VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams

Chen Li
University of California, Irvine
CA 92697, USA
chenli@ics.uci.edu

Bin Wang
Northeastern University
Liaoning 110004, China
binwang@mail.neu.edu.cn

Xiaochun Yang
Northeastern University
Liaoning 110004, China
yangxc@mail.neu.edu.cn

ABSTRACT

Many applications need to solve the following problem of approximate string matching: from a collection of strings, how to find those similar to a given string, or the strings in another (possibly the same) collection of strings? Many algorithms are developed using fixed-length grams, which are substrings of a string used as signatures to identify similar strings. In this paper we develop a novel technique, called VGRAM, to improve the performance of these algorithms. Its main idea is to judiciously choose high-quality grams of variable lengths from a collection of strings to support queries on the collection. We give a full specification of this technique, including how to select high-quality grams from the collection, how to generate variable-length grams for a string based on the preselected grams, and what is the relationship between the similarity of the gram sets of two strings and their edit distance. A primary advantage of the technique is that it can be adopted by a plethora of approximate string algorithms without the need to modify them substantially. We present our extensive experiments on real data sets to evaluate the technique, and show the significant performance improvements on three existing algorithms.

1. INTRODUCTION

Motivation: As textual information is prevalent in information systems, many applications have an increasing need to support approximate string queries on data collections. Such queries ask for, from a given collection of strings, those strings that are similar to a given string, or those from another (possibly the same) collection of strings. This collection could be the values from a column in a table, a set of words in a dictionary, or a set of predefined entity names such as company names and addresses. The following are several examples.

Data Cleaning: Information from multiple data sources can have various inconsistencies. The same real-world entity can be represented in slightly different formats, such as "`PO Box 23, Main St.`" and "`P.O. Box 23, Main St`". There could be even errors in the data due to the process it was collected. For these reasons, data cleaning often needs to find from a collection of entities those similar to a given entity, or all similar pairs of entities from two collections.

Query Relaxation: When a user issues an SQL query to a DBMS, her input values might not match those interesting entries exactly, due to possible errors in the query, inconsistencies in the data, or her limited knowledge about the data. By supporting query relaxation, we can return the entries in the database (e.g., "`Steven Spielburg`)" that are similar to a value in the query (e.g., "`Steve Spielberg`"), so that the user can find records that could be of interests to her.

Spellchecking: Given an input document, a spellchecker finds potential candidates for a possibly mistyped word by searching in its dictionary those words similar to the word.

There is a large amount of work on supporting such queries efficiently, such as [1, 2, 3, 10, 23, 24, 25]. These techniques assume a given similarity function to quantify the closeness between two strings. Different string-similarity functions have been proposed [21], such as edit distance [18], Jaro metric [13], and token-based cosine metric [2, 6]. Among them, edit distance is a commonly used function due to its applicability in many applications. Many algorithms have focused on approximate string queries using this function. The idea of *grams* has been widely used in these algorithms. A gram is a substring of a string that can be used as a signature of the string. These algorithms rely on index structures based on grams and the corresponding searching algorithms to find those strings similar to a string.

Dilemma of Choosing Gram Length: The gram length can greatly affect the performance of these algorithms. As an example, Fig. 1 shows the distributions of the gram frequencies for different gram lengths for a DBLP data set of 276,699 article titles. (Details of the data are explained in Section 7.) The x-axis is the rank of a gram based on its frequency, and the y-axis is the frequency of the gram. The distributions show that there are some grams that are very popular in the data set. For instance, the 5-gram `ation` appeared 113,931 times! Other popular 5-grams include `tions`, `ystem`, `ting`, and `catio`. As a consequence, a string can have a high chance to have a popular gram. Similar distributions were observed in other data sets as well.

Algorithms based on fixed-length grams have a dilemma in deciding the length of grams. As an illustrative example, consider algorithms (e.g., [23, 24, 25]) that are based on an inverted-list index structure to find similar strings. These algorithms use various filtering techniques to prune strings.

VLDB '07, September 23-28, 2007, Vienna, Austria.

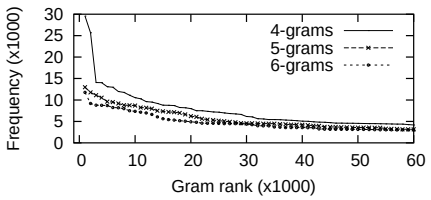


Figure 1: Gram frequencies in DBLP titles (not all grams are shown).

(More details are given in Section 6.) One important filter is called "count filter," which is using the following fact. If the edit distance between two strings is within a threshold, then they should share enough common grams. A lower bound of the number of common grams depends on the length of the grams and the edit distance threshold. If we increase the gram length, there could be fewer strings sharing a gram, causing the inverted lists to be shorter. Thus it may decrease the time to merge the inverted lists. On the other hand, we will have a lower threshold on the number of common grams shared by similar strings, causing a less selective count filter to eliminate dissimilar string pairs. The number of false positives after merging the lists will increase, causing more time to compute their real edit distances (a costly computation) in order to verify if they are in the answer to the query. The dilemma also exists in spirit in other algorithms as well. See Section 6.3 for another example.

Our Contributions: The dilemma is due to the "one-for-all" principle used in these algorithms. Based on this observation, in this paper we develop a novel technique, called VGRAM, to improve the performance of these algorithms. Its main idea is to judiciously choose high-quality grams of variable lengths from a collection of strings to support queries on the collection. At a high level, VGRAM can be viewed as an index structure associated with a collection of strings, on which we want to support approximate queries. An overview of the technique is the following.

- We analyze the frequencies of variable-length grams in the strings, and select a set of grams, called *gram dictionary*, such that each selected gram in the dictionary is not too frequent in the strings.
- For a string, we generate a set of grams of variable lengths using the gram dictionary.
- We can show that if two strings are within edit distance k, then their sets of grams also have enough similarity, which is related to k. This set similarity can be used to improve the performance of existing algorithms.

We study several challenges that arise naturally when using this simple but powerful idea. (1) How to generate variable-length grams for a string? For the case of using fixed-length grams, it is straightforward to generate grams for strings, but the answer becomes not obvious in our case. In Section 3 we show how to generate such grams using a precomputed gram dictionary. (2) How to construct a high-quality gram dictionary? The selected grams can greatly affect the performance of queries. In Section 4 we develop an efficient algorithm for generating such a gram dictionary based on an analysis of gram frequencies.

(3) What is the relationship between the similarity of the gram sets of two strings and their string similarity? The relationship is no longer obvious as compared to the fixed-length-gram case, since the strings can generate grams with different lengths. In Section 5 we show that such a relationship still exists, and the analysis is technically very nontrivial. (4) How to adopt VGRAM in existing algorithms? A primary advantage of the technique is that it can be used by a plethora of approximate string algorithms without substantially modifying the algorithms. In Section 6 we use three existing algorithms in the literature to show how to adopt the technique. It is worth mentioning that when adopting VGRAM in these algorithms, it guarantees that it does not miss true answers, i.e., there are no false negatives.

We have conducted extensive experiments to evaluate the technique. The results, as reported in Section 7, show that the technique can be adopted easily by these algorithms and achieve a significant improvement on their performance. The technique can also greatly reduce the index size of those algorithms based on inverted lists, even after considering the small index overhead introduced by the technique. In addition, the index structure used by the technique can be easily maintained dynamically, and be utilized for algorithms inside relational DBMS, as discussed in Section 8. The technique is extendable to variants of the edit distance function.

1.1 Related Work

In the literature "approximate string matching" also refers to the problem of finding a pattern string approximately in a text. There have been many studies on this problem. See [19] for an excellent survey. The problem studied in this paper is different: searching in a *collection of strings* those similar to a single query string ("selection") or those similar to another collection of strings ("join"). In this paper we use "approximate string matching" to refer to our problem.

Many algorithms (e.g., [23, 24, 25]) for supporting approximate string queries use an inverted-list index structure of the grams in strings, especially in the context of record linkage [16]. Various filtering techniques are proposed to improve their performance. These techniques can be adopted with modifications inside a relational DBMS to support approximate string queries using SQL [3, 10]. Motivated by the need to do fuzzy queries, several algorithms have been proposed to support set-similarity joins [5, 22]. These algorithms find, given two collections of sets, those pairs of sets that share enough common elements. These algorithms can be used to answer approximate queries due to the relationship between string similarity and the similarity of their gram sets. We will give a detailed description of some of these algorithms in Section 6. Our VGRAM technique can be used by these algorithms to improve their performance.

The idea of using grams of variable lengths has been used in other applications such as speech recognition [20], information retrieval [7, 9], and artificial intelligence [11]. The same idea has also been considered in the database literature for the problem of substring selectivity estimation for the SQL `LIKE` operator [4, 12, 17]. For instance, [4] proposed the concept of "shortest identifying substring," whose selectivity is very similar to that of its original string. [12, 17] studied how to choose, in a suffix tree, a set of strings whose frequency (or "count") is above a predefined threshold due to storage constraint. It is based on the assumption that low-frequency substrings are relatively less important for substring selectivity estimation. Compared to these earlier studies, ours is the first one using this idea to answer

approximate string queries on string collections. Since our addressed problem is different, our approach to selecting variable-length grams is also different from previous ones. In addition, our results on analyzing similarity between the gram sets of two similar strings and adopting VGRAM in existing algorithms are also novel.

Kim et al. [14] proposed a technique called "n-Gram/2L" to improve space and time efficiency for inverted index structures. Fogla and Lee [8] studied approximate substring matching and proposed a method of storing grams as a trie without losing any information. Compared to these two studies, our work focuses on approximate string queries on string collections and the corresponding filtering effect of variable-length grams. Another related work is a recent study in [2] on approximate string joins using functions such as cosine similarity.

2. PRELIMINARIES

Let Σ be an alphabet. For a string s of the characters in Σ, we use "$|s|$" to denote the length of s, "$s[i]$" 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.

Q-Grams: Given a string s and a positive integer q, a *positional q-gram* of s is a pair (i, g), where g is the q-gram of s starting at the i-th character, 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 string s. There are $|s| - q + 1$ positional q-grams in $G(s, q)$. For instance, suppose $q = 3$, and $s =$ `university`, then $G(s, q) = \{(1,$ `uni`$), (2,$ `niv`$), (3,$ `ive`$), (4,$ `ver`$), (5,$ `ers`$), (6,$ `rsi`$), (7,$ `sit`$), (8,$ `ity`$)\}$. A slightly different definition of positional gram set was introduced in [10]. According to this definition, we introduce two characters α and β that do not belong to Σ, and extend a string by prefixing $q-1$ copies of α and suffixing $q - 1$ copies of β. We use a sliding window of size q on the new string to generate positional q-grams. All the results in this paper carry over to this definition as well, with necessary minor modifications.

Approximate String Queries: The *edit distance* (a.k.a. Levenshtein distance) between two strings s_1 and s_2 is the minimum number of edit operations of single characters that are needed to transform s_1 to s_2. Edit operations include insertion, deletion, and substitution. We denote the edit distance between s_1 and s_2 as $ed(s_1, s_2)$. For example, $ed($"`Steven Spielburg`", "`Steve Spielberg`"$) = 2$. We consider two types of approximate string queries on a given collection of strings S (possibly with duplicates). (1) *Approximate-string selections*: for a query string Q, find all the strings s in S such that $ed(Q, s) \leq k$, where k is a given distance threshold. (2) *Approximate-string joins*: given a collection S' (possibly the same as S), find string pairs in $S \times S'$ whose edit distance is not greater than a threshold k.

3. VARIABLE-LENGTH GRAMS

Let S be a collection of strings, on which we want to use VGRAM. The technique uses two integer parameters, q_{min} and q_{max}, such that $q_{min} < q_{max}$, and we consider grams of lengths between q_{min} and q_{max}. In this section we study how to convert a string to a set of variable-length grams, by using a predefined set of grams, called a "gram dictionary," which is obtained from S. In Section 4 we will study how to construct such a gram dictionary from S.

3.1 Gram Dictionary

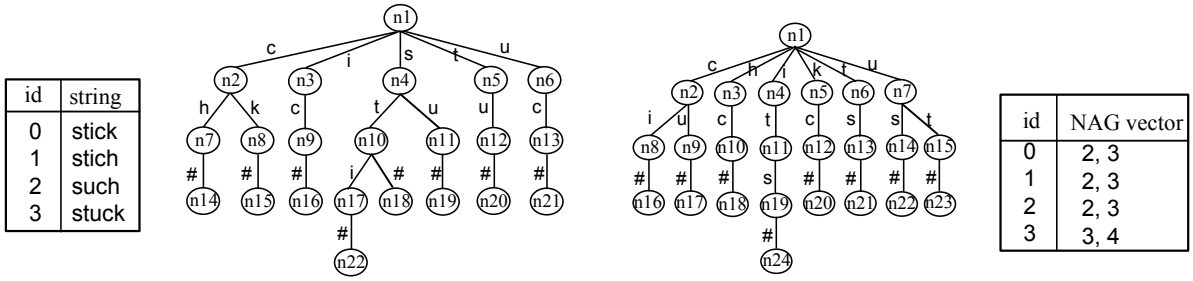
If a gram g_1 is a proper prefix of a gram g_2, we call g_1 a *prefix gram* of g_2, and g_2 an *extended gram* of g_1. For instance, the gram `uni` is a prefix gram of `univ`, while the latter is an extended gram of the former.

A *gram dictionary* is a set of grams $\mathcal{D}$ of lengths between q_{min} and q_{max}. Notice that the gram dictionary could be constructed independently of a collection of strings S, even though for performance reasons we tend to compute a gram dictionary by analyzing gram frequencies of the string collection. A gram dictionary $\mathcal{D}$ can be stored as a trie. The trie is a tree, and each edge is labeled with a character. To distinguish a gram from its extended grams, we preprocess the grams in $\mathcal{D}$ by adding to the end of each gram a special endmarker symbol [15] that does not belong to the alphabet Σ, e.g., `#`. A path from the root node to a leaf node corresponds to a gram in $\mathcal{D}$. (The endmarker symbol is not part of the gram.) We call this gram the *corresponding gram* of this leaf node. In addition, for each gram in $\mathcal{D}$, there is a corresponding root-to-leaf path on the trie. For example, Fig. 2(b) shows a trie for a gram dictionary of the four strings in Fig. 2(a), where $q_{min} = 2$ and $q_{max} = 3$. (Figs. 2(b)-(d) show a VGRAM index for the strings. The rest of the index will be explained in Section 5.) The dictionary includes the following grams: {`ch`, `ck`, `ic`, `sti`, `st`, `su`, `tu`, `uc`}. The path $n_1 \rightarrow n_4 \rightarrow n_{10} \rightarrow n_{17} \rightarrow n_{22}$ corresponds to the gram `sti`.

3.2 Generating Variable-Length Grams

For the case of using a fixed gram length q, we can easily generate the set of q-grams for a string by sliding a window of size q over the string from left to right. When using a gram dictionary $\mathcal{D}$ to generate a set of variable-length grams for a string s, we still use a window to slide over s, but the window size varies, depending on the string s and the grams in $\mathcal{D}$. Intuitively, at each step, we generate a gram for the longest substring (starting from the current position) that matches a gram in the dictionary. If no such gram exists in $\mathcal{D}$, we will generate a gram of length q_{min}. In addition, for a positional gram (a, g) whose corresponding substring $s[a, b]$ has been subsumed by the substring $s[a', b']$ of an earlier positional gram (a', g'), i.e., $a' \leq a < b \leq b'$, we ignore the positional gram (a, g).

Formally, we decompose string s to its set of positional grams using the algorithm in Fig. 3. We start by setting the current position to the first character of s. In each step, from the current position, we search for the longest substring of s that appears in the gram dictionary $\mathcal{D}$ using the trie. If we cannot find such a substring, we consider the substring of length q_{min} starting from this position. In either case, we check if this substring is a proper substring of one of the already-produced substrings (considering their positional information in s). If so, we do not produce a positional gram for this new substring, since it has already been subsumed by an earlier positional gram. Otherwise, we produce a positional gram for this substring. We move the current position to the right by one character. We repeat the step above until the position is greater than $|s| - q_{min} + 1$. The generated set of positional grams for a string s is denoted by $VG(s, \mathcal{D}, q_{min}, q_{max})$, or simply $VG(s)$ if the other parameters are clear in the context.



(a) strings (b) Gram dictionary as a trie (c) Reversed-gram trie (d) NAG vectors

Figure 2: A VGRAM index for strings.

Algorithm: VGEN
Input: Gram dictionary $\mathcal{D}$, string s, bounds q_{min}, q_{max}
Output: a set of positional grams for s

(1) position $p = 1$; VG = empty set;
(2) WHILE ($p \leq |s| - q_{min} + 1$) {
(3) Find a longest gram in $\mathcal{D}$ using the trie to match a substring t of s starting at position p;
(4) IF (t is not found) $t = s[p, p + q_{min} - 1]$;
(5) IF (positional gram (p, t) is not subsumed by any positional gram in VG)
(6) Insert (p, t) to VG;
(7) $p = p + 1$;
}
(8) RETURN VG;

Figure 3: Decomposing a string to positional grams of variable lengths using a gram dictionary.

For example, consider a string s=`universal` and a gram dictionary $\mathcal{D}$ = {`ni`, `ivr`, `sal`, `uni`, `vers`}. Let q_{min} be 2 and q_{max} be 4. By setting $p = 1$ and $G = \{\}$, the algorithm starts at the first character `u`. The longest substring starting at `u` that appears in $\mathcal{D}$ is `uni`. Thus the algorithm produces a positional gram $(1, \texttt{uni})$ and inserts it to VG. Then the algorithm moves to the next character `n`. Starting from this character, the longest substring that appears in $\mathcal{D}$ is `ni`. However, since this candidate positional gram (2, `ni`) is subsumed by the previous one, the algorithm does not insert it into VG. The algorithm moves to the next character `i`. There is no substring starting at this character that matches a gram in $\mathcal{D}$, so the algorithm produces a positional gram (3, `iv`) of length $q_{min} = 2$. Since it is not subsumed by any positional gram in VG, the algorithm inserts it to VG. The algorithm repeats until the position is at the $(|s| - q_{min} + 2)$-nd character, which is the character `l`. The generated positional gram set is VG = {$(1, \texttt{uni})$, $(3, \texttt{iv})$, $(4, \texttt{vers})$, $(7, \texttt{sal})$}.

4. CONSTRUCTING GRAM DICTIONARY

In this section we study, for a given collection S of strings, how to decide a high-quality gram dictionary. We assume the two length bounds q_{min} and q_{max} are given, and later we will discuss how to choose these two parameters. We develop an efficient two-step algorithm to achieve the goal. In the first step, we analyze the frequencies of q-grams for the strings, where q is within q_{min} and q_{max}. In the second step, we select grams with a small frequency.

4.1 Step 1: Collecting Gram Frequencies

One naive way to collect the frequencies is the following. For each string s in S, for each q between q_{min} and q_{max}, we generate all its q-grams of s. For each q-gram, we count its frequency. This approach is computationally expensive, since it generates too many grams with their frequencies. To solve this problem, our algorithm uses a trie (called "frequency trie") to collect gram frequencies.[1] The algorithm avoids generating all the grams for the strings based on the following observation. Given a string s, for each integer q in $[q_{min}, q_{max} - 1]$, for each positional q-gram (p, g), there is a positional gram (p, g') for its extended q_{max}-gram g'. For example, consider a string `university`, and its positional gram (2, `niv`). Let $q_{min} = 2$ and $q_{max} = 4$. There is also a positional 4-gram (2, `nive`) starting at the same position. Therefore, we can generate q_{max}-grams for the strings to do the counting on the trie without generating the shorter grams, except for those grams at the end of a string.

Based on this observation, the algorithm collects gram frequencies as follows. Each node n in the frequency trie has a frequency value $n.freq$. We initialize the frequency trie to be empty. For each string s, we first generate all its positional q_{max}-grams. For each of them, we locate the corresponding leaf node, or insert it to the trie if the gram does not exist (the frequency for this leaf node is initialized to 0). For each node on the path from the root to this leaf node, including this leaf node, we increment its frequency by 1. At each q-th node ($q_{min} \leq q < q_{max}$) on the path, we create a leaf node by appending an edge with the special endmarker symbol `#`, if this new leaf node does not exist. This new leaf node represents the fact that the q_{max}-gram has a prefix gram of length q that ends at this new leaf node. Notice that for the *leaf node* n' of each such prefix gram, we do *not* increment the frequency of n' by 1, since its parent node already did the counting.

We deal with those characters at the end of the string separately, since they do not produce positional q_{max}-grams. In particular, for each position $p = |s| - q_{max} + 2, \ldots, |s| - q_{min} + 1$ of the string, we generate a positional gram of length $|s| - p + 1$, and repeat the same procedure on the trie as described above. For instance, if $q_{min} = 2$ and $q_{max} = 4$, for the string s = `university`, we need to generate the following positional grams (8, `ity`) and (9, `ty`) of length between 2 and 3, and do the counting on the trie.

After step 1, we have constructed a trie with a frequency in each node. For example, Fig. 4 shows the frequency trie for the strings in Fig. 2(a). For instance, the frequency number

[1]The data structure of a trie with string frequencies is also used in earlier studies [12, 17].

"2" at node n_{43} means that the gram `stic` occurred 2 times in the strings. The frequency number "3" at node n_{10} means that the gram `st` appears 3 times.

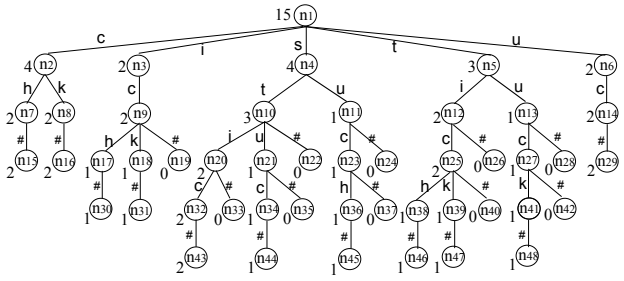


Figure 4: A gram-frequency trie.

4.2 Step 2: Selecting High-Quality Grams

In this step, we judiciously prune the frequency trie and use the remaining grams to form a gram dictionary. The intuition of the pruning process is the following. (1) Keep short grams if possible: If a gram g has a low frequency, we eliminate from the trie all the extended grams of g. (2) If a gram is very frequent, keep some of its extended grams. As a simple example, consider a gram `ab`. If its frequency is low, then we will keep it in the gram dictionary. If its frequency is very high, we will consider keeping this gram and its extended grams, such as `aba`, `abb`, `abc`, etc. The goal is that, by keeping these extended grams in the dictionary, the number of strings that generate an `ab` gram by the VGEN algorithm could become smaller, since they may generate the extended grams instead of `ab`.

```
FUNCTION Prune(Node n, Threshold T)
1.  IF (each child of n is not a leaf node) {
        // the root→ n path is shorter than q_min
2.      FOR (each child c of n);
3.          CALL Prune(c, T); // recursive call
4.      RETURN;
5.  }
    // a gram corresponds to the leaf-node child of n
6.  L = the (only) leaf-node child of n;
7.  IF (n.freq ≤ T) {
8.      Keep L, and remove other children of n;
9.      L.freq = n.freq;
10. }
11. ELSE {
12.     Select a maximal subset of children of n (excluding L),
           so that the summation of their freq values and
           L.freq is still not greater than T;
13.     Add the freq values of these children to
           that of L, and remove these children from n;
14.     FOR (each remaining child c of n excluding L)
15.         CALL Prune(c, T); // recursive call
16. }
```

Figure 5: Pruning a subtrie to select grams.

Formally, we choose a *frequency threshold*, denoted by T. We prune the trie by calling the function Prune shown in Fig. 5, by passing as the parameters the root of the frequency trie and the threshold T. At each step, we check if the current node n has a leaf-node child. (A leaf node has, from its parent, an edge labeled by the endmarker symbol `#`.) If it does not have any leaf-node child, then the path from the root to this node corresponds to a gram shorter than q_{min}, so we recursively call the function for each of its children.

If this node has a leaf-node child L, then there is a gram g corresponding to L. We consider the frequency of node n, i.e., $n.freq$. If it is already not greater than T, then we keep this gram. In addition, we remove the children of n except L, and assign the frequency of n to L. After this pruning step, node n has a single leaf-node child L.

If $n.freq > T$, we want to keep some of its extended grams of g, hoping the new frequency at node L could be not greater than T. The algorithm selects a maximal subset of n's children (excluding L), so that the summation of the frequencies of these nodes and $L.freq$ is still not greater than T. (Intuitively, the node L is "absorbing" the frequencies of the selected children.) For the remaining children (excluding L), we recursively call the function on each of them to prune the subtree. The following are three possible *pruning policies* to be used to select a maximal subset of children to remove (line 12).

- SmallFirst: Choose children with the *smallest* frequencies.
- LargeFirst: Choose children with the *largest* frequencies.
- Random: Randomly select children so that the new $L.freq$ after absorbing the frequencies of the selected children is not greater than T.

For instance, in the frequency trie in Fig. 4, assume threshold $T = 2$. As the algorithm traverses the trie top down, it reaches n_{10}, whose frequency 3 is greater than T. The node has a single leaf child node, n_{22}, whose frequency is 0, meaning there is no substring of `st` in the data set without an extended gram of `st`. The node n_{10} has two other children, n_{20} with a frequency 2 and n_{21} with a frequency 1. By using the SmallFirst policy, the algorithm chooses n_{21} to prune, and updates the frequency of n_{22} to 1. By using LargeFirst, the algorithm chooses n_{20} to prune, and updates the frequency of n_{22} to 2. By using Random, the algorithm randomly chooses one of these two children to prune, and adds the corresponding frequency to that of n_{22}. Fig. 2(b) shows the final trie using the Random policy.

Remarks: (1) Notice that it is still possible for this algorithm to select grams with a frequency greater than T. This threshold is mainly used to decide what grams to prune. The frequencies of the selected grams also depend on the data set itself. For instance, consider the case where we had a collection of N identical strings of `abc`. No matter what the threshold T is, each selected gram must have the same frequency, N. When we adopt VGRAM in existing algorithms, our technique does guarantee no false negatives.

(2) Deciding q_{min} and q_{max}: We assumed parameters q_{min} and q_{max} are given before constructing the trie to decide a gram dictionary. If these values are not given, we can initially choose a relatively small q_{min} and large q_{max}, and run the algorithm above to decide a gram dictionary. After that, we can change q_{min} and q_{max} to the length of the shortest and the longest grams in the dictionary, respectively.

5. SIMILARITY OF GRAM SETS

We now study the relationship between the similarity of two strings and the similarity of their gram sets generated using the same gram dictionary.

5.1 Fixed-Length Grams

We first revisit the relationship between the similarity of the sets of *fixed-length* grams of two strings and their edit distance. From a string's perspective, k edit operations can in worst case "touch" $k \cdot q$ grams of the string. As a consequence, if two strings s_1 and s_2 have an edit distance not greater than k, then their sets of positional grams $G(s_1, q)$ and $G(s_2, q)$ should share at least the following number of common grams (ignoring positional information):[2]

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

Arasu et al. [1] showed the following. For each string, we represent its set of grams of length q as a bit vector (ignoring positional information). For two strings within an edit distance k, the hamming distance of their corresponding bit vectors is not greater than the following string-independent *hamming-distance bound*.

$$\mathcal{B}_h(s_1, s_2, q, k) = 2 \cdot k \cdot q. \quad (2)$$

5.2 Effect of Edit Operations on Grams

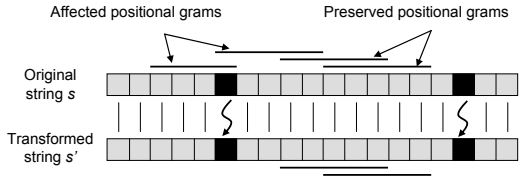


Figure 6: Preserved positional grams versus affected positional grams.

Now let us consider variable-length grams. For two strings s and s', let $VG(s)$ and $VG(s')$ be their positional gram sets generated based on a gram dictionary $\mathcal{D}$ with two gram-length parameters q_{min} and q_{max}. Fig. 6 shows the effect of edit operations on the string s. For each character $s[i]$ in s that is aligned with a character $s'[j]$ in s', if there is positional gram (i, g) in $VG(s)$, and there is a positional gram (j, g) in $VG(s')$, such that $|i - j| \leq ed(s, s')$, we call (i, g) a *preserved positional gram*. Other positional grams in $VG(s)$ are called *affected positional grams*. Our goal is to compute the number of preserved positional grams in $VG(s)$ after k edit operations, even if we do not know exactly what the transformed string s' is. The affected positional grams due to an edit operation depend on the position of the gram and the edit operation. Next we will analyze the effect of an edit operation on the positional grams.

Consider a deletion operation on the i-th character of s, and its effect on each positional gram (p, g) that belongs to one of the following four categories, as illustrated in Fig. 7.

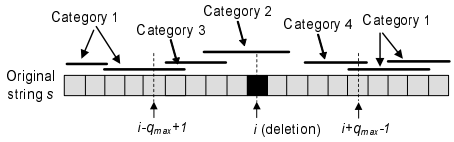


Figure 7: Four categories of positional grams based on whether they can be affected due a deletion operation on the i-th character.

[2]This formula assumes we do not extend a string by prefixing and suffixing special characters. A slightly different formula can be used when we do the string extension [10].

Category 1: Consider the following window $[a, b]$ including the character $s[i]$, where $a = max\{1, i - q_{max} + 1\}$, and $b = min\{|s|, i + q_{max} - 1\}$. If the positional gram (p, g) is not contained in this window, i.e., $p < i - q_{max} + 1$ or $p + |g| - 1 > i + q_{max} - 1$, this deletion does not affect the positional gram.

Category 2: If the positional gram overlaps with this character, i.e., $p \leq i \leq p + |g| - 1$, then it could be affected by this deletion.

Category 3: Consider a positional gram (p, g) on the left of the i-th character, and contained in the window $[a, i - 1]$, i.e., $i - q_{max} + 1 \leq p < p + |g| - 1 \leq i - 1$. These positional grams could be potentially affected due to this deletion. To find out which positional grams could be affected, we do the following. Consider the position $j = a$, and the substring $s[j, i - 1]$. If this substring is a *prefix* of a gram g' in the dictionary $\mathcal{D}$, then all the positional grams contained in the interval $[j, i - 1]$ could be potentially affected due to the deletion. The reason is that these positional grams could be subsumed by a longer substring (see Line 5 in Fig. 3). We mark these positional grams "potentially affected." If no extended gram g' exists in the dictionary, this deletion does not affect this positional gram (p, g). We increment the position j by one, and repeat the checking above, until we find such a gram g' in $\mathcal{D}$, or when $j = i - q_{min} + 1$.

Category 4: Symmetrically, consider a positional gram (p, g) on the right of the i-th character, and contained in the window $[i + 1, b]$, i.e., $i + 1 \leq p < p + |g| - 1 \leq i + q_{max} - 1$. These positional grams could be potentially affected due to this deletion. To find out which grams could be affected, we do the following. Consider the position $j = b$, and the substring $s[i + 1, j]$. If there is a gram g' in the dictionary such that g is a *suffix* of g', then all the positional grams contained in the interval $[i + 1, j]$ could be potentially affected due to the deletion, for the same reason described above. We mark these positional grams "potentially affected." If no extended gram g' exists in the dictionary, this deletion does not affect this positional gram (p, g). We decrement the position j by one, and repeat the checking above, until we find such a gram g' in $\mathcal{D}$, or when $j = i + q_{min} - 1$.

For instance, consider the example in Section 3.2, where we have a string s=`universal`, a gram dictionary $\mathcal{D} = \{$`ni`, `ivr`, `sal`, `uni`, `vers`$\}$, $q_{min} = 2$, and $q_{max} = 4$. The generated positional gram set is $VG(s) = \{(1, \texttt{uni}), (3, \texttt{iv}), (4, \texttt{vers}), (7, \texttt{sal})\}$. Consider a deletion on the 5-th character `e` in the string s. In the analysis of the four categories, we have $i = 5$, $i - q_{max} + 1 = 2$, so $a = 2$. In addition, $i + q_{max} - 1 = 8$, so $b = 8$. The positional gram $(1, \texttt{uni})$ belongs to category 1, since its starting position is before $a = 2$. Thus it will not be affected due to this deletion. $(7, \texttt{sal})$ also belongs to category 1, since its end position is after 8, and it will not be affected due to this deletion. Category 2 includes a positional gram, $(4, \texttt{vers})$, which could be affected by this deletion. Category 3 includes a single positional gram, $(3, \texttt{iv})$. Since there is a gram `ivr` in $\mathcal{D}$ that has the substring $s[3, 4]$ (which is `iv`) as a prefix, $(3, \texttt{iv})$ could be affected due to this deletion. In particular, after deleting the letter `e`, we could generate a new gram `ivr`, causing the gram `iv` to disappear. In conclusion, the positional grams $(3, \texttt{iv})$ and $(4, \texttt{vers})$ can be affected due to this deletion. In fact, the set of positional grams for the new string s' is: $VG(s') = \{(1, \texttt{uni}), (3, \texttt{ivr}), (5, \texttt{rs}), (6, \texttt{sal})\}$. Similarly, we can show that for a deletion on the 6-th character (`r`) on

the original string s, it can only affect the positional gram $(4, \texttt{vers})$. In particular, $(3, \texttt{iv})$ cannot be affected since there is no gram in $\mathcal{D}$ that has the substring `ive` as a prefix.

The analysis for a substitution operation is identical to the analysis above. The analysis for an insertion operation is almost the same, except that an insertion happens in a "gap," i.e., the place between two consecutive characters, before the first character, or after the last character. The analysis is valid with small modifications on the conditions to check which positional grams belong to which category.

Reversed-Gram Trie: For each character (for deletion and substitution) or gap (for insertion), we can easily decide the category of a positional gram using its starting position and gram length. To decide what positional grams in category 3 could be affected due to an operation, we need to check if the gram dictionary has a gram that has a given substring as a prefix. This test can be done efficiently using the trie for the dictionary. However, to decide what positional grams in category 4 could be affected, we need to check, for a given substring, whether the dictionary contains a gram that has this substring as a *suffix*. To support this test, we reverse each gram in $\mathcal{D}$, and build a trie using these reversed grams. This trie is called a *reserved-gram trie*, and is also part of the VGRAM index. Fig. 2(c) shows the reversed-gram trie for the dictionary stored in Fig. 2(b).

5.3 NAG Vectors

For each string s in the collection S, we want to know how many grams in $VG(s)$ can be affected by k edit operations. We precompute an upper bound of this number for each possible k value, and store the values (for different k values) in a vector for s, called the *vector of number of affected grams* ("NAG vector" for short) of string s, denoted by $NAG(s)$. The k-th number in the vector is denoted by $NAG(s, k)$. As we will see in Section 6, such upper bounds can be used to improve the performance of existing algorithms.

Ideally we want the values in $NAG(s)$ to be as tight as possible. For an integer $k > 0$, we can compute an upper bound based on the analysis in Section 5.2 as follows. For each of its $|s|$ characters and $|s| + 1$ gaps, we calculate the set of positional grams that could be affected due to an edit operation at this position (character or gap). For each character and gap, we calculate its number of potentially affected positional grams. For these $2|s| + 1$ numbers, we take the k largest numbers, and use their summation as $NAG(s, k)$. Fig. 2(d) shows the NAG vectors for the strings.

LEMMA 1. *For a string s_i, let $VG(s_i)$ and $NAG(s_i)$ be the corresponding set of variable-length positional grams and NAG vector of s_i, respectively. Suppose two strings s_1 and s_2 have $ed(s_1, s_2) \leq k$.*

- *The following is a lower bound on the number of common grams (ignoring positional information) between $VG(s_1)$ and $VG(s_2)$ (using the same gram dictionary).*

$$\mathcal{B}_{vc}(s_1, s_2, k) = max\big(|\,VG(s_1)| - NAG(s_1, k), \\ |\,VG(s_2)| - NAG(s_2, k)\big). \quad (3)$$

- *The following is an upper bound on the hamming distance between the bit vectors (ignoring positional information) corresponding to $VG(s_1)$ and $VG(s_2)$ (using the same gram dictionary):*

$$\mathcal{B}_{vh}(s_1, s_2, k) = NAG(s_1, k) + NAG(s_2, k). \quad (4)$$

This lemma shows that we can easily use NAG vectors to compute the similarity of the variable-gram sets of two similar strings.

6. ADOPTING VGRAM TECHNIQUE

In this section, we use three existing algorithms in the literature to show how to adopt VGRAM to improve their performance. Let S be a collection of strings. We have built a VGRAM index structure for S, which includes a gram dictionary $\mathcal{D}$ stored as a gram-dictionary trie, a reverse-gram trie, and a precomputed NAG vector $NAG(s)$ for each string s in S.

6.1 Algorithms Based on Inverted Lists

Algorithms such that those in [22, 23, 25, 26] could be implemented based on inverted lists of grams. For a string s in S, we generate its set of positional q-grams, for a constant q. For each of them, we insert the string id, together with the position of the gram in the string, to the inverted list of the gram. For an approximate selection query that has a string Q and an edit-distance threshold k, we want to find strings s in S such that $ed(s, Q) \leq k$. To answer the query, we use the q-grams of Q to search in their corresponding inverted lists, and merge these lists to find candidate strings. Several filtering techniques can be used: (1) *Length filtering*: $|s|$ and $|Q|$ differ by at most k. (2) *Position filtering*: the positions of each pair of common grams should differ by at most k. (3) *Count filtering*: the strings should share enough grams, and Equation 1 gives a lower bound of the number of common grams between the two strings.[3] For those strings that share enough pairs, we remove false positives by checking if their edit distance to Q is not greater than k. This algorithm is called MergeCount in [22]. An approximate string join of two string collections R and S can be implemented by calling MergeCount for each string in R on the inverted-list index of S. This implementation of approximate-string joins is called ProbeCount in [22].

To adopt VGRAM in these algorithms, we only need to make minor changes. (1) Instead of generating fixed-length q-grams, we call the VGEN algorithm to convert a string s to a set of positional variable-length grams $VG(s)$. (2) For two strings s_1 and s_2, instead of using the value in Equation 1 as a lower bound on the number of common grams, we use the new bound in Equation 3. In the equation, if s_i is in S, then $|\,VG(s_i)|$ and $NAG(s_i)$ are precomputed in the VGRAM index. If s_i is a string in a query, then $|\,VG(s_i)|$ and $NAG(s_i)$ are precomputed efficiently using the VGRAM index structure on the fly. The rest of these algorithms remains the same as before. As we will see in the experiments, adopting VGRAM can improve the performance of the algorithms and reduce their inverted-list size as well.

6.2 Algorithm: ProbeCluster

Sarawagi and Kirpai [22] proposed an algorithm called ProbeCluster to support efficient set-similarity joins [5]. Given a collection $\mathcal{S}$ of sets, this algorithm can find all pairs of sets from $\mathcal{S}$ whose number of common elements is at least a predefined threshold. This algorithm can be used to do a self approximate-string join of edit distance k on a collection of strings, after converting each string to a set of fixed-length

[3]String pairs with a zero or negative count bound need to be processed separately [10].

grams, and treating two string-position pairs as the same element if they use the same gram, and their positions differ by at most k. We use the bound $\mathcal{B}_c(s_1, s_2, q, k)$ in Equation 1 as the set-similarity threshold. (The algorithm still works even if different set pairs have different set-similarity thresholds.) When performing a self-join on the same collection of strings, the ProbeCluster algorithm improves the ProbeCount algorithm by using several optimizations. One optimization is that it scans the data only once, and conducts the join while building the inverted lists at the same time. Another optimization is to reduce the size of each inverted list by clustering sets (strings) with many common grams, and storing pointers to these clusters of strings instead of those individual strings. The algorithm constructs the clusters on-the-fly during the scan. For each record, it uses inverted lists of clusters to prune irrelevant clusters, before doing a finer-granularity search of string pairs.

To adopt VGRAM in ProbeCluster, we just need to make the same two minor modifications described above: (1) We call VGEN to convert a string to a set of variable-length grams; (2) We use Equation 3 instead of Equation 1 as a set-similarity threshold for the sets of two similar strings.

6.3 Algorithm: PartEnum

Arasu et al. [1] developed a novel algorithm, called PartEnum, to do set-similarity joins. The main idea of the algorithm is the following. Assume there are N elements corresponding to all possible grams. We view a subset of these N elements as a bit vector. If the hamming distance between two bit vectors is not greater than n, then after partitioning each vector to $n-1$ equi-size partitions, the two vectors should agree on at least one partition. The same observation can be extended by considering combinations of these partitions. Based on this idea, for the vector of each set, the algorithm first divides the vector into some partitions. For each partition, the algorithm further generates a set of signatures by using combinations of finer partitions. Using these signatures we can find pairs of bit vectors whose hamming distance is not greater than a given threshold. We can use this algorithm to do approximate-string joins with an edit distance threshold k, since the hamming distance of the bit vectors of the q-gram sets of two strings within edit distance k must be not greater than the upper bound in Equation 2. The dilemma of choosing gram length (see Section 1) also exists for this algorithm. As noticed by the authors, increasing the value of q can result in a larger (thus weaker) threshold in Equation 2. On the other hand, a smaller value of q means that the elements of the algorithm input are drawn from a smaller domain.

To adopt VGRAM in this algorithm, we notice from Equation 4 that different string pairs could have different upper bounds on their gram-based hamming distances. Suppose we want to do an approximate string join between two string collections, R and S, with an edit-distance threshold k. Assume we have a VGRAM index on R. For each string s in S, we compute its $VG(s)$ and $NAG(s, k)$ using the VGRAM index of R. (Such a step can be avoided when we do a self join of R.) Let $B_m(S)$ be the maximal value of these $NAG(s, k)$'s for different s strings. Similarly, let $B_m(R)$ be the maximal value of the $NAG(r, k)$'s for different r strings in R, and this value can be easily precalculated when constructing the VGRAM index structure. We can use $B_m(R) + B_m(S)$ as a new (constant) upper bound on the gram-based hamming distance between a string in R and a string in S.

Optimization can be done by utilizing the different hamming-distance bounds for different string pairs. We illustrate an optimization using an example. Assume the $NAG(r, k)$ values of strings r in R are in the range of $[1, 12]$, while the maximal upper bound for S, i.e., $B_m(S)$, is 10. We partition the strings in R into three groups: R_1 with $NAG(r, k)$ values in $[1, 4]$, R_2 with $NAG(r, k)$ values in $[5, 8]$, and R_3 with $NAG(r, k)$ values in $[9, 12]$. (Other partition schemes are also possible.) For R_1 strings, we generate a set of signatures using the hamming-distance bound $4 + B_m(S) = 14$, while we also generate a set of signatures for S using the same bound 14. We use these signatures to join R_1 with S to find similar pairs. Similarly, we join R_2 with S by using their signatures based on the hamming-distance bound $8 + B_m(S) = 18$; we join R_3 with S by using their signatures based on the hamming-distance bound $12 + B_m(S) = 22$. Notice that each of the joins is very efficient since (1) there are fewer R strings; (2) each hamming-distance bound is customized and tighter than the constant bound for the entire collection R, giving the algorithm a better chance to choose better signatures. We could further improve the performance by partitioning S into different groups, and generating different sets of signatures for different groups using different hamming-distance bounds.

7. EXPERIMENTS

In this section, we present our experimental results of the VGRAM technique. We evaluated the effect of the different factors on the performance of VGRAM. We also adopted VGRAM in the three existing algorithms to show the performance improvements. We used the following three data sets in the experiments.

- *Data set 1*: person names. It was downloaded from the Web site of the Texas Real Estate Commission.[4] The file included a list of records of person names, companies, and addresses. We used about $151K$ person names, with an average length of 33.
- *Data set 2*: English dictionary. We used the English dictionary from the Aspell spellchecker for Cygwin. It included $149,165$ words, with an average length of 8.
- *Data set 3*: paper titles. It was from the DBLP Bibliography.[5] It included about $277K$ titles, with an average string length of 62.

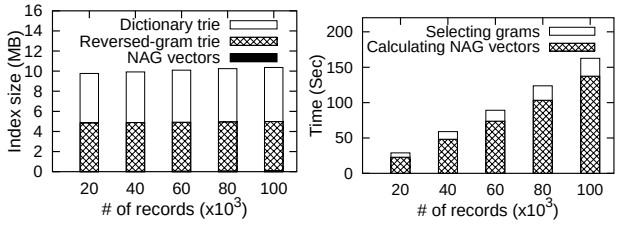
Whenever larger datasets were needed, we randomly selected records from a data set, made minor modifications, and inserted the new records into the data set. For approximate selection queries, we generated a query by randomly selecting a string from a data set, and making minor changes to the string to form a query string. For each string we extended it with multiple copies of a prefix character and multiple copies of a suffix character, and both characters were not part of the alphabet of the dataset. We got consistent results for these data sets. Due to space limitation, for some experiments we report the results on some of the data sets.

All the algorithms were implemented using Microsoft Visual C++. The experiments were run on a Dell GX620 PC with an Intel Pentium 3.40GHz Dual Core CPU and 2GB memory, running a Windows XP operating system.

[4] www.trec.state.tx.us/LicenseeDataDownloads/trecfile.txt
[5] www.informatik.uni-trier.de/~ley/db/

7.1 VGRAM Overhead

Index Size: We evaluated the overhead of VGRAM. We chose the DBLP data due to its larger size and longer strings. We varied the string number and collected the index size. We used the following setting: q_{min}=5, q_{max}=7, frequency threshold $T = 500$, and the LargeFirst pruning policy. Fig. 8(a) shows the index size of VGRAM for different data sizes, including its dictionary trie, reversed-gram trie, and the NAG vectors (each value in the vectors was stored as a byte). The vector size is too small to be seen. When there were 20K strings, the index was only 9.75MB. When the record number increased to 100K, the index size was still just 10.27MB. The slow growth is because the number of grams in the index does not increase much as the data size increases. The experiments on the other two data sets showed similar results: the index size was even smaller, and grew slowly as the data size increased. For example, for the person name data set, we used q_{min}=4, q_{max}=6, frequency threshold T = 1000, and the LargeFirst policy. When there were 10K strings, the index was only 1.6MB. When the number of records increased to 500K, the index was still just 4.41MB.



(a) Index size. (b) Construction time.

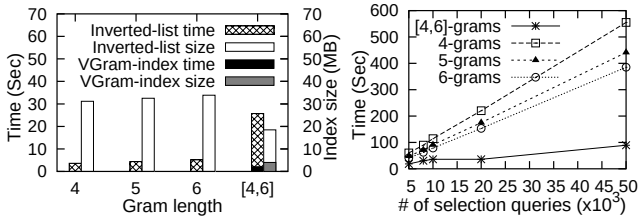
Figure 8: VGRAM index and construction time (DBLP titles).

Fig. 8(b) shows the construction time of VGRAM, including the time to construct its gram dictionary, and the time to calculate the NAG vectors. It shows that a large portion of the time was spent on calculating the NAG vectors. The construction time grew linearly as the data size increased. When there were 20K strings, it took about 30 seconds, and the time grew to 160 seconds for 100K strings.

7.2 Benefits of Using Variable-Length Grams

We compared the performance of algorithms using fixed-length grams and that of using variable-length grams. For the data set of 150K person names, for fixed-length grams, we varied the q value between 4 and 6. For each q, we built an inverted-list index structure using the grams. We generated a set of approximate selection queries with an edit distance threshold $k = 1$. We increased the number of selection queries, and measured the total running time. In addition, we also used VGRAM to build an index structure. We used the MergeCount algorithm as described in Section 6.1, since it is a classic algorithm representing those based on merging inverted lists of grams. We measured the running times for both the original algorithm and the one adopting the VGRAM technique based on the following setting: $q_{min} = 4$, q_{max} = 6, frequency threshold T = 1000, and LargeFirst pruning policy.

Fig. 9(a) shows the construction time and index size for



(a) Construction time/size. (b) Query performance.

Figure 9: Performance of fixed-length grams and variable-length grams.

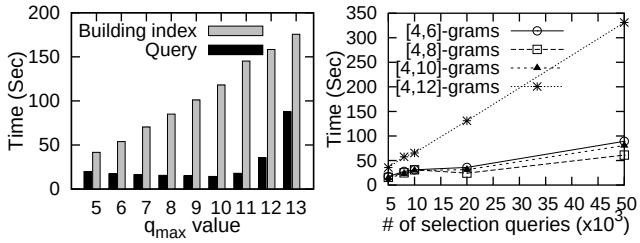
different gram-length settings. For fixed-length grams, as q increased, the index-building time increased from 3.6 seconds ($q = 4$) to 5.2 seconds ($q = 6$). VGRAM took 2.1 seconds to build its own index structure, and 23.6 seconds to build the corresponding inverted lists. So the construction-time overhead was small. In addition, by paying the cost of the VGRAM index, we can significantly reduce the inverted-list index size. The VGRAM index was about 4.0MB, while its inverted-list size was 14.4MB, which was about 46% of the inverted-list index size for $q = 4$ (31.1MB), and 43% of the index size of $q = 6$ (33.8MB). Since the VGRAM index increased very slowly (shown in Fig. 8(a)), the reduction on the total index size will become more significant as the data size increases. In addition to saving the index storage, VGRAM also improved the query performance, as shown in Fig. 9(b). For different q values, the best performance for the fixed-length approach was achieved when $q = 6$. When there were $50K$ selection queries, this approach took about 385 seconds, while by adopting VGRAM it took only 89 seconds, which improved the performance by more than 3 times!

7.3 Effect of q_{max}

We next evaluated the effect of the q_{max} value. We used the same data set with the same setting for VGRAM. We set q_{min} to 4, and varied q_{max} from 6 to 13. We set frequency threshold T to be 1000. Fig. 10(a) shows the time of building the index (including the VGRAM index and inverted lists) and the total running time for $5K$ selection queries with an edit distance 1. It shows that as q_{max} increased, the time of building the index always increased. The main reason is that the frequency trie became larger, which took more time to prune subtries to decide the grams. In addition, it also took more time to compute the NAG vectors for the strings.

An interesting observation is that, as q_{max} increased, the total selection-query time first decreased, reached a minimal value at $q_{max} = 10$, then started increasing. The main reason is that when q_{max} was small, increasing this value gave VGRAM a good chance to find high-quality grams. However, when q_{max} became too big, there could be many relatively long grams, causing the count bounds for strings to be loose. Thus it could reduce the effect of the count filtering technique, resulting in more false positives to be verified.

Fig. 10(b) shows the total times of answering different numbers of selection queries for different q_{max} values. Overall, the technique achieved the best performance when q_{max} = 8. These results suggest that q_{max} should not be too big.



(a) Construction time and dictionary quality (measured as query time).

(b) Query performance.

Figure 10: Effect of different q_{max} values.

7.4 Effect of Frequency Threshold

We evaluated the effect of the frequency threshold T on the time of building the index structure and the query performance (see Section 4.2). We ran the MergeCount algorithm on the person-name data set with the following setting: 150K person names, $q_{min} = 4$, $q_{max} = 12$, edit distance threshold $k = 1$, and Random pruning policy. Fig. 11(a) shows the time of building the index structure for different T values. It included the times of different steps: building the initial frequency trie, pruning the trie to decide grams, calculating the NAG vectors, and building inverted lists. The time for building the reversed-gram trie was negligible. We can see that most of the time was spent on building the initial frequency trie and computing the NAG vectors. As T increased, most times did not change much, while the time of calculating the vectors increased slightly. Fig. 11(b) shows how the total index size changed for different thresholds. Fig. 11(c) shows how the query time changed as T increased. The running time first decreased, then increased. The best performance was achieved when T was around 1500.

7.5 Effect of Different Pruning Policies

We evaluated the policies, Random, SmallFirst, and LargeFirst, to select children of a node in the frequency trie to decide what grams to keep in the final dictionary (Section 4.2). Since there was little difference in their times to build the index structure, we only report the results on the performance of queries using the corresponding gram dictionary.

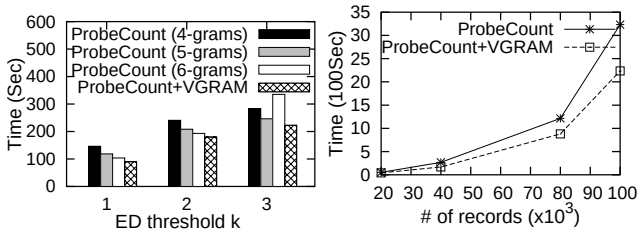
Table 1: Effect of three pruning policies to construct a gram dictionary.

Policy	Total time (sec)	Avg. # of grams per query	Total length of inverted lists per query	Avg. # of candidate strings
Random	51.03	18.3	20.4K	137
SmallFirst	59.69	20.3	33.7K	124
LargeFirst	48.02	17.7	16.9K	137

Table 7.5 shows the results over the data set of person names with the following setting: 100K person names, $q_{min} = 4$, $q_{max} = 6$, frequency threshold $T = 500$, and 10K selection queries with an edit-distance threshold $k = 2$. The three policies had different effects on the query performance. LargeFirst had the best performance, while SmallFirst had the worst one. From the table we can see that, using the dictionary produced by SmallFirst, a string can generate relatively more grams, resulting in more inverted lists to merge. The relatively more inverted lists resulted in more time to merge them. On the other hand, the average number of candidates for each query (last column) was similar for these policies, so they had the similar amount of time to postprocess the candidates. On the contrary, the gram dictionary produced by LargeFirst converted a string to fewer grams, resulting in fewer lists to merge. These factors make LargeFirst produce the best gram dictionary.

7.6 Improving ProbeCount

We have implemented the ProbeCount algorithm for approximate string joins. We used the algorithm to do a self join of a subset of the records in the person-name data set. We varied q from 4 to 6, and evaluated the performance of the algorithm adopting the VGRAM technique with the following setting: $q_{min} = 4$, $q_{max} = 6$, and $T = 200$. We used different edit-distance thresholds $k = 1, 2, 3$, and varied the number of records in the join. Fig. 12(a) shows the time of the basic ProbeCount algorithm on $50K$ records and the improved one called ProbeCount+VGRAM. The results show that adopting VGRAM increased the performance of the algorithm. For instance, when $k = 1$ and $q = 6$, the basic algorithm took 104 seconds, while the ProbeCount+VGRAM algorithm took 90 seconds.



(a) Time versus k value.

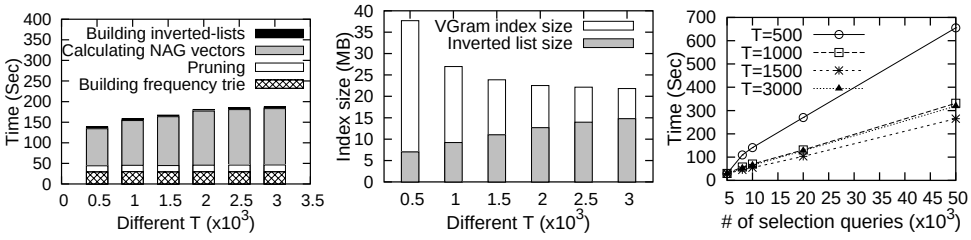
(b) Time versus data size.

Figure 12: Improving ProbeCount (for VGRAM, $q_{min} = 4$ and $q_{max} = 6$).

Fig. 12(b) shows the results of ProbeCount and ProbeCount+VGRAM for different data sizes (number of records) for $k = 3$. We used 5-grams for ProbeCount, since it gave the algorithm the best performance. When adopting VGRAM, we chose $q_{min} = 4$ and $q_{max} = 6$, and let $T = 1000$. As the data size increased, both algorithms took more time. Adopting VGRAM reduced the query time by about 19.5% for 20K person names, and 30.8% for 100K person names.

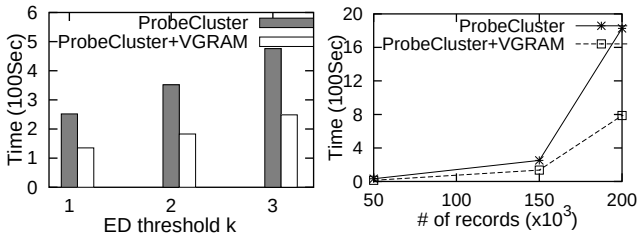
7.7 Improving ProbeCluster

We implemented the ProbeCluster join algorithm and the corresponding algorithm adopting VGRAM, called ProbeCluster+VGRAM. We did a self join of the person name data set, by using $q = 5$ for ProbeCluster. For VGRAM, we used $q_{min} = 5$, $q_{max} = 7$, and $T = 1000$. The index structure was assumed to have been constructed. Fig. 13(a) shows the performance improvements for different k values. For instance, when $k = 2$, the basic algorithm took 352 seconds, while the improved algorithm used only 183 seconds.



(a) Construction time. (b) Index size. (c) Query performance.

Figure 11: Effect of frequency threshold.



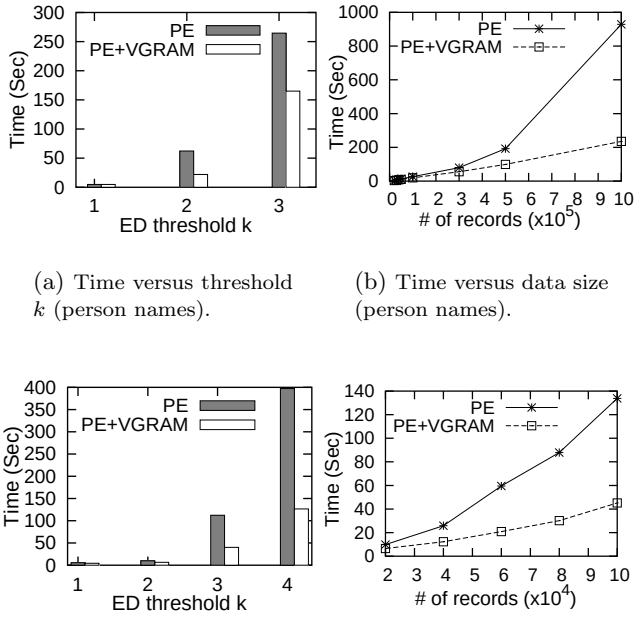
(a) Time versus k value. (b) Time versus data size.

Figure 13: Improving ProbeCluster.

Fig. 13(b) shows the total running time of the two algorithms for different data sizes (number of records), when k=2. The results show the performance improvement. For instance, when there were $200K$ records, the basic algorithm took about $1,827$ seconds, while the improved algorithm used 786 seconds only.

7.8 Improving PartEnum

We implemented the PartEnum join algorithm and the corresponding algorithm adopting VGRAM. We did a self join of the person name data set, by using $q = 4$ for PartEnum. For VGRAM, we used $q_{min} = 4$, $q_{max} = 6$, and $T = 1000$. We assumed the index structure was already constructed. Fig. 14(a) shows the performance improvements for different k values. For instance, when $k = 2$ and the data size = 150K, the basic algorithm took 73 seconds, while the improved algorithm used only 26 seconds. Fig. 14(b) shows how the total running time changed as the data size (number of records) changed, when k=2. For instance, where doing a self-join of 1 million records, VGRAM reduced the time of PartEnum from about 950 seconds to 220 seconds. The reduction was consistent for different data sizes. We also did the experiments for the DBLP title data set. We chose 20K titles for a self-join and the results for different k values are shown in Figs. 14(c). In addition, Fig. 14(d) shows the improvement for different data sizes. Our results on the aspell dictionary data set were similar; adopting VGRAM reduced the running time of PartEnum by around 30%.



(a) Time versus threshold k (person names).

(b) Time versus data size (person names).

(c) Time versus threshold k (DBLP titles).

(d) Time versus data size (DBLP titles).

Figure 14: Improving PartEnum.

8. DISCUSSIONS AND CONCLUSIONS

Dynamic Maintenance: The VGRAM index structure can be easily maintained for string insertions and deletions. When a new string is inserted, we calculate its NAG vector using the index and store it in the index. Deletion can be processed similarly. If there are a lot of updates on the string collection, and the quality of grams changes too much, we can rebuild the index structure efficiently.

Other Edit Distance Variants: The VGRAM technique can be extended slightly to support approximate queries using variants of edit distance. (1) Approximate substring queries [10]: A query asks for strings from a collection that have a substring similar to a given string or a collection of strings. VGRAM can be used to answer such queries based on the observation that if string s_1 is similar to a substring of string s_2, then s_1 and s_2 should share enough common grams. (2) Edit distance allowing block moves [10]: A vari-

ant of edit distance allows a move of a block of contiguous characters with a constant cost. The extended edit distance between two strings is the minimum cost of operations (insertion, deletion, substitution, all with a unit cost, and block move with a constant cost) needed to transform one string to the other string. VGRAM can be used to answer such queries by analyzing the effect of each move operation on the grams close to the three "gaps" generated by the move.

Using VGRAM in DBMS: Several recent studies [1, 3, 10] have developed techniques to support approximate string queries inside a relational DBMS using SQL queries. We can adopt the VGRAM technique in these algorithms inside a DBMS as follows. The trie and the reversed-gram trie can be stored and implemented in an application level on top of the DBMS. For instance, we could implement the VGEN algorithm in Fig. 3 as a user-defined function (UDF) to generate a set of positional grams for a string. The NAG vectors for the strings can be stored as a table inside the DBMS. Utilizing these tables, with a small amount of code at the application level, we can adopt the VGRAM technique inside a DBMS to support approximate queries.

Conclusions: In this paper we have developed a novel technique, called VGRAM, to improve performance of approximate string queries. It is based on the idea of choosing variable-length, high-quality grams, which can be used to identify similar strings based on their common grams. We gave a full specification of the technique, including its index structure, how to generate grams for a string using the index structure, and the relationship between the similarity of two strings and the similarity of their grams. We showed how to adopt this technique in a variety of existing algorithms. Our extensive experiments on real data sets have shown that the technique can be easily used to improve these algorithms, without substantial modifications of these algorithms.

Acknowledgements: Chen Li was partially supported by the NSF CAREER Award No. IIS-0238586 and a Google Research Award. Bin Wang and Xiaochun Yang were partially supported by the Program for New Century Excellent Talents in Universities (NCET), NSF China grant 60503036, and the Fok Ying Tong Education Foundation Award 104027. Part of the work was done when Bin Wang and Xiaochun Yang visited UC Irvine in 2006.

9. 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] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and Efficient Fuzzy Match for Online Data Cleaning. In *SIGMOD*, pages 313–324, 2003.

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

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

[6] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, pages 201–212, 1998.

[7] J. F. de Silva and G. P. Lopes. A local maxima method and a fair dispersion normalization for extracting multiword units. In *6th Meeting on the Mathematics of Language*, pages 369–381, 1999.

[8] P. Fogla and W. Lee. q-gram matching using tree models. *IEEE Transactions on Knowledge and Data Engineering*, 18(4), 2006.

[9] D. Gayo-Avello, D. Á. Gutiérrez, and J. Gayo-Avello. Application of variable length *n*-gram vectors to monolingual and bilingual information retrieval. In *CLEF*, pages 73–82, 2004.

[10] 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.

[11] J. Houvardas and E. Stamatatos. N-gram feature selection for authorship identification. In *AIMSA*, pages 77–86, 2006.

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

[13] M. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14(5–7), 1995.

[14] 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.

[15] D. E. Knuth. Sorting and Searching. *The Art of Computer Programming*, 3:481, 1993.

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

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

[18] V. Levenshtein. Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones. *Profl. Inf. Transmission*, 1:8–17, 1965.

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

[20] T. Niesler and P. Woodland. A variable-length category-based n-gram language model. In *Proc. ICASSP '96*, pages 164–167, 1996.

[21] B.-W. On, D. Lee, J. Kang, and P. Mitra. Comparative study of name disambiguation problem using a scalable blocking-based framework. In *JCDL*, 2005.

[22] S. Sarawagi and A. Kirpai. Efficient Set Joins on Similarity Predicates. In *SIGMOD*, pages 743–754, 2004.

[23] E. Sutinen and J. Tarhio. On Using q-Grams Locations in Approximate String Matching. In *ESA*, pages 327–340, 1995.

[24] E. Sutinen and J. Tarhio. Filtration with q-Samples in Approximate String Matching. In *CPM*, 1996.

[25] E. Ukkonen. Approximae String Matching with q-Grams and Maximal Matching. *Theor. Comut. Sci.*, 1:191–211, 1992.

[26] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, 1999.