider the corresponding substring $s[i, j] = g_1$. If the corresponding longer substring $s[i, j+1] = g$, then we do the following: (1) Add this string id to the complete list of the gram g; (2) Remove this occurrence of string id s from the local list of g_1 (if any); and (3) Remove the occurrence of string id s from the local list of g_2 (if any). As a result, the local lists of grams g_1 and g_2 could shrink.

The process of computing the local list $L(g, \mathcal{D}')$ is more subtle. Clearly $L(g, \mathcal{D}') \subseteq C(g)$. One question is whether we could assign all the ids in $C(g)$ to $L(g, \mathcal{D}')$. The following example shows that we cannot do this simple assignment in some cases. Consider the example in Fig. 6(b), in which we add the new gram $g =$ `ing`. If the original dictionary had a gram `ingo`, whose local list has a string `bingo`, then this string id should not appear on the local list of the new gram `ing`, because this string will not generate an `ing` gram using the new dictionary. This example shows that in the worst case, to compute $L(g, \mathcal{D}')$, we might need to access the lists of all grams in dictionary $\mathcal{D}$ that have this new gram g as a substring, and identify some string ids to be removed from the complete list $C(g)$. Clearly this process can be very inefficient. However, the following lemma shows in some cases we do not need this expensive process.

LEMMA 1. *If the new gram g is a longest gram in the new dictionary $\mathcal{D}'$, then $L(g, \mathcal{D}') = C(g)$.*

This lemma says that if each time we choose a new gram that does not have a longer gram in the original dictionary, then we can safely assign the complete list $C(g)$ to its local list $L(g, \mathcal{D}')$. In the rest of this section, we assume the new gram has this property, and our algorithm in Section 5 can satisfy this requirement easily by doing a breadth-first traversal of the trie.

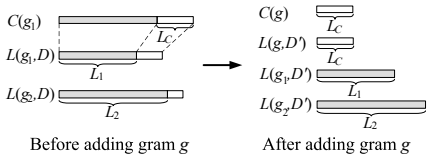


Figure 7: Changing inverted lists of grams g_1 and g_2 after adding a new gram g.

As summarized in Fig. 7, adding g to the existing gram dictionary $\mathcal{D}$ will introduce a complete list $C(g)$ and an identical local list $L(g, \mathcal{D}')$ for this new gram, which will take some string ids from the original local lists $L(g_1, \mathcal{D})$ and $L(g_2, \mathcal{D})$ of g_1 and g_2, respectively. As a result, the total index size (without the complete lists) will decrease.

4.2 Effects on Lower Bounds

Consider a query $\sigma(Q, k)$. Equation 4 in Section 3.2 shows a lower bound on the number of common grams between the

Query Q	Dictionary	Gram set	$NAG(Q,1)$	Lower bound	Candidate ids
bingon	$\mathcal{D}_0$	{bi, in, ng, go, on}	2	3	1, 2, 3, 4, 6
	$\mathcal{D}_1$	{bi, ing, go, on}	2	2	1, 3, 4, 6
	$\mathcal{D}_2$	{bin, ing, go, on}	2	2	1, 6
bitting	$\mathcal{D}_0$	{bi, it, tt, ti, in, ng}	2	4	3, 4
	$\mathcal{D}_1$	{bi, it, tt, ti, ing}	2	3	3, 4
	$\mathcal{D}_2$	{bi, it, tt, ti, ing}	3	2	1, 3, 4

Table 1: Effects of adding a new gram on a dictionary on queries when edit-distance threshold $k = 1$. In the last column, each underlined string id is in the answer to the query.

query string Q and a string s in the answer. The lower bound is decided by both the number of grams in $VG(Q)$ and the k-th value in the NAG vector, $NAG(Q,k)$. Adding a gram to $\mathcal{D}$ could affect both numbers. [1] Consider the strings in Fig. 1(a) and two queries, $Q_1 =$ `bingon` and $Q_2 =$ `bitting`, both with an edit-distance threshold 1. Table 1 shows the different effects of adding a new gram to an original dictionary on the two queries.

Effects on $VG(Q)$: If the query string Q does not include a substring of gram g, clearly adding g to the dictionary will not affect $VG(Q)$. If Q does include a substring of g (possibly multiple times), for each of them $Q[i,j] = g$, consider the longest suffix gram g_2 of g that exists in the original dictionary (Fig. 6(b)). There are two cases.

- If this substring $Q[i,j]$ produces a gram g_2 in $VG(Q)$, then this gram will be subsumed by the new gram g using the new dictionary. Thus it will not appear in the new set of grams for Q, causing the size $|VG(Q)|$ to decrease by 1. For example, consider string `bingon` in Table 1. Using $\mathcal{D}_0$, `bingon` produces five grams. Using $\mathcal{D}_1$, which has included one more gram `ing`, the string produces four grams. The gram `in` has been replaced by a new gram `ing`, while `ng` is no longer produced.
- If substring $Q[i,j]$ does not produce a gram g_2 using the original dictionary, then this substring will produce the same number of grams in $VG(Q)$ using the new dictionary. For instance, for the string `bingon` in Table 1, when we add the gram `bin` to $\mathcal{D}_1$ to get $\mathcal{D}_2$, the number of grams for the string does not change.

To summarize, after adding g to the dictionary, the size $|VG(Q)|$ will either remain the same or decrease.

Effects on $NAG(Q,k)$: Take the string `bitting` in Table 1 as an example. The addition of the new gram `bin` to the dictionary $\mathcal{D}_1$ causes value $NAG($`bitting`$, 1)$ to increase from 2 to 3. In general, if string Q does not include the gram g_1 nor gram g_2, then adding the gram g to the dictionary will not affect the NAG vector of Q. Otherwise, g could affect the vector in the following way. It could cause the values of the PGB vector of the string to increase, because an edit operation at a position could destroy more grams. In particular, due to the addition of g, a position in string Q could either overlap with more grams, or an operation at this position could destroy one more gram that cannot be destroyed before adding g. For instance, in the string `bingon` in our running example, consider its third character `n`. After adding `bin` to $\mathcal{D}_1$ to get $\mathcal{D}_2$, the PGB value at this position changes from 1 to 2. The reason is that this position overlaps with one gram when $\mathcal{D}_1$ is used. When we use $\mathcal{D}_2$, this position overlaps with two grams. Notice that increasing a value in the PGB vector does not necessarily increase the final NAG values, because the latter are computed using the dynamic programming algorithm presented in Section 3.2.

Table 2 summarizes the effect of a substring $Q[i,j]$ on the lower bound of the query string after the new gram g is added to the dictionary. Take case 2 as an example. It shows that if the substring includes one of g_1 and g_2, but not g, then after the gram g is added to the dictionary, this substring will cause the lower bound of Q either unchanged, or to decrease by 1.

4.3 Effects on Candidates

The above analysis has shown that, after adding the gram g to $\mathcal{D}$, the local inverted lists of the data collection can change, the query string Q could generate a new set of grams, and the lower bound $|VG(Q)| - NAG(Q,k)$ could decrease. As a consequence, the set of candidates satisfying the new lower bound can also change. Table 2 also summarizes the effects of a substring $Q[i,j]$ on string candidates after the gram g is added. For case (1), the substring does not include the gram g_1, nor gram g_2. Thus this addition of g will not affect the candidates for the query (represented as "No change" in the table).

For case (2), the substring includes only one of the grams g_1 and g_2, but not g. The set of grams for this string will not change, i.e., $VG(Q,\mathcal{D}) = VG(Q,\mathcal{D}')$. As shown in Fig. 7, the corresponding local inverted list of g_1 or g_2 will change. For those string ids on this list that are not moved to the local list of the new gram, as represented as L_1 or L_2 in the figure, if the lower bound does not change, then whether these string ids are candidates for the query or not will not be affected by this new gram (represented as "No change" in the table). If the bound has decreased by 1, then some of the string ids on this list, which are not candidates before, could become candidates, as represented as "No → Yes" in Table 2. This case is "bad" since more string ids need to be postprocessed. For those string ids that are moved to the local list of g, represented as L_c in the figure, if the lower bound does not change, some of them that are candidates before adding g might no longer be candidates after adding g, represented as "Yes → No" in the table. This case is good since more false positives have been pruned. If the bound decreases by 1, then some of them who are not candidates before can become candidates, indicated by "No → Yes". Similar analyses can be done for the other two cases.

5. ALGORITHM FOR GENERATING A GRAM DICTIONARY

We develop an algorithm for automatically generating a high-quality gram dictionary $\mathcal{D}$ for a string collection S to support queries on S. From Section 4 we can see that adding

[1] Although this discussion is based on a query string, the result is also valid for a string in the collection S in general.

Case	Condition on $Q[i,j]$	Lower Bound	Candidates satisfying lower bound
1	No g_1 and no g_2	No change	No change
2	One of g_1 and g_2	No change or -1	No change, "Yes→ No", or "No → Yes"
3	g	No change or -1	No change or "Yes→ No"
4	g_1 and g_2, but no g	No change, -1, or -2	No change, "Yes→ No", or "No → Yes"

Table 2: Effects of a substring $Q[i,j]$ of a query string Q on the lower bound $|VG(Q)| - NAG(Q,k)$ and the string candidates satisfying the lower bound after the gram g is added to the dictionary.

a gram to an existing dictionary can have different effects on the performance of different queries. Therefore, choosing a good gram dictionary has to be *cost based*, and it depends on a given workload of queries. We thus assume we are given a set of queries $W = \{\sigma(Q_1, k_1), \ldots, \sigma(Q_n, k_m)\}$ on the string collection S, and we want to generate a gram dictionary to optimize the *overall* performance of these queries, measured by their average query time.

5.1 Algorithm Overview

Fig. 8 formally describes the algorithm. Its main idea is the following. We first sample some strings from S to get a new collection S' to be used to generate inverted lists for the grams (line 1). The reason to do sampling is to reduce the space requirement by the algorithm and the estimation time for each new possible gram to add.

Algorithm: GramGen
Input: String collection S, query workload W, and q_{min};
Output: a gram dictionary as a trie $\mathcal{T}$.
Method:
(1) Get a sample collection S' from strings in S;
(2) Generate q_{min}-grams for strings in S';
(3) Use the grams to construct a trie $\mathcal{T}$;
(4) Initialize each leaf node's complete list and (identical) local list of string ids using the generated grams;
(5) Queue $\mathcal{Q}$ = {leaf nodes in $\mathcal{T}$};
(6) WHILE ($\mathcal{Q}$ is not empty) {
(7) Queue for the next level: $\mathcal{Q}_{new} = \emptyset$;
(8) Reorder queue $\mathcal{Q}$;
(9) FOR (each node n in $\mathcal{Q}$) {
(10) $n = \mathcal{Q}.pop()$; // gram "g_1" in Fig. 6(a)
(11) Use strings on the complete list of n to add children to n by appending a character;
(12) FOR (each child node c of n) {
// c corresponds to new gram "g" in Fig. 6(a)
(13) Compute the complete and local lists of c using the complete list of n;
(14) Find the node n' in $\mathcal{T}$ whose gram is the longest suffix of the gram for c;
// n' corresponds to gram "g_2" in Fig. 6(a)
(15) IF (**Evaluate**$(\mathcal{T}, n, c, n', W)$ returns "yes") {
// Add this new gram to the dictionary
(16) $\mathcal{Q}_{new}.push(c)$;
(17) Adjust local lists of n and n';
}
ELSE
// do not add this gram to the dictionary
(18) Remove node c and its lists;
} // "IF"
} // "FOR"
(19) Remove complete list of n; // no longer useful
} // "FOR"
(20) $\mathcal{Q} = \mathcal{Q}_{new}$; // process the next level
} // "WHILE"
(21) RETURN $\mathcal{T}$;

Figure 8: Algorithm for generating a gram dictionary.

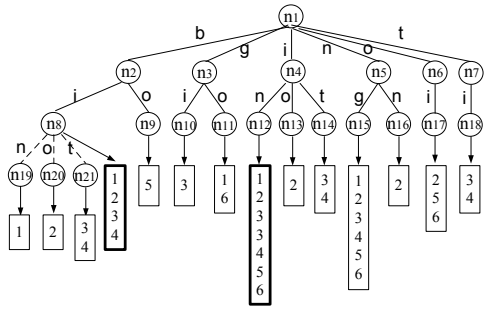
In each iteration, we store the current gram dictionary as a trie, in which all the paths starting from the root of length at least q_{min} correspond to the grams in the dictionary. In order to efficiently maintain these lists incrementally after each new gram is added (line 16), we want to minimize the number of lists that need to be updated. Lemma 1 in Section 4.1 shows that if each new gram has the longest length among the existing grams, we only need to update the lists of two existing grams using the procedure discussed in Section 4.1 (line 17). For this reason, we do a *breadth-first traversal* of the trie to add grams, and grow the grams level-by-level. This traversal is implemented by using two queues: the queue $\mathcal{Q}$ includes the leaf nodes of the current level, and the queue $\mathcal{Q}_{new}$ includes the leaf nodes of the next level to be further considered (lines 5, 6, 7, 16, and 20).

In the algorithm we maintain both the complete list and local list of string ids for each gram, which have several advantages. (1) They can help us select an extended gram (line 11). (2) They can be used to estimate the performance improvement (if any) by adding an extended gram to the dictionary (line 15). (3) If the sampled strings include all the strings in S, then after the algorithm terminates, we will already obtain the (local) inverted lists of grams for the final dictionary, thus we do not need another scan of the data set to generate the inverted-list index structure. Notice that after we finish processing all the extended grams for a node n, we can safely remove the complete list of n to save space, since this complete list is no longer needed (line 19).

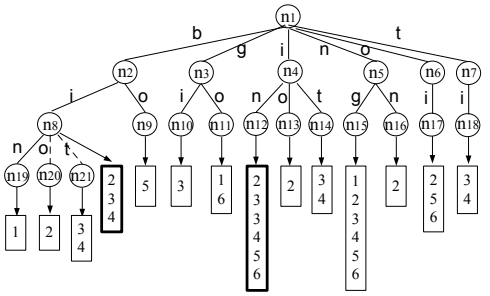
There are different orders to visit the current leaf nodes to identify grams to extend. As in [18], we can use three orders. (1) SmallFirst: Choose a child with the shortest local list; (2) LargeFirst: Choose a child with the longest local list; (3) Random: Randomly select a child. We implement such an order in line 8. In line 11, we need to find possible extended grams for an existing gram at node n as follows. For each string s on the complete list of n, consider the substring $s[i,j]$ corresponding to the gram of n. If the string has a substring $s[i,j+1]$, we add a child of node n with the character $s[j]$, if this child is not present.

EXAMPLE 1. *We use Fig. 9 to show one iteration in the algorithm. Fig. 9(a) shows a trie for the six strings in Fig. 1(a). For simplicity, we only draw the local lists for the grams. So far we have processed all the grams of length $q_{min} = 2$. Now we consider expanding the node n_8, which corresponds to the gram* `bi`*. We generate its three children corresponding to the extended grams* `bin`*,* `bio`*, and* `bit`*, by scanning the strings on the complete list of* `bi`*, which is the same as its local list so far: $\langle 1, 2, 3, 4\rangle$. For each of the children, we compute their complete and (identical) local lists by scanning the complete list of n_8, and identifying the substrings corresponding to this new gram. At this stage, we do not change the local list of n_8 yet, since we have not decided whether some of the extended grams will be added. Suppose by running the function* Evaluate() *we decide to add the gram*

`bin` *to the dictionary. In this case, we modify the local list of node* n_8 *and the local list of gram* `in` *(node* n_{12}*). The new trie is shown in Fig. 9(b).* □



(a) Before adding gram `bin`.



(b) After adding gram `bin`.

Figure 9: One step of adding a gram in algorithm GramGen**. A list with a thicker line is modified. Each dotted line points to a child that could potentially be added.**

5.2 Estimating the Benefits of a Gram

A critical step in the algorithm is to decide whether adding a candidate gram g for child node c can improve the overall performance of the query workload (procedure Evaluate in line 15). A naive way to do this evaluation is to run the query workload W using the new gram dictionary after adding g, and compare the performance with that on the original dictionary $\mathcal{D}$. If it does improve the performance, then we add g to the dictionary. Notice that the trie already includes the local lists for the grams, which can be maintained incrementally by modifying just three lists for each potential new gram. Thus re-running the query workload does not require rebuilding the inverted-list index structure. Still this approach could be computationally prohibitive since it needs to rerun the queries for each potential new gram.

In order to do this evaluation efficiently, we can *estimate* the effect on the query performance by adding the gram g corresponding to the child node c. Recall that the time to answer a query $\sigma(Q,k)$ mainly consists of two parts: the time to access the inverted lists of the generated grams of string Q, and the time to postprocess the candidates to remove false positives by computing their distances to Q. Based on the analysis in Section 4, we know that adding the gram g has several effects on the query performance. Next we will discuss how to estimate these effects quantitatively.

We first estimate how adding the gram g affects the inverted lists. When we expand the node n by adding the children as potential grams, we have already computed the complete list and local list for each child, including g. Thus we know their local lists if g is added. But we do not know the new local lists of nodes n and n' (corresponding to "g_1" and "g_2" in Fig. 6(a) respectively) after g is added. The size of each of the two new lists can be estimated by assuming all the string ids in the new list of g will be removed from the original local lists of n (for g_1) and n' (for g_2).

The exact amount of time to access the inverted lists of the grams from a query string Q depends on the specific algorithm used for this process. In the literature there are several algorithms for doing this step [22, 17]. Different algorithms have different time complexities in terms of the lengths of these lists. As an illustrative example, we use the *HeapMerge* algorithm in [22] to show how changes on the lists affect the performance of accessing the lists using this merge algorithm. The main idea of this algorithm is to have a cursor on each list, and maintain a heap for the string ids currently pointed by the cursors on the lists. In each iteration we process the top element (a string id) on the heap, and count the frequency of this id. We add the string id to the set of candidates if its frequency is at least the specified threshold. We move the cursor of this list to the next one, and add its new id back to the heap. We repeat the process until all the ids on the lists are processed. If h be the number of grams of string Q, and M be the total size of the lists of these grams, then the time complexity of this algorithm is $O(M \log h)$.

Now let us see how the changes of the lists affect the performance of accessing the lists when answering the queries in the workload W. For each query $\sigma(Q,k)$, we use the four cases described in Table 2 to discuss how to do the analysis. Let $h = |VG(Q,\mathcal{D})|$, and M be the total length of the lists of these grams, and H be the average of the h values for all the queries in W.

- Case 1: If Q does not have a substring of g_1 nor g_2, then the time to access the lists of grams of Q does not change after adding g.
- Case 2: If Q has only one substring of g_1 or g_2, then after adding g, the new list-access time can be estimated as: $\alpha\big((M-|L(g,\mathcal{D}')|)\log h\big)$, in which α is a constant to convert list length to running time. (This constant can be easily maintained by running some sample queries.) So the reduced time can be estimated as: $T_2 = \alpha\big(|L(g,\mathcal{D}')|\log H\big)$.
- Case 3: If Q has only one substring of g, then after adding g, the new list-access time can be estimated as $\alpha\big((M - |L(g_1,\mathcal{D})| - |L(g_2,\mathcal{D})| + |L(g,\mathcal{D}')|)\log(h+1)\big)$. So the reduced time can be estimated as $T_3 = \alpha\big((|L(g_1,\mathcal{D})| + |L(g_2,\mathcal{D})| - |L(g,\mathcal{D}')|)\log H\big)$.
- Case 4: If Q has both g_1 and g_2, but no g, then after adding g, the new list-access time can be estimated as $\alpha\big(M - 2|L(g,\mathcal{D}')|\log(h)\big)$. So the reduced time can be estimated as $T_4 = \alpha\big(2|L(g,\mathcal{D}')|\log H\big)$.

These formulas can be adjusted accordingly if Q has multiple substrings satisfying these conditions. One way to estimate the overall effect on the list-access time by the query workload W is to go through the queries one by one, and apply the above formulas for each of them, and compute the summation of their effects. An alternative way, which is more efficient, is the following. We build another trie for the grams from the queries. This trie, denoted by $\mathcal{T}_W$, is constructed and maintained incrementally in the same way as the trie for the strings in the collection S during the run-

ning of the algorithm GramGen. Using this trie $\mathcal{T}_W$, we can compute how many queries are in each of the cases 2 - 4. Let p_2, p_3, and p_4 be the number of queries in W belonging to cases 2, 3, and 4, respectively. The overall reduction on the list-access time for the queries can be estimated as $p_2 \times T_2 + p_3 * T_3 + p_4 * T_4$.

We use the same methodology to estimate the overall effect of adding gram g on the number of candidates. The main idea is to consider queries in cases 2 - 4. Using the analysis in Section 4, we classify the queries in each case based on the effects of g on their lower bounds and numbers of candidates. We multiply the number of queries and the estimated benefit for each query, and compute the summation of these benefits. In addition, recent techniques as presented in [19] can also be used to do such estimations.

6. EXPERIMENTS

In this section, we report our experimental results of the proposed techniques. We used three real data sets to do the evaluation, as summarized in Table 3. The data set of "Article Titles" included article titles downloaded from the DBLP Bibliography.[2] The data set of "Movie Titles" included movie titles downloaded from the Web site of the Internet Movie Database (IMDB)[3]. The data set of "Actor Names" included actor names from the same IMDB site.

Data set	String #	Length			Range of # of injected edit operations
		Min	Max	Avg	
Article Titles	277,000	6	207	66	[1, 6]
Movie Titles	855,000	8	249	35	[1, 3]
Actor Names	1,200,000	4	74	17	[1, 2]

Table 3: Descriptions of the original data sets.

For each data set, we generated query workloads as follows. We randomly chose a string from the data set, and introduced a random number of edit operations on the string at random positions. The range of this random number is shown in the last column in Table 3. Let Q be the new string, and k be this random number. We formed an approximate string query $\sigma(Q, k)$. We generated a query workload W of 1,000,000 strings for a data set. We selected a subset W_1 of the query workload, and ran our algorithm to generate a gram dictionary for the data set using W_1. We evaluated the quality of the gram dictionary by running another subset W_2 of queries in W, assuming W_1 and W_2, both from W, have the same probabilistic distribution. We measured their average running time on the dictionary. In the experiments the number of queries in workload W_2 was always $1,000$. We used the DivideSkip algorithm in [17]. For each original data set, whenever larger data sets were needed, we used the same procedure described above to add new strings by sampling strings in the data set and introducing random edit operations to the new strings. We implemented the three policies of choosing children to extend (line 8 in Fig. 8). The observation was consistent with our results in [18]: the LargeFirst policy consistently gave the best performance. Thus in the experiments we used this policy.

All the algorithms were implemented using GNU C++. The experiments were run on a Dell GX620 PC with an Intel Pentium 2.40GHz Dual Core CPU and 2GB memory with a 250GB disk, running an Ubuntu (Linux) operating system. Index structures were assumed to be in memory.

[2]www.informatik.uni-trier.de/∼ley/db

[3]www.imdb.com

6.1 Effect of Tightening Lower Bounds Using Dynamic Programming

We first evaluated the effect of tightening lower bounds using the dynamic programming algorithm (Section 3). We used a data set with one million actor names and constructed a gram dictionary using the following setting: $100,000$ sampled strings, $5,000$ queries in workload W_1, and $q_{min} = 4$. We used the algorithm in [18] for computing a lower bound of the number of common grams between strings by choosing k largest values in the PGB vector of the string. This algorithm is denoted by k-Max. We also implemented our new dynamic programming algorithm for computing a lower bound, denoted by DP. Fig. 10(a) shows the different average lower bounds for different edit distance thresholds computed by these two algorithms. The x-axis is the edit distance threshold for different queries in the workload W_2. Fig. 10(b) shows how the average query time changed when we increased the number of strings in a data set by selecting a subset of strings in the original data set.

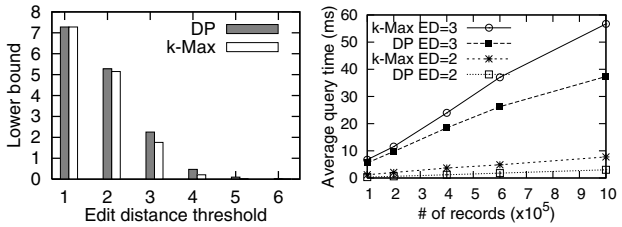
When the threshold was 1, both algorithms computed the same lower bounds with the average value, 7.28. When the threshold was 2, the average lower bound computed by k-Max was 5.15, while it was 5.28 by DP. Although the lower bound computed by DP increased slightly, the performance had a better improvement. For instance, when we used the data set of 1 million strings, the average query time using the bound of k-Max was $9.8ms$, while the time was reduced to $2.8ms$ by using the lower bounds computed by DP! When the edit distance threshold was 3, the average lower bound by k-Max was 2.25, while it was 1.76 by DP. This tighter lower bound reduced the average query time from $50.7ms$ to $23.3ms$ for the data set of 1 million strings. We observed similar reductions on other edit distance thresholds, and did not report the results in Fig. 10(b).

Figs. 10(c)-(f) show the results for the other two data sets. The dynamic programming algorithm also computed tighter lower bounds, and thus reduced query time. The time reduction for these two data sets was not as significant as that for the data set of actor names. The main reason is that the strings in the last two data sets were longer, which resulted in larger lower bounds. These relatively large lower bounds can already prune many string candidates, and tightening these bounds did not improve the query performance too much. As a conclusion, the benefits of using the dynamic programming algorithm are more significant when the query performance is very sensitive to changes of lower bounds, especially for relatively short strings.

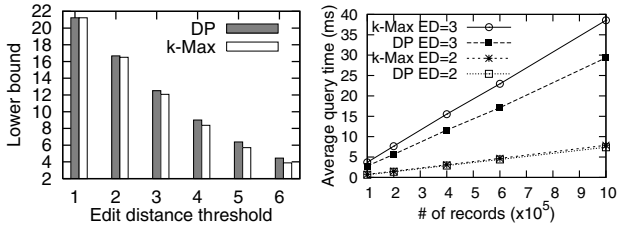
6.2 Quality of Gram Dictionary

We evaluated the algorithm GramGen for generating gram dictionaries. For each of the three original data sets, we generated a data set with 1 million strings. We used $q_{min} = 5$ for article titles, and $q_{min} = 4$ for movie titles and actor names. For each data set, we fixed a workload W_1 of queries, and varied the sampling ratio in the algorithm from 0.1% to 4%. We also fixed a sampling ratio, and varied the number of queries in the workload.

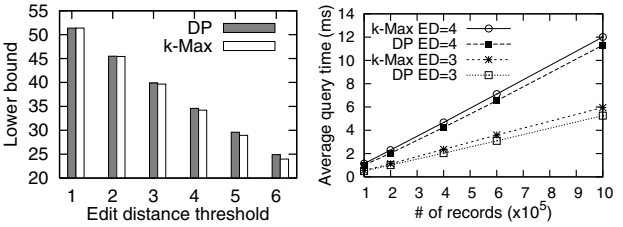
Accuracy of Gram-Benefit Estimation: The quality of its final dictionary depends on how accurately we can es-



(a) Actor names. (b) Actor names.



(c) Movie titles. (d) Movie titles.



(e) Article titles. (f) Article titles.

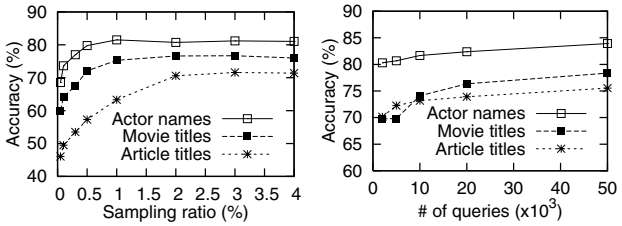
Figure 10: k-Max **and dynamic programming.**

timate the benefits of a new gram. We conducted experiments to evaluate the accuracy of the Evaluate estimation method. For each potential gram g, we used a brute force approach by running the queries in the workload W_1 using the dictionary $\mathcal{D}$ without the gram g, and computing their average time. We also ran the queries with the dictionary $\mathcal{D} \cup \{g\}$, and measured the average time. Notice that this step, which was time consuming, is mainly for comparison purposes, and it is not needed in our algorithm. If the average time decreases, we should add this gram to $\mathcal{D}$. We also *estimated* the benefits of gram g, as described in Section 5.2, and made a decision about whether to add g. If this decision is consistent with the decision of the brute force approach, we call it a *correct decision*. We measured the accuracy of the estimation as follows:

$$\text{Accuracy} = \frac{\text{\# of correct decisions}}{\text{\# of evaluated grams}} \times 100\%.$$

Fig. 11(a) shows the accuracy results for different sampling ratios for the three data sets, when we used $4,000$ queries in workload W_1. When the sampling ratio was 0.5% ($5,000$ strings), the accuracy was already 79.8% for the data set of actor names, 72.1% for the movie titles, and 57.3% for the article titles. As we further increased the sampling ratio, the accuracy kept increasing (with a slower pace), since the sampled strings more accurately represented the distribution of grams in the entire data set.

Fig. 11(b) shows how changing the number of queries in the workload W_1 affected the estimation accuracy for the three data sets, when the sampling ratio was 4% ($40,000$ strings). When there were $2,000$ queries, the accuracy was 80.3% for the data set of actor names. It increased to 83.9% when the number of queries increased to $50,000$. Similar results were seen for the other two data sets. The reason for the accuracy improvement is that our estimation method uses frequencies of gram lists from the queries as a representative distribution. The more queries we have, the more representative these lists are for the real distribution.

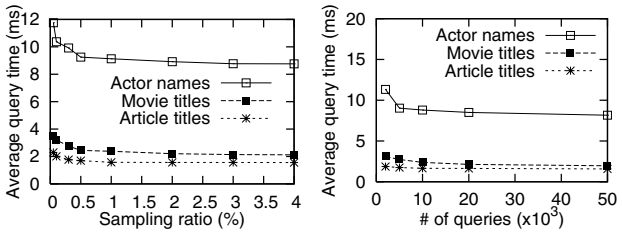


(a) Sampling ratio. (b) # of workload queries.

Figure 11: Accuracy of estimating gram benefits.

Query Performance: We evaluated the quality of a generated gram dictionary for each data set by running a workload W_2 of $1,000$ queries on the data set, using the generated dictionary. Fig. 12(a) shows how the average running time of queries changed if we increased the sampling ratio to generate the gram dictionary, for the three data sets. The analysis is similar to that of Fig. 11(a). As we increased the sampling ratio, since our method gave more accurate estimations, we added high-quality grams to the dictionary, which reduced the running time of queries.

Fig. 12(b) shows how the average query time changed if we increased the number of queries in the workload W_1 to generate the gram dictionary. The analysis is similar to that of Fig. 11(b). As we increased the number of queries in the workload, the frequencies of lists used by our method became a better representation of the real distribution, making our method generate better grams. Thus the running time of queries reduced.



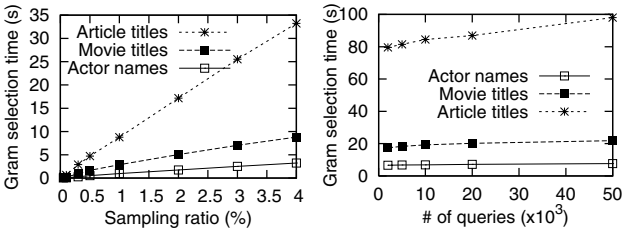
(a) Sampling ratio. (b) # of workload queries.

Figure 12: Average query time.

Dictionary-Construction Time: Fig. 13(a) shows how the total time to construct a gram dictionary changed when we increased the sampling ratio. This construction time grew linearly as the sampling ratio increased. For instance, for the data set of movie titles, when we sampled 0.5% of the strings (i.e., $5,000$ sampled strings), the total construction time was only 1.65 seconds. When we sampled 4% of the strings, the total construction time was 8.81 seconds. For the data set of movie titles, when we sampled 0.5% of the strings, the total construction time was only 4.72 seconds. When we sampled 4% of the strings, the total construction time was 33.23 seconds. The reason for this growth is that, as we sampled more strings from each data set, it took more time to incrementally maintain the complete list and local

list for each gram. Thus the time in each step in the algorithm increased, causing more time to construct the final dictionary. The dictionary-construction time for the article data set was the longest among the three data sets, since the strings in this data set were the longest. It shows that the construction time is also affected by the string lengths, which affect the lengths of gram lists.

Fig. 13(b) shows how the construction time changed when we increased the number of queries in the workload W_1, when the sampling ratio was fixed to be 4%. As this number increased, the construction time also increased. The reason for this growth is that, as we had more queries in the workload W_1, it took more time to incrementally maintain the trie built for the queries. Notice that since the sampled strings were fixed, increasing the number of queries in W_1 did not increase the number of grams that needed to be evaluated. Thus the linear growth rate was smaller than that of the sampling ratio.



(a) Sampling ratio. (b) # of workload queries.

Figure 13: Dictionary-construction time.

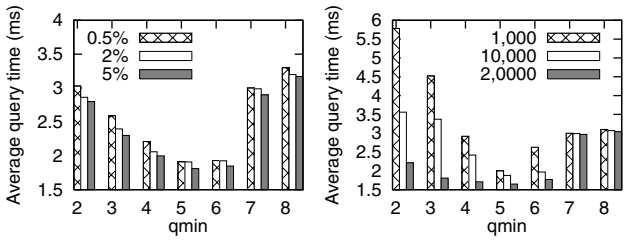
6.3 Choosing q_{min}

Algorithm GramGen assumes a given q_{min} parameter. We evaluated how different q_{min} values affect the quality of the final gram dictionary. For the data set of article titles, we first fixed a workload W_1 of 3,000 queries, and used different sampling ratios for the data strings. We used different q_{min} values. The results of average query performance using the final gram dictionary are shown in Fig. 14(a). It shows that, for a fixed sampling ratio, as we increased the q_{min} value, the average query time first decreased. After q_{min} was greater than 5, the query time started increasing. That is, the quality of the final gram dictionary first increased, then started decreasing. This minimum point did not change even when we varied the sampling ratio. We also did the experiments by fixing the sampling ratio to 2%, and changing the number of queries in workload W_1. The results in Fig. 14(b) show that we have the same minimum point $q_{min} = 5$ independently from the number of workload queries. We had the same observation for the other two data sets.

These experimental results suggested the following way to choose an optimal q_{min} automatically. We fixed a sampling ratio and a number of queries in workload W_1. We ran the GramGen algorithm for different q_{min} values, and compared the average query performance for the generated gram dictionary for each q_{min} value. We chose the q_{min} value that can give us the smallest query running time.

6.4 Comparison with Algorithm Prune

In [18] we proposed a heuristic-based algorithm, called Prune, for generating a gram dictionary for a collection of strings. This algorithm requires a few important parameters, and does not consider a workload of queries. We used

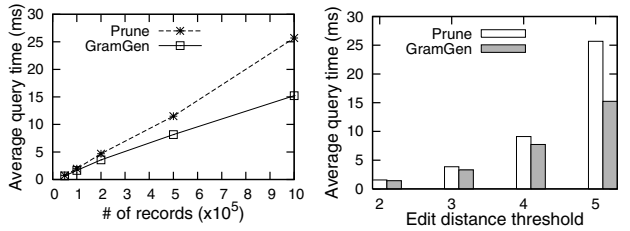


(a) Sampling ratio. (b) # of workload queries.

Figure 14: Effect of q_{min} on query time (article titles).

this algorithm, and ran it on a data set of article titles. We manually tuned those parameters in order to achieve the best performance, and the optimal setting was: $q_{min} = 5$, $q_{max} = 7$, frequency threshold $T = 2,000$, and LargeFirst policy. Then we ran our algorithm GramGen to generate a gram dictionary, with the following setting: 1% sampling ratio, 2,000 workload queries, $q_{min} = 5$ (automatically generated). We compared the query performance using these two different dictionaries.

Fig. 15(a) shows how the average query time using the two dictionaries changed as the size of the data set increased. Fig. 15(b) shows the query performance results for queries with different edit distance thresholds, when the data set had 1 million strings. The figures show that the gram dictionary generated by our new algorithm had a higher quality than that by the previous algorithm.



(a) Overall query performance. (b) Queries with different edit distance thresholds.

Figure 15: Comparison of dictionaries generated by algorithms GramGen and Prune (article titles).

7. RELATED WORK AND CONCLUSIONS

In the literature the term *approximate string matching* also means the problem of finding within a long text string those substrings that are similar to a given query pattern. See [20] for an excellent survey on research related to this problem. In this paper, we use this term to refer to the problem of finding from a collection of strings those similar to a given query string.

Many algorithms have been developed for the problem of approximate string joins based on various similarity functions [1, 2, 4, 6, 8, 22], especially in the context of record linkage [14]. Some of them are proposed in the context of relational DBMS systems. The VGRAM [18] technique has been shown to improve those algorithms based on edit distance. Several recent papers have mainly focused on approximate *selection* (or search) queries [9, 17]. In this paper we mainly focused on selection queries due to its importance to many applications. Although our discussions mainly as-

sumed an index of inverted lists of grams, they are also valid for other similar index structures, since we mainly focused on how the bags of string ids for grams change when adding a new gram to the dictionary. For example, if we use a hash table to manage these bags, as used in [1], the discussions are still valid after minor changes according to the algorithm.

There are recent studies on the problem of estimating the selectivity of SQL LIKE substring queries [5, 11, 15], and approximate string queries [12, 16, 19]. Some of the methods in these techniques can be adopted to solve the estimation problems in generating a gram dictionary. Notice that our estimation subproblems are more complicated due to the fact that the overall performance of queries is affected by several factors, such as the lists of grams, the method to compute the lower bound of common grams between similar strings, and the number of candidates satisfying the lower bound. It also depends on the specific algorithm used to access the inverted lists of grams.

Another related study is [10], which proposed a gram-selection technique for indexing text data under space constraints. They mainly considered SQL LIKE queries using fixed-length grams. Our work differs from theirs since we focused on approximate string queries using variable-length grams. [3] studied the problem of selecting *word* grams for improving classification performance of email messages. Other related studies include [7, 13, 21].

Conclusions: In this paper, we studied a fundamental problem in answering approximate queries on a collection of strings in the context of the VGRAM technique: what is the relationship between a predefined gram dictionary and the performance of queries? We proposed a dynamic programming algorithm for computing a tight lower bound on the number of common grams shared by two similar strings in order to improve query performance. We analyzed how adding a gram to an existing dictionary affects the index structure of the string collection, and the performance of queries. We proposed an efficient algorithm for automatically generating a high-quality gram dictionary. Our extensive experiments on real data sets show that these techniques can greatly improve approximate string queries.

Acknowledgements: Xiaochun Yang and Bin Wang were partially supported by the Program for New Century Excellent Talents in Universities (NCET-06-0290), NSF China grant 60503036, and the Fok Ying Tong Education Foundation Award 104027. The work was done when they visited UC Irvine in 2007. Chen Li was partially supported by the NSF CAREER Award No. IIS-0238586, NSF Award IIS-0742960, and a Google Research Award.

8. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all-pairs similarity search. In *WWW Conference*, 2007.

[3] V. R. Carvalho and W. W. Cohen. Improving "email speech acts" analysis via n-gram selection. In *Analyzing Conversations in Text and Speech (ACTS) Workshop at HLT-NAACL*, pages 35–41, 2006.

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[5] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE*, pages 227–238, 2004.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[7] P. Fogla and W. Lee. q-Gram matching using tree models. *IEEE TKDE*, 18(4), 2006.

[8] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[9] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Set similarity selection queries at interactive speeds. In *ICDE*, 2008.

[10] B. Hore, H. Hacigumus, and B. Iyer. Indexing text data under space constraints. In *Proceedings of CIKM*, pages 198–207, 2004.

[11] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *PODS*, pages 249–260, 1999.

[12] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.

[13] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-Gram/2L: a space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[14] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Tutorial*, pages 802–803, 2005.

[15] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD*, pages 282–293, 1996.

[16] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[17] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[18] C. Li, B. Wang, and X. Yang. VGRAM: improving performance of approximate queries on string collections using variable length grams. In *VLDB*, pages 303–314, 2007.

[19] A. Mazeika, M. H. Bóhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Transactions on Database Systems*, 32(2), 2007.

[20] G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surveys*, 33(1):31–88, 2001.

[21] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, 2003.

[22] S. Sarawagi and A. Kirpai. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754, 2004.