Space-Constrained Gram-Based Indexing for Efficient Approximate String Search (Technical Report)

Alexander Behm Shengyue Ji Chen Li Jiaheng Lu
Department of Computer Science, University of California, Irvine, CA 92697, USA
{abehm,shengyuj,chenli,jlu2}@ics.uci.edu

***Abstract*— Answering approximate queries on string collections is important in applications such as data cleaning, query relaxation, and spell checking, where inconsistencies and errors exist in user queries as well as data. Many existing algorithms use gram-based inverted-list indexing structures to answer approximate string queries. These indexing structures are "notoriously" large compared to the size of their original string collection. In this paper, we study how to reduce the size of such an indexing structure to a given amount of space, while retaining efficient query processing. We first study how to adopt existing inverted-list compression techniques to solve our problem. Then, we propose two novel approaches for achieving the goal: one is based on discarding gram lists, and one is based on combining correlated lists. They are both orthogonal to existing compression techniques, exploit a unique property of our setting, and offer new opportunities for improving query performance. For each approach we analyze its effect on query performance and develop algorithms for wisely choosing lists to discard or combine. Our extensive experiments on real data sets show that our approaches provide applications the flexibility in deciding the tradeoff between query performance and indexing size, and can outperform existing compression techniques. An interesting and surprising finding is that while we can reduce the index size significantly (up to 60% reduction) with tolerable performance penalties, for 20-40% reductions we can even improve query performance compared to original indexes.**

I. Introduction

Many information systems need to support approximate string queries: given a collection of textual strings, such as person names, telephone numbers, and addresses, find the strings in the collection that are similar to a given query string. The following are a few applications. In record linkage [17], 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 `spielberg` versus `spielburg`. In Web search, many search engines provide the "Did you mean" feature, which can benefit from the capability of finding keywords similar to a keyword in a search query. Other information systems such as Oracle and Lucene also support approximate string queries on relational tables or documents.

Various functions can be used to measure the similarity between strings, such as edit distance (a.k.a. Levenshtein distance), Jaccard similarity, and cosine similarity. Many algorithms are developed using the idea of "grams" of strings. A *q-gram* of a string is a substring of length q that can be used as a signature for the string. For example, the 2-grams of the string `bingo` are `bi`, `in`, `ng`, and `go`. These algorithms rely on an index of inverted lists of grams for a collection of strings to support queries on this collection. Intuitively, we decompose each string in the collection to grams, and build an inverted list for each gram, which contains the id of the strings with this gram. For instance, Fig. 1 shows a collection of 6 strings and the corresponding inverted lists of their 2-grams.

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

(a) Strings.

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

(b) Inverted lists of string ids.

Fig. 1. Strings and their inverted lists of 2-grams.

The algorithms answer an approximate string query using the following observation: if a string r in the collection is similar enough to the query string, then r should share a certain number of common grams with the query string. Therefore, we decompose the query string to grams, and locate the corresponding inverted lists in the index. We find those string ids that appear at least a certain number of times on these lists, and these candidates are post-processed to remove the false positives.

Motivation: These gram-based inverted-list indexing structures are "notorious" for their large size relative to the size of their original string data. This large index size causes problems for applications. For example, many systems require a very high real-time performance to answer a query. This requirement is especially important for those applications adopting a Web-based service model. Consider online spell checkers used by email services such as Gmail, Hotmail, and Yahoo! Mail, which have millions of online users. They need to process many user queries each second. There is a big difference between a 10ms response time versus a 20ms response time, since the former means a throughput of 100 queries per second (QPS), while the latter means 20 QPS. Such a high-performance requirement can be met only if the index is in memory. In another scenario, consider the case where these algorithms are implemented inside a database system, which can only allocate a limited amount of memory for the inverted-list index, since there can be many other tasks in the database system that also need memory. In both scenarios, it is very critical to reduce the index size as much as we can to meet a given space constraint.

Contributions: In this paper we study how to reduce the size of such index structures, while still maintaining a high query performance. In Section III we study how to adopt existing inverted-list compression techniques to our setting [32]. That is, we partition an inverted list into fixed-size segments and compress each segment with a word-aligned integer coding scheme. To support fast random access to the compressed lists, we can use synchronization points [24] at each segment, and cache uncompressed segments to improve query performance. Most of these compression techniques were proposed in the context of information retrieval, in which conjunctive keyword queries are prevalent. In order to ensure correctness, lossless compression techniques are usually required in this setting.

The setting of approximate string search is unique in that a candidate result must occur only a *certain number* of times among all the inverted lists, and not necessarily on all the inverted lists. We exploit this unique property to develop two novel approaches for achieving the goal. The first approach is based on the idea of discarding some of the lists. We study several technical challenges that arise naturally in this approach (Section IV). One issue is how to compute a new lower bound on the number of common grams (whose lists are not discarded) shared by two similar strings, the formula of which becomes technically interesting. Another question is how to decide lists to discard by considering their effects on query performance. In developing a cost-based algorithm for selecting lists to discard, we need to solve several interesting problems related to estimating the different pieces of time in answering a query. For instance, one of the problems is to estimate the number of candidates that share certain number of common grams with the query. We notice that several related techniques [15], [22], [19] cannot be used directly to solve these problems (see Section IV-A.4). We develop a novel algorithm for efficiently and accurately estimating this number. We also develop several optimization techniques to improve the performance of this algorithm for selecting lists to discard.

The second approach is combining some of the correlated lists (Section V). This approach is based on two observations. First, the string ids on some lists can be correlated. For example, many English words that include the gram "`tio`" also include the gram "`ion`". Therefore, we could combine these two lists to save index space. Each of the two grams shares the union list. Notice that we could even combine this union list with another list if there is a strong correlation between them. Second, recent algorithms such as [20], [10] can efficiently handle long lists to answer approximate string queries. As a consequence, even if we combine some lists into longer lists, such an algorithm can still achieve a high performance. We study several technical problems in this approach, and analyze the effect of combining lists on a query. Also, we exploit a new opportunity to improve the performance of existing list-merging algorithms. Based on our analysis we develop a cost-based algorithm for finding lists to combine.

We have conducted extensive experiments on real datasets for the list-compression techniques mentioned above (Section VI). While existing inverted-list compression techniques can achieve compression ratios up to 60%, they considerably increase the average query running time due to the online decompression cost. We two novel approaches are orthogonal to existing inverted-list-compression techniques, and offer unique optimization opportunities for improving query performance. Note that using our novel approaches we can still compute the *exact* results for an approximate query without missing any true answers. The experimental results show that (1) the novel techniques can outperform existing compression techniques, and (2) the new techniques provide applications the flexibility in deciding the tradeoff between query performance and indexing size. An interesting and surprising finding is that while we can reduce the index size significantly (up to a 60% reduction) with tolerable performance penalties, for 20-40% reductions we can even improve the query performance compared to the original index. Our techniques work for commonly used functions such as edit distance, Jaccard, and cosine. We mainly focus on edit distance as an example for simplicity.

A. Related Work

In the literature the term *approximate string query* also means the problem of finding within a long text string those substrings that are similar to a given query pattern. See [25] for an excellent survey. 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.

In the field of list compression, many algorithms [23], [31] are developed to compress a list of integers using encoding schemes such as LZW, Huffman codes, and bloom filters. In Section III we discuss in more detail how to adopt these existing compression techniques to our setting. One observation is that these techniques often need to pay a high cost of increasing query time, due to the online decompression operation, while our two new methods could even reduce the query time. In addition, the new approaches and existing techniques can be integrated to further reduce the index size, as verified by our initial experiments.

Many algorithms have been developed for the problem of approximate string joins based on various similarity functions [2], [3], [4], [6], [9], [27], [28], [29], especially in the context of record linkage [17]. Some of them are proposed in the context of relational DBMS systems. Several recent papers focused on approximate *selection* (or search) queries [10], [20]. The techniques presented in this paper can reduce index sizes, which should also benefit join queries, and the corresponding cost-based analysis for join queries needs future research. Hore et al. [12] proposed a gram-selection technique for indexing text data under space constraints, mainly considering SQL LIKE queries. Other related studies include [7], [16], [26]. There are recent studies on the problem of estimating the selectivity of SQL LIKE substring queries [5], [14], [18], and approximate string queries [22], [15], [19], [11].

Recently a new technique called VGRAM [21], [30] was proposed to use variable-length grams to improve approximate-string query performance and reduce index size. This technique, as it is, can only support edit distance, while the techniques presented in this paper support a variety of similarity functions. Our techniques can also provide the user the flexibility to choose the tradeoff between index size and query performance, which is not provided by VGRAM. Our experiments show that our new techniques outperform VGRAM, and potentially they can be integrated with VGRAM to further reduce the index size (Sections VI-D and VII).

II. Preliminaries

Approximate String Search: Let S be a collection of strings. An approximate string search query includes a string s and a threshold k. It asks for all $r \in S$ such that the distance between r and s is within the threshold k. Various distance functions can be used, such as edit distance, Jaccard similarity and cosine similarity. Take edit distance as an example. Formally, 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 two strings s_1 and s_2 as $ed(s_1, s_2)$. For example, $ed($“Levenshtein”, “Levnshtain”$) = 2$. Using this function, an approximate string search with a query string q and threshold k is finding all $s \in S$ such that $ed(s, q) \leq k$.

Grams: 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. We introduce two characters α and β not in Σ. Given a string s and a positive integer q, we extend s to a new string s' by prefixing $q - 1$ copies of α and suffixing $q - 1$ copies of β.[1] 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 $G(s, q)$, or simply $G(s)$ when the q value is clear in the context, is obtained by sliding a window of length q over the characters of s'. For instance, suppose $\alpha = \#$, $\beta = \$$, $q = 3$, and s = irvine. We have: $G(s, q) = \{$(1, ##i), (2, #ir), (3, irv), (4, rvi), (5, vin), (6, ine), (7, ne\$), (8, e\$\$)$\}$. The number of positional q-grams of the string s is $|s| + q - 1$. For simplicity, in our notations we omit positional information, which is assumed implicitly to be attached to each gram.

[1]The results in the paper extend naturally to the case where we do not extend a string to produce grams.

Inverted Lists: We assume an index of inverted lists for the grams of strings in the collection S. In the index, for each gram g of the strings in S, we have a list l_g of the ids of the strings that include this gram (possibly with the corresponding positional information). All the ids in l_g are sorted in the increasing order. It is observed in [27] that an approximate query with a string s can be answered by solving the following generalized problem:

> *T-occurrence Problem*: Find the string ids that appear at least T times on the inverted lists of the grams in $G(s, q)$, where T is a constant related to the similarity function, the threshold in the query, and the gram length q.

Take edit distance as an example. For a string $r \in S$ that satisfies the condition $ed(r, s) \leq k$, it should share at least the following number of q-grams with s:

$$T_{ed} = (|s| + q - 1) - k \times q. \tag{1}$$

Intuitively, the query has $|s| + q - 1$ grams, and each edit operation can destroy at most q grams of length q. Thus at least $(|s| + q - 1) - k \times q$ grams in $G(s)$ can “survive.”

Several existing algorithms [20], [27] are proposed for answering approximate string queries efficiently. They first solve the T-occurrence problem to get a set of string candidates, and then check their real distance to the query string to remove false positives. Note that if the threshold $T \leq 0$, then the entire data collection needs to be scanned to compute the results. We call it a *panic case*. One way to reduce this scan time is to apply filtering techniques. For example, length filtering and prefix filtering [9], [20] can separate the collection to different groups. Given a query, we only need to scan some of groups to find the results. To summarize, the following are the pieces of time needed to answer a query:

- If the lower bound T (also called “merging threshold”) is positive, then the time includes the time to traverse the lists of the grams in the query to find candidates (called “merging time”) and the time to remove the false positives from the candidates (called “post-processing time”).
- If the lower bound T is zero or negative, we need to spend the time (called “scan time”) to scan the entire data set, possibly using filtering techniques.

In the following sections we adopt existing techniques and develop new techniques to reduce this index size. For simplicity, we first focus on the edit distance function. In Section VII we will see that the results can be naturally extended to other commonly used functions.

III. Adopting Existing Inverted-List-Compression Techniques

There are many techniques [32] focusing on inverted-list compression in the literature, which mainly study the problem of representing integers on inverted lists efficiently to save storage space. In this section we will briefly review these techniques, adopt them to our problem setting and then discuss the limitations of these techniques.

A. Encoding and Decoding Lists of Integers

Most of the list-compression techniques exploit the fact that ids on an inverted list are monotonically increasing integers. For example, suppose we have $l = (id_1, id_2, \dots, id_n)$, $id_i < id_{i+1}$ for $1 \leq i < n$. If we take the differences of adjacent integers to construct a new list $l' = (id_1, id_2 - id_1, id_3 - id_2, ..., id_n - id_{n-1})$, the integers in this new list tend to be smaller than the original ids. This l' list is called the gapped representation of the inverted list l, since it stores the "gap" between each pair of adjacent integers. Using this gapped form, many integer-compression techniques such as gamma codes, delta codes, and Golomb codes [32] can efficiently encode inverted lists by using shorter representations for small integers. We choose one of the recent techniques called Carryover-12 [1] to adopt in our setting.

B. Segmenting, Compressing, and Indexing Inverted Lists

An issue arises when using the encoded, gapped representation of a list. Since decoding is usually achieved in a sequential way, a sequential scan on the list might not be affected too much. However, random accesses could become expensive. Even if the compression technique allows us to decode the desired integer directly, the gapped representation still requires restoring of all preceding integers. Many efficient list-merging algorithms such as DivideSkip [20] rely heavily on binary search on the inverted lists. This problem can be solved by segmenting the list and introducing *synchronization points* [24]. Each segment is associated with a synchronization point and decoding can start from any synchronization point, so that only one segment needs to be decompressed in order to read a specific integer. One way to adopt this technique is to have each segment contain the same number of intergers. Since different encoded segments could have different sizes, it is necessary to index the starting offset of each encoded segment, so that they can be quickly located and decompressed. Figure 2 illustrates the idea of segmenting inverted lists and indexing compressed segments.

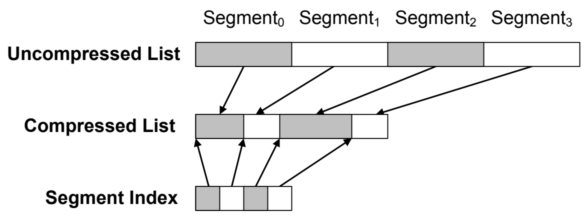


Fig. 2. Inverted-list compression with segmenting and indexing

A simple way to access elements is by decoding the corresponding segment for each integer access. If multiple integers within the same segment are requested, the segment will be decompressed multiple times. The repeated efforts can be alleviated using caching. Once a segment is decoded, it will remain in the cache for a while. All integer accesses to that segment will be answered using the cache without decoding the segment again, until the segment is replaced from the cache. We allocate a global cache pool for all inverted lists. More details are given in our experiments (Section VI).

C. Limitations of Existing Techniques

Most of these existing techniques were initially designed for compressing disk-based inverted indexes. Using a compressed representation, we can not only save disk space, but also decrease the number of disk I/Os. Even with the decompression overhead, these techinques can still improve query performance since disk I/Os are usually the major cost. When the inverted lists are in memory, these techinques require additional decompression operations, compared to non-compressed indexes. Thus, the query performance can only decrease. Another limitation is that these approaches have limited flexibility in trading query performance with space savings. Next we propose two novel methods to reduce the size of inverted lists, while retaining good query performance.

IV. Discarding Inverted Lists

In this section we study how to reduce the size of an inverted-list index by discarding some of the lists. That is, for all the grams from the strings in S, we only keep inverted lists for *some* of the grams, while we do not store those of the other grams. A gram whose inverted list has been discarded is called a *hole gram*, and the corresponding discarded list is called its *hole list*. Notice that a hole gram is different from a gram that has an empty inverted list. The former means the ids of the strings with this gram are not stored in the index, while the latter means no string in the data set has this gram.

We study the effect of hole grams on query answering. In Section IV-A we analyze how they affect the merging threshold, the list merging and post-processing, and discuss how the new running time of a single query can be estimated. For a single query, we present a dynamic programming algorithm for computing a tight lower bound on the number of nonhole grams shared by two similar strings assuming the edit distance function. Based on our analysis, we propose an algorithm to wisely choose grams to discard in the presence of space constraints, while retaining efficient processing. We develop various optimization techniques to improve the performance of this algorithm (Section IV-B).

A. Effects of Hole Grams on a Query

For an approximate query with a string s if $G(s)$ does not have hole grams, then the query is unaffected. If it does have hole grams, then the query may be affected in the following ways:

- The lower bound on the number of nonhole grams shared by a similar string in the collection could become smaller.
- As a consequence, the merging time and post-processing time could both increase.
- However, due to having a lower number of elements among the lists to process, the merging time could also decrease. Also, in come cases the modified threshold could decrease the post-processing time.
- If the new lower bound becomes zero or negative, then the query can only be answered by a scan.

1) Effects on Merging Threshold: Consider a string r in the collection S such that $ed(r,s) \leq k$. For the case without hole grams, r needs to share at least $T = (|s| + q - 1) - k \times q$ common grams in $G(s)$ (Equation 1). To find such an r, in the corresponding T-occurrence problem, we need to find string ids that appear on at least T lists of the grams in $G(s)$.

If $G(s)$ does have hole grams, the id of string r could have appeared on some of the hole lists. But we do not know on how many hole lists this string r could appear, since these lists have been discarded. We can only rely on the lists of those nonhole grams to find candidates. Thus the problem becomes deciding a lower bound on the number of occurrences of string r on the *nonhole* gram lists.

One simple way to compute a new lower bound is the following. Let H be the number of hole grams in $G(s)$, where $|G(s)| = |s| + q - 1$. Thus, the number of nonhole grams for s is $|G(s)| - H$. In the worst case, every edit operation can destroy at most q nonhole grams, and k edit operations could destroy at most $k \times q$ nonhole grams of s. Therefore, r should share at least the following number of nonhole grams with s:

$$T' = |G(s)| - H - k \times q. \tag{2}$$

We can use this new lower bound T' in the T-occurrence problem to find all strings that appear at least T' times on the nonhole gram lists as candidates.

The following example shows that this simple way to compute a new lower bound is pessimistic, and the real lower bound could be tighter. Consider a query string s = `irvine` with an edit-distance threshold $k = 2$. Suppose $q = 3$. Thus the total number of grams in $G(s)$ is 8. There are two grams `irv` and `ine` as shown in Figure 3. Using the formula above, an answer string should share at least 0 nonhole gram with string s, meaning the query can only be answered by a scan. This formula assumes that a single edit operation could destroy at most 3 grams, and two edit operations destroy at most 6 grams. However, a closer look at the positions of the hole grams tells us that a single edit operation can destroy at most 2 nonhole grams, and two edit operations can destroy at most 4 nonhole grams. Figure 3 shows two deletion operations that can destroy the largest number of nonhole grams, namely 4. Thus, a tighter lower bound is 2 and we can avoid the panic case. This example shows that we can exploit the positions of hole grams in the query string to compute a tighter threshold in the T-occurrence problem. We develop a dynamic programming algorithm to compute a tight lower bound on the number of common nonhole grams in $G(s)$ an answer string needs to share with the query string s with an edit-distance threshold k (a similar idea is also adopted in an algorithm in [30] in the context of the VGRAM technique [21]).

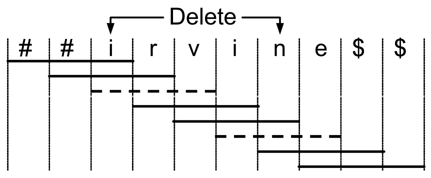


Fig. 3. A query string `irvine` with two hole grams. A solid horizontal line denotes a nonhole gram, and a dashed line denotes a hole gram.

Subproblem: Let $0 \leq i \leq |s|$ and $1 \leq j \leq k$ be two integers. Let $P(i,j)$ be an upper bound on the number of grams that can be destroyed by j edit operations that are at positions *no greater than i*. The overall problem we wish to solve becomes $P(|s|, k)$.

Initialization: Let $D_i^{d/s}$ denote the maximum possible number of grams destroyed by a deletion or substitution operation at position i, and let D_i^{ins} denote the maximum possible number of grams destroyed by an insertion operation after position i. For each $0 \leq i \leq |s|$ we set $P(i,0) = max(D_i^{d/s}, D_i^{ins})$.

Recurrence function: Consider the subproblem of computing a value for entry $P(i,j)$. Let g_i denote the gram starting from position i, which may either be a hole or a nonhole gram. If it is a hole then we can set $P(i,j) = P(i-1,j)$ because an edit operation at this position cannot be the most destructive one. Recall that we have already discarded the list belonging to the gram at i. If the gram starting from i is not a hole then we need to distinguish three cases:

- We have a deletion/substitution operation at position i. This will destroy grams up to position $i-q+1$ and consume one edit operation. The number of grams destroyed is $D_i^{d/s}$. Therefore, we can set $P(i,j) = P(i-q, j-1) + D_i^{d/s}$.
- We have an insertion operation after position i. This will destroy grams up to position $i-q+2$ and consume one edit operation. The number of grams destroyed is D_i^{ins}. Therefore, we can set $P(i,j) = P(i-q+1, j-1) + D_i^{ins}$.
- There is no operation at i. We can set $P(i,j) = P(i-1,j)$.

The following is a summary of the recurrence function:

$$P(i,j) = max \begin{cases} g_i \text{ is a hole} : P(i-1,j) \\ g_i \text{ is a gram} : \begin{cases} \text{delete/substitute } i : \\ P(i-q, j-1) + D_i^{d/s} \\ \text{insertion after } i : \\ P(i-q+1, j-1) + D_i^{ins} \\ \text{noop } i : \\ P(i-1,j) \end{cases} \end{cases} \tag{3}$$

After we have computed the maximun number of grams destroyed by k edit operations (denoted by D_{max}), we can get the new bound using the same idea as in Equation 1, except that the maximum number of grams destroyed is not $k \times q$ but D_{max}.

2) Effects on List-Merging: The running time of some merging algorithms (e.g. HeapMerge, ScanCount [20]) is insensitive to the merging threshold T and mainly depends on the total number of elements in all inverted lists. Therefore, the running time for these algorithms can only decrease by discarding some of the lists. Other merging algorithms (e.g. MergeOpt, DivideSkip [20]) separate the inverted lists into two groups which are processed separately: long lists and short lists. These algorithms perform a heap-based merging on the short lists to get candidate string ids that might occurr T times on all lists. In order to get the final solution to the T-occurrence problem, the algorithms verify those candidates against the long lists using binary search. The performance of these algorithms depends on which lists are put into the group of short list and long lists. For the algorithms mentioned here, the number of long lists is some function of T. Hence, their performance is sensitive to changes in T. In general, by discarding lists their performance may be affected positively or negatively. Which one is the case depends on the function for separating the lists into groups, and on the choice of lists to discard, i.e. the discarded lists may have belonged to the group of long lists or short lists. Another class of algorithms utilizes T to skip elements on the inverted lists which cannot be answers to the query. For example, based on T the MergeSkip and DivideSkip [20] algorithms perform a binary search to locate the next element on a list that could potentially be an answer to the query, skipping irrelevant elements. For these algorithms a higher T supports the skipping of elements. Therefore, decreasing T by discarding some lists can negatively affect their performance. However, by discarding some lists conceptually we skip all elements on those lists which may overcompensate for the decreased T, leading to a performance improvement. Note that the DivideSkip algorithm is sensitive to T in two ways: (1) because it uses the concept of short lists and long lists and (2) because it uses T to skip irrelevant elements.

3) Effects on Post-Processing: Intuitively, for a given query, introducing hole grams may only increase the number of candidates to post-process. We will show that this intuition is correct *if* we use the simple formula for calculating the new threshold T'. Surprisingly, if we use the dynamic programming algorithm to derive a tighter T', then the number of candidates for post-processing can even decrease.
Take the example given in Fig. 3. Suppose the edit distance threshold $k = 2$. Say that some string id i only appears on the inverted lists of `irv` and `ine`. Since $T = 2$ it is a candidate result. If we choose to discard the grams `irv` and `ine` as shown in Fig. 3, then each edit operation can destroy at most two non-hole grams and therefore $T' = T = 2$. After discarding the lists, the string i is not a candidate anymore since all the lists containing it have been discarded. This reduces the cost for postprocessing. Note that any string id which appears only on `irv` and `ine` cannot be a result to the query and would have been removed from the results during post-processing. We will now formalize the above notions.
Consider a query with string s, allowing k edit operatios, a bag of grams $G(s)$, a bag of hole-grams $H \subseteq G(s)$ and an original merging threshold T.
Lemma 1: If Equation 2 is used to compute the new merging threshold, introducing hole grams cannot decrease the number of candidates for postprocessing.
Let us examine a string id that occurred e times on all lists before discarding. If the string was a candidate before, then $e \geq T$. We may have discarded $\delta \leq |H|$ lists on which e occurred. Therefore, $e - \delta \geq T - |H|$, and the string will still be a candidate. If the string was not a candidate before, then $e < T$. The formula $e - \delta < T - |H|$ is not satisfiable for all $\delta \leq |H|$, and therefore the string may become a new candidate. Intuitively, consider the case where we discarded $|H|$ lists on which e did not occurr. It may be that $e \geq T - |H|$ although $e < T$.
Lemma 2: If the dynamic programming algorithm in Equation 3 is used to compute the new merging threshold, introducing hole grams could sometimes decrease the number of candidates for postprocessing.
The example given above ilustrates this and we shall now generalize the idea. Due to favorable positions of the hole grams it may be that a tighter bound yields $T' \geq T - |H|$ and therefore $T - |H| \leq T' \leq T$. A string id that occurred e times before discarding with $e \geq T$ may cease to be a candidate after introducing hole grams. This is because we may have discarded $\delta \leq |H|$ lists containing e and if $\delta > e - T$ then $e - \delta < T' \leq T$.

4) Estimating the Performance of a Query with Hole Grams: Since we are evaluating whether it is a wise choice to discard a specific list l_i, we would like to know, by discarding list l_i, how the performance of a single query Q will be affected using the indexing structure. In the previous sections we have analyzed the effects of introducing hole grams on a query. We now quantify these effects by estimating the running time of a query with hole grams.

Estimating the Merging Time: If the new merging threshold $T' > 0$, we need to solve the T-occurrence problem by accessing the inverted lists of the nonhole grams to find string ids that appear at least T' times. Here, we will consider one of the most efficient merging algorithms DivideSkip [20] which is sensitive to changes in T. Let the lists $l_1, l_2, \ldots, l_N$ be sorted in a decreasing order of their size. Then the running time for DivideSkip can be estimated as:

$$C_1 * \frac{\sum_{j=L+1}^{N} |l_j|}{T - L} * \sum_{i=1}^{L} log|l_i| + C_2 * \sum_{j=L+1}^{N} |l_j| * log(N - L) \quad (4)$$

For $C_1 = 1$ and $C_2 = 1$, the above equation represents the worst cost for the DivideSkip algorithm. The right side of the sum shows the cost of merging the short lists. The left side of the sum presents the cost for verifying the candidates from the short lists against the long lists. The coefficient C_1 represents that the actual number of candidates to verify against the long lists can be smaller than the maximim possible one (given in $\frac{\sum_{j=L+1}^{N}|l_j|}{T-L}$). The coefficient C_2 captures that skipping operations are performed while merging the short lists, and therefore not all elements need to be pushed and popped from the heap. To estimate the running time of DivideSkip we decide the coefficients C_1 and C_2 offline by running a subset of possible queries on the data set, collecting their merging times, and doing a linear regression. If filtering techniques are applied, we can decide C_1 and C_2 for each group separately to increase the accuracy of our estimation.

Estimating the Post-Processing Time: For each candidate from the T-occurrence problem, we need to compute the corresponding distance to the query to remove the false positives. This time can be estimated as

$$(\text{number of candidates}) \times (\text{average edit-distance time}).$$

Therefore, the main problem becomes how to estimate the number of candidates after solving the T-occurrence problem. This problem has been studied in the literature recently. For instance, Mazeika et al. [22] studied how to estimate this number. The methodology in the SEPIA technique [15] in the context of selectivity estimation of approximate string queries could be adopted with minor modifications to solve our problem. The technique in [19] might also be applicable. While these techniques could be used in our context, they have two limitations. First, their estimation is not 100% accurate, and an inaccurate result could greatly affect the accuracy of the estimated post-processing time, thus affecting the quality of the selected nonhole lists. Second, this estimation may need to be done very often when choosing lists to discard, and therefore needs to be efficient.

We develop an efficient, *incremental* algorithm that can compute a *very accurate* number of candidates for query Q if list l_i is discarded. The algorithm is called ISC, which stands for "Incremental-Scan-Count," where its idea comes from an algorithm called ScanCount developed in [20]. Although the original ScanCount is not the most efficient one for the T-occurrence problem, it has the nice property that it can be run incrementally.

Figure 4 shows the intuition behind this ISC algorithm. First, we analyze the query Q on the original indexing structure without any lists discarded. For each string id in the collection, we remember how often it occurs on all the inverted lists of the grams in the query and store them in an array C. (Later we will discuss how to reduce the size of this data structure.) Now we want to know if a list is discarded, how it affects the number of occurrences of each string id. For each string id r on list l belonging to gram g to be discarded, we decrease the corresponding value $C[r]$ in the array by the number of occurrences of g in the query string, since this string r will no longer have g as a nonhole gram.

After discarding this list for gram g, we first compute the new merging threshold T'. Then, we can find the new set of candidates by scanning the array C and recording those positions (corresponding to string ids) whose value is at least T'.

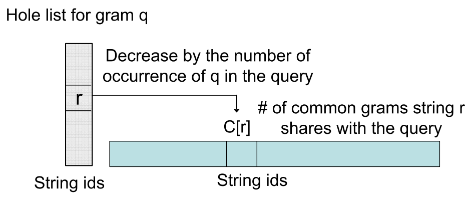

Hole list for gram q
Decrease by the number of occurrence of q in the query
r
C[r]
# of common grams string r shares with the query
String ids
String ids

Fig. 4. Intuition behind the Incremental-Scan-Count (ISC) algorithm.

In the example in Fig. 5 the hole list includes string ids 0, 2, 5, and 9. For each of the four ids, we decrease the corresponding value in the array by 1 (assuming the hole gram occurrs once in the query). Suppose the new threshold T' is 3. We scan the new array to find those string ids whose occurrence among all non-hole lists is at least 3. These strings, which are 0, 1, and 9 (in bold face in the figure), are candidates for the query using the new threshold after this list is discarded.

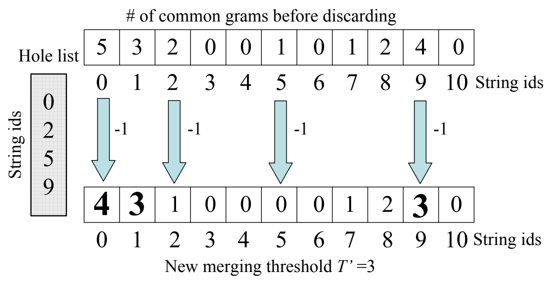

# of common grams before discarding
Hole list
5 3 2 0 0 1 0 1 2 4 0
0 1 2 3 4 5 6 7 8 9 10 String ids
String ids
0 2 5 9
-1
-1
-1
-1
4 3 1 0 0 0 0 1 2 3 0
0 1 2 3 4 5 6 7 8 9 10 String ids
New merging threshold T′ =3

Fig. 5. Running the ISC algorithm.

Since most of the values in the array are zero, we can keep a vector to store references to those positions in the array whose original values (without any lists discarded) are not zero. Instead of scanning the whole array to find the new set of candidates we can scan this vector and follow the references to the elements in the array that may be candidates.

Estimating the Scan Time: If the new merging threshold $T' > 0$, then the query can only be ansered by a scan. The time of a scan can be estimated in the same manner as the post-processing time, except that the number of candidates is the number of strings in the collection. If filtering techniques are applied then the number of candidates is the sum of all distinct string ids of groups that need to be considered by the query.

B. Algorithms for Choosing Inverted-Lists to Discard

In this section, we study how to wisely choose lists to discard in order to satisfy a given space constraint. The following are several simple approaches:

- LongList: Choose the longest lists to discard.
- ShortList: Choose the shortest lists to discard.
- RandomList: Choose random lists to discard.

These naive approaches blindly discard lists without considering the effects on query performance. Clearly, a good choice of lists to discard depends on the query workload. Based on our previous analysis, we present a cost-based algorithm called DiscardLists, as shown in Figure 6. Given the initial set of inverted lists, the algorithm iteratively selects lists to discard, based on the size of a list and its effect on the average query performance for a query workload $\mathcal{Q}$ if it is discarded. $\mathcal{Q}$ can be obtained from query logs, or from a uniformly distributied workload over the dataset when initially there is no query log available. The algorithm keeps selecting lists to discard until the total size of the remaining lists meets the given space constraint (line 2).

```
Algorithm: DiscardLists
Input: Inverted lists $L = \{l_1, \ldots, l_n\}$
       Constraint $B$ on the total list size
       Query workload $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$
Output: A set $D$ of lists in $L$ that are discarded
Method:
1.  $D = \emptyset$;
2.  WHILE ($B <$ (total list size of $L$)) {
3.      FOR (each list $l_i \in L$) {
4.          Compute size reduction $\Delta^i_{size}$ if discarding $l_i$
5.          Compute difference of average query time $\Delta^i_{time}$
                for queries in $\mathcal{Q}$ if discarding $l_i$
        }
6.      Use $\Delta^i_{size}$'s and $\Delta^i_{time}$'s of the lists to decide what
            lists to discard
7.      Add discarded lists to $D$
8.      Remove the discarded lists from $L$
    }
9.  RETURN $D$
```

Fig. 6. Cost-based algorithm for choosing inverted lists to discard.

In each iteration (lines 3-8), the algorithm needs to evaluate the quality of each remaining list l_i, based on the expected effect of discarding this list. The effect includes the reduction Δ^i_{size} on the total index size, which is the length of this list. It also includes the change Δ^i_{time} on the average query time for the workload $\mathcal{Q}$ after discarding this list.[2]

In line 6 in Figure 6, in each iteration of the DiscardLists algorithm, we need to use the Δ^i_{size}'s and Δ^i_{time}'s of the lists to decide what lists should be really discarded. There are many different ways to make this decision. One way is to choose a list with the smallest Δ^i_{time} value (notice that it could be negative). Another way is to choose a list with the smallest $\Delta^i_{time}/\Delta^i_{space}$ ratio.

There are several ways to reduce the computation time of the estimation:

(1) When discarding the list l_i, those queries whose strings do not have the gram of l_i will not be affected, since they will still have the same set of nonhole grams as before. Therefore, we only need to re-evaluate the performance of the queries whose strings have this gram of l_i. In order to find these strings efficiently, we build an inverted-list index structure for the *queries*, similar to the way we construct inverted lists for the strings in the collection. When discarding the list l_i, we can just consider those queries on the *query* inverted list of the gram for l_i.

[2]Surprisingly, Δ^i_{time} can be both positive and negative, since in some cases discarding lists can even reduce the average running time for the queries.

(2) We run the algorithm on a random subset of the strings. As a consequence, (a) we can make sure the entire inverted lists of these sample strings can fit into a given amount of memory. (b) We can reduce the array size in the ISC algorithm, as well as its scan time to find candidates. (c) We can reduce the number of lists to consider initially since some infrequent grams may not appear in the sample strings.

(3) We run the algorithm on a random subset of the queries in the workload $\mathcal{Q}$, assuming this subset has the same distribution as the workload. As a consequence, we can reduce the computation to estimate the scan time, merging time, and post-processing time (using the ISC algorithm).

(4) We do not discard those very short lists, thus we can reduce the number of lists to consider initially.

(5) In each iteration of the algorithm, we choose multiple lists to discard based on the effect on the index size and overall query performance. In addition, for those lists that have very poor time effects (i.e., they affect the overall performance too negatively), we do not consider them in future iterations, i.e., we have decided to keep them in the index structure. In this way we can reduce the number of iterations significantly.

V. Combining Inverted Lists

In this section, we study how to reduce the size of an inverted-list index by *combining* some of the lists. Intuitively, when the lists of two grams are similar to each other, using a single inverted list to store the union of the originial two lists for both grams could save some space. One subtlety in this approach is that the string ids on a list are treated as a *set* of ordered elements (without duplicates), instead of a *bag* of elements. By combining two lists we mean taking the *union* of the two lists so that space can be saved. Notice that the T lower bound in the T-occurrence problem is derived from the perspective of the grams in the query. (See Equation 1 in Section II as an example.) Therefore, if a gram appears multiple times in a data string in the collection (with different positions), on the corresponding list of this gram the string id appears only once. [CHECK] If we want to use the positional filtering technique (mainly for the edit distance function) described in [9], [20], for each string id on the list of a gram, we can keep a range of the positions of this gram in the string, so that we can utilize this range to do filtering. When taking the union of two lists, we need to accordingly update the position range for each string id.

We will first discuss the data structure and the algorithm for efficiently combining lists in Section V-A, and then analyze the effects of combining lists on query performance in Section V-B. We also show that an index with combined inverted lists gives us a new opportunity to improve the performance of list-merging algorithms (Section V-B.1). Based on our analysis we propose the CombineLists algorithm for choosing lists to combine in the presence of space constraints, while retaining efficient processing (Section V-C).

A. Combining Lists

This section studies the data structure and algorithm used to combine lists. In the original inverted-list structure, each gram g is mapped to a list l of string ids: $g \rightarrow l$. Combining two lists l_1 and l_2 will produce a new list $l_{new} = l_1 \cup l_2$, which is also sorted by string ids. All grams that were previously mapped to l_1 and l_2 (there could be several grams due to earlier combining operations) will be mapped to l_{new}. In this fashion we can support combining more than two lists iteratively. At the first glance a reverse mapping from lists to grams may seem to be necessary, because we have to track all grams associated with a list in order to update them. However, a data structure called Disjoint-Set with the algorithm Union-Find [8] can be utilized to efficiently combine more than two lists. The Disjoint-Set data structure is a special forest in which each node holds only the pointer to its parent. To apply the Disjoint-Set structure to this problem, each inverted list corresponds to a tree and each node corresponds to a gram. All nodes in the same tree share the same inverted list, which is stored as a single copy attached to the root. Initially all grams are disjoint tree roots, with their individual inverted lists attached to each of them. As we combine two trees, the root of one of them becomes a child of the root of the other, which takes over all string ids from the list of the first tree to generate the union list. Figure 7 shows the Disjoint-Set structure used with the Union-Find algorithm when combining list of g_2 and list of g_3, while g_2 and g_1 are already sharing the same list $l_1 \cup l_2$. After the combination of the two trees corresponding to the two lists, the result list is $l_1 \cup l_2 \cup l_3$ and attached to root g_1. The Union-Find algorithm can also shorten the path from nodes to their root to speed up inverted-list lookup operations.

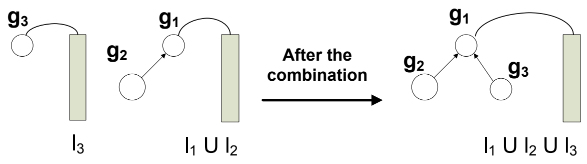


Fig. 7. Combining list of g_2 with list of g_3 using Union-Find

To satisfy a specific space constraint, we would like to know the exact size of saved space when combining lists. The size reduction of combining two lists l_1 and l_2 can be computed as

$$\Delta_{size}^{(1,2)} = |l_1| + |l_2| - |l_1 \cup l_2| = |l_1 \cap l_2|.$$

The larger the intersection of two lists is, the more benefit we can get from combining them.

B. Effects of Combining Lists on Query Performance

In this section we study how query performance will be affected by combining lists. For a similarity query with a string s, if the lists of the grams in $G(s)$ are combined (possibly with lists of grams not in $G(s)$), then the performance of this query can be affected in the following ways:

- Different from the approach of discarding lists, the lower bound T in the T-occurrence problem remains the same, since an answer still needs to appear at least this number of times on the lists. Therefore, if a query was not in a panic case before, then it will not be in a panic case after combining inverted lists.
- The lists will become longer. As a consequence, it will take more time to traverse these lists to find candidates during list merging. In addition, more false positives may be produced to be post-processed.

1) Effects on List-Merging: As inverted lists get combined, some of them will become longer. Here we take one of the list-merging algorithms DivideSkip as an example. In Forumla 4, as some $|l_i|$'s become larger, the time for merging the short lists and the time for verifying the candidates against the long lists could increase. In this sense it appears that combining lists can only increase the list-merging time in query answering. However, the following observation opens up opportunities for us to further decrease the list-merging time, given an index structure with combined lists.

We notice that a gram could appear in the query string s multiple times (with different positions), thus these grams share common lists. In the presence of combined lists, it becomes possible for even different grams in $G(s)$ to share lists. This sharing suggests a way to improve the performance of existing list-merging algorithms for solving the T-occurrence problem [27], [20]. A simple way to use one of these algorithms is to pass it a list for each gram in $G(s)$. Thus we pass $|G(s)|$ lists to the algorithm to find string ids that appear at least T times on these (possibly shared) lists. We can improve the performance of the algorithm as follows. We first identify the shared lists for the grams in $G(s)$. For each *distinct* list l_i, we also pass to the algorithm the number of grams sharing this list, denoted by w_i. Correspondingly, the algorithm needs to consider these w_i values when counting string occurrences. In particular, if a string id appears on the list l_i, the number of occurrences should increase by w_i, instead of "1" in the traditional setting. In this way, we can reduce the number of lists passed to the algorithm, thus possibly even reducing its running time. The algorithms in [27] already consider different list weights, and the algorithms in [20] can be modified slightly to consider these weights.[3]

2) Effects on Post-processing: After combining two lists l_1 and l_2, more candidates will be generated for post-processing. The problem becomes computing the number of candidates generated from the list-merging algorithm. [CHECK]We notice that although different list-merging algorithms can have different performances, the results of the T-occurrence problem are exactly the same, given the same index structure. Therefore we can use any list-merging algorithm to compute the number of candidates. Before combining any lists, the candidate set generated from a list-merging algorithm contains all correct answers and some false positives. We are particu-

[3]Interestingly, our experiments showed that, even for the case we do not combine lists, this optimization can already reduce the running time of existing list-merging algorithms by up to 20% for a data set with duplicate grams in a string.

larly interested to know how many new false positives will be generated by combining two lists l_1 and l_2. The ISC algorithm described in Section IV-A.4 can be modified to adapt to this setting.

Recall that in the ISC algorithm, a ScanCount vector is maintained for a query Q to keep track of the number of grams Q shares with each string id in the data set. The strings whose corresponding values in the ScanCount vector are at least T will be candidate answers. By combining two lists l_1 and l_2, the lists of those grams that are mapped to l_1 or l_2 will be conceptually extended. Figure 8 describes the intuition of the algorithm. Every gram that was previously mapped to l_1 or l_2 will now be mapped to $l_1 \cup l_2$. The extended part of l_1 is $ext(l_1) = l_2 \backslash l_1$. Let $w(Q, l_1)$ denote the number of times grams of Q reference l_1. The ScanCount value of each string id in $ext(l_1)$ will be increased by $w(Q, l_1)$. Since for each reference, all string ids in $ext(l_1)$ should have their ScanCount value increased by one, the total incrementation will be $w(Q, l_1)$ (*not* $w(Q, l_2)$). The same operation needs to be done for $ext(l_2)$ symmetrically. It is easy to see the ScanCount values are monotonically increasing as lists are combined. The strings whose ScanCount values increase from below T to at least T become new false positives after l_1 and l_2 are combined.

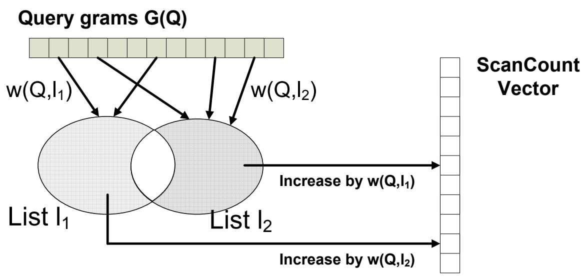


Fig. 8. ISC algorithm for computing the number of candidates for a query Q after combining lists l_1 and l_2.

Figure 9 shows an example, in which $l_1 = \{0, 2, 8, 9\}$, $l_2 = \{0, 2, 3, 5, 8\}$. Before combining l_1 and l_2, two grams of Q are mapped to l_1 and three grams are mapped to l_2. Therefore, $w(Q, l_1) = 2$ and $w(Q, l_2) = 3$. For every string id in $ext(l_1) = \{3, 5\}$, their corresponding values in the ScanCount vector will be increased by $w(Q, l_1)$. Let C denote the ScanCount vector. $C[3]$ will be increased from 6 to 8, while $C[5]$ will be increased from 4 to 6. Given the merging threshold $T = 6$, the change on C[5] indicates that string 5 will become a new false positive. The same operation is carried out on $ext(l_2)$.

C. Algorithms for Choosing Lists to Combine

Based on the previous analysis, we present our algorithms for choosing inverted lists to combine.

1) Basic algorithm: The basic algorithm consists of two steps: discovering candidate gram pairs that are correlated, and selecting some of them to combine.

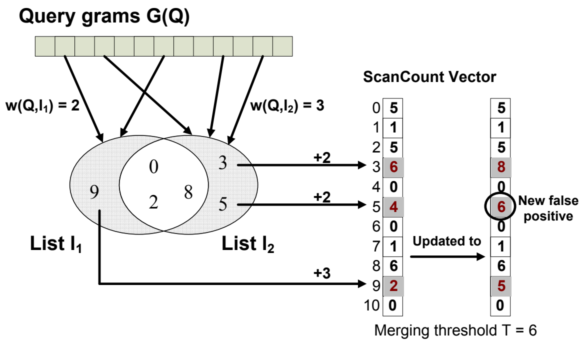


Fig. 9. Example of ISC for computing new false positives after combining lists l_1 and l_2.

Step 1: Discovering Candidate Gram Pairs We are only interested in combining *correlated* lists. Jaccard similarity is used to measure the correlation of two lists. It is defined as:

$$Corr(l_1, l_2) = \frac{|l_1 \cap l_2|}{|l_1 \cup l_2|}. \quad (5)$$

Two lists are considered for combining only if their correlation is greater than a threshold. Clearly it is computationally prohibitive to consider all pairs of grams. Here we present two efficient methods for generating such pairs.

- *Using Adjacent Grams*: We only consider pairs of *adjacent* grams in the strings. If we use q-grams to construct the inverted lists, we can just consider those $(q + 1)$-grams. Each such gram corresponds to a pair of q-grams. For instance, if $q = 3$, then the 4-gram `tion` corresponds to the pair $(\texttt{tio}, \texttt{ion})$. For each such adjacent pair, we treat it as a candidate pair if the Jaccard similarity of their corresponding lists (Equation 5) is greater than a predefined threshold.
- *Using Locality-Sensitive Hashing*: The above approach cannot find strongly correlated grams that are not adjacent in strings. In the literature there are efficient techniques for finding strongly correlated pairs of lists. One of them is called Locality-Sensitive Hashing (LSH) [13]. Using a small number of so-called MinHash signatures for each list, we can use LSH to find those gram pairs whose lists satisfy the above correlation condition with a high probability.

Step 2: Selecting Candidate Pairs to Combine

The second step is selecting candidate pairs to combine. We iteratively pick gram pairs and combine their lists if their correlation satisfies the threshold. Notice that each time we process a new candidate gram pair, since the list of each of them could have been combined with other lists, we still need to verify their (possibly new) correlation before deciding whether we should combine them. After processing all these pairs, we check if the index size meets a given space constraint. If so, the process stops. Otherwise, we decrease the correlation threshold and repeat the process above, until the new index size meets the given space constraint.

This basic algorithm blindly chooses candidate pairs to combine without considering the consequence on the overall query performance. Based on our previous discussions, we propose a new algorithm to wisely choose lists to combine in the second step.

2) Cost-Based Algorithm: Figure 10 shows the cost-based algorithm which takes the estimated cost of a query workload into consideration when choosing lists to combine. It iteratively selects pairs to combine, based on the space saving and the impact on the average query performance of a query workload $\mathcal{Q}$. The algorithm keeps selecting pairs to combine until the total size of the inverted lists meets a given space constraint B. For each gram pair (g_i, g_j), we need to get their current corresponding lists, since their lists could have been combined with other lists (lines 3 and 4). We check whether these two lists are the same list as reference (line 5), and also whether their correlation is above the threshold (line 6). Then we compute the size reduction (line 8) and estimate the average query time difference based on Formula 4 and the ISC algorithm (line 9), based on which we decide the next list pair to combine (lines 10 and 11).

```
Algorithm: CombineLists
Input: Candidate gram pairs $P = \{(g_i, g_j)\}$
         Constraint $B$ on the total list size
         Query workload $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$
Output: Combined lists.
Method:
1.   WHILE ((expected total index size) $> B$ ) {
2.     FOR (each gram pair $(g_i, g_j) \in P$) {
3.       $l_i$ = current list of $g_i$
4.       $l_j$ = current list of $g_j$
5.       if ($l_i$ and $l_j$ are the same list as reference
6.         or $corr(l_i, l_j) < \delta$)
7.         remove $(g_i, g_j)$ from $P$ and continue
8.       Compute size reduction $\Delta_{size}^{(l_i,l_j)}$ if combining $l_i$, $l_j$
9.       Compute difference of average query time $\Delta_{time}^{(l_i,l_j)}$
           for queries in $\mathcal{Q}$ if combining $l_i$, $l_j$
     }
10.    Use $\Delta_{size}^{(l_i,l_j)}$'s and $\Delta_{time}^{(l_i,l_j)}$'s of the gram pairs to decide
         which pair to combine
11.    Combine the two lists $l_i$ and $l_j$ based on the decision
12.    Remove the combined gram pair from $P$
   }
```

Fig. 10. CombineLists algorithm to select gram pairs to combine.

We can use similar optimization techniques as described in IV to improve the performance of CombineLists.

VI. Experiments

In this section, we report our experimental results of the proposed techniques on three real data sets.

- **IMDB Actor Names**: It consists of the actor names downloaded from the IMDB website[4]. There were 1,199,299 names. The average string-length was 17 characters.
- **WEB Corpus Word Grams**: This data set[5], contributed by Google Inc., contained word grams and their observed frequency counts on the Web. We randomly chose 2 million records with a size of 48.3MB. The number of words of a string varied from 3 to 5. The average string-length was 24.
- **DBLP Paper Titles**: It includes paper titles downloaded from the DBLP Bibliography site[6]. It had 274,788 paper titles. The average string-length was 65.

For all experiments the gram length q was 3, and the inverted-list index was held in main memory. Also, for the cost-based DiscardLists and CombineLists approaches, by doing sampling we guaranteed that the index structures of sample strings fit into memory. We used the DivideSkip algorithm described in [20] to solve the T-occurrence problem due to its high efficiency. From each data set we used 1 million strings to construct the inverted-list index (unless specified otherwise). We tested query workloads using different distributions, e.g., a Zipfian distribution or a uniform distribution. To do so, we randomly selected 1,000 strings from each data set and generated a workload of 10,000 queries according to some distribution. We conducted experiments using edit distance, Jaccard similarity, and cosine similarity. We mainly focused on the results of edit distance (with a threshold 2). We report additional results of other functions in Section VII-B. All the algorithms were implemented using GNU C++ and run on a Dell PC with 2GB main memory, and a 3.4GHz Dual Core CPU running the Ubuntu operating system.

A. Evaluating the Carryover-12 Compression Technique

We evaluated the performance of the Carryover-12 based compression technique discussed in Section III. We used a segment size of 128 4-byte integers, which was a good value for the performance and compression ratio, and examined the query performance with different cache sizes. Figure 11 shows that query performance benefits from using a larger cache. On the WebCorpus data set, when we used no cache the average query time was 64.4ms, which is more than 8 times the average query time with a cache of 5000 slots. Since the whole purpose of compressing inverted list is to save space, it is contradictory to improve query performance by increasing the cache size indefinitely. Figures 11(a) and 11(b) show that 20,000 is a reasonable number of cache slots (approximately 10MB), therefore we use this number slots for experiments with Carryover-12 throughout this section.

B. Evaluating the DiscardLists Algorithm

In this section we evaluate the performance of the DiscardLists algorithm for choosing inverted-lists to discard. In addition to the three basic methods to choose lists to discard (LongList, ShortList, RandomList), we also implemented the following cost-based methods:

- PanicCost: In each iteration we discard the list with the smallest ratio between the list size and the number of

[4]www.imdb.com

[5]www.ldc.upenn.edu/Catalog, number LDC2006T13

[6]www.informatik.uni-trier.de/~ley/db

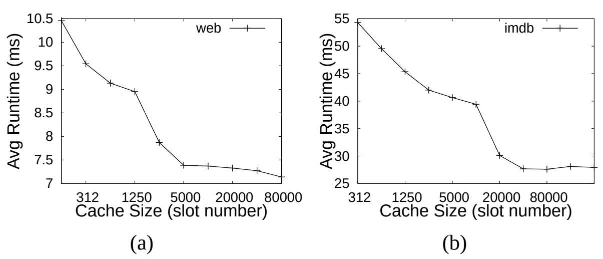


Fig. 11. Benefits of using cache when decoding compressed lists.

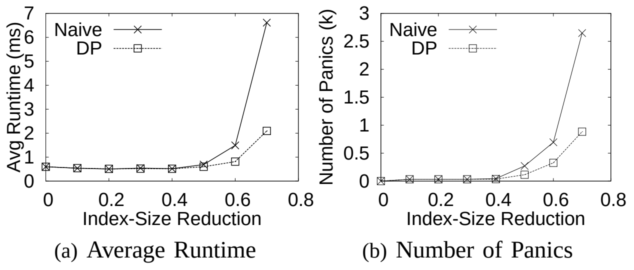


Fig. 12. Benefits of using dynamic programming to tighten bounds.

additional panic cases. Another similar approach, called PanicCost$^+$, discards the list with the smallest number of additional panic cases, disregarding the list length.

- TimeCost: It is similar to PanicCost, except that we use the ratio between the list size and the total time effect of discarding a list (instead of the number of additional panics). Similarly, an approach called TimeCost$^+$ discards the list with the smallest time effect.

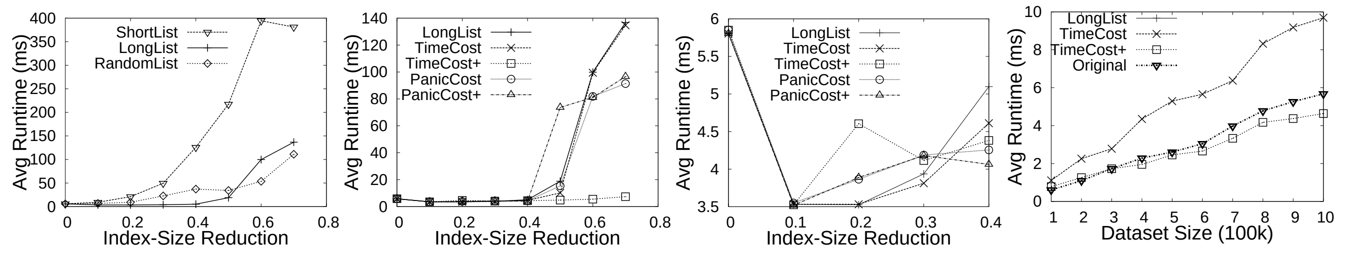
The index-construction time mainly consisted of two major parts: selecting lists to discard and generating the final inverted-list structure. The time for generating samples was negligible. For the LongList, ShortList, and Random approaches, the time for selecting lists to discard was small, whereas in the cost-based approaches the list-selection time was prevalent. In general, increasing the size-reduction ratio also increased the list-selection time. For instance, for the IMDB dataset, at a 70% reduction ratio, the total index-construction time for the simple methods was about half a minute. The construction time for PanicCost and PanicCost$^+$ was similar. The more complex TimeCost and TimeCost$^+$ methods needed 108s and 353s, respectively.

Benefits of Tighter Bounds: We first examined the effects of using the dynamic programming algorithm for computing a tighter bound in the presence of hole lists, as described in Section IV-A.1. We used the DBLP data set, and ran a workload of 10,000 queries with a Zipfian distribution. We used the LongList method to reduce the index size. Figure 12(a) shows the average running time for both the naive method for computing a bound using Equation 2 (marked as "Naive"in the figure) and the dynamic programming algorithm for computing a bound (marked as "DP"). The x-axis is the total memory reduction ratio, where "0" means no size reduction, while the y-axis is the average query time. We see that the dynamic programming approach indeed computed a tighter bound, which notably reduced the average query time. This benefit is largely due to the decrease in the number of panics, as illustrated in Figure 12(b). In the interest of brevity we omitted the results for the other data sets, since they show a very similar trend.

Comparing Simple Methods with Cost-Based Methods: Next we compare the the simple methods for choosing lists to discard with the cost-based ones. For each data set, we conducted experiments by indexing one million strings and running a workload of 10,000 queries with a Zipfian distribution. For those cost-based approaches, we used a sampling ratio of 0.1% for the data strings and a ratio of 25% for the queries.

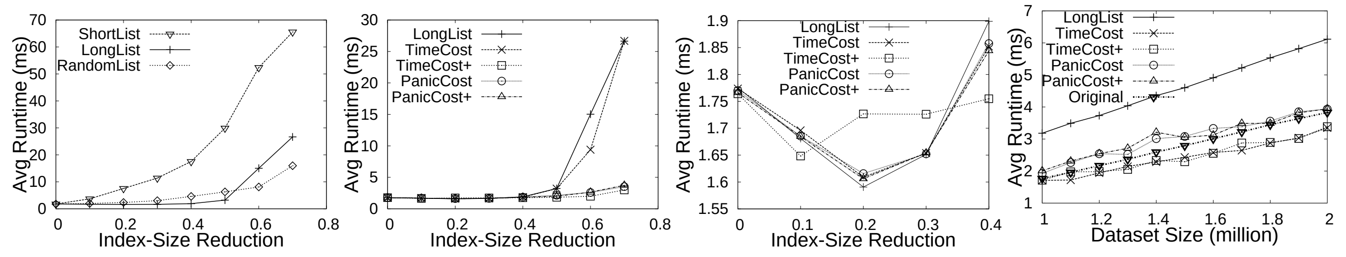
We first considered the three simple methods, namely LongList, ShortList, Random. Figs. 13(a) and 14(a) show how discarding lists affected the average query performance as we increased the index-size-reduction ratio. (Due to space limitation, we do not show the results of the DBLP data set, which were consistent with the results of the other two data sets.) We can see that LongList and Random methods are equally promising, providing compression ratios up to 50% with little penalty in the query execution time. For instance, in Fig. 13(a), when there was no index reduction, the original average query time was 5.8ms. When the reduction ratio was 50%, the running time increased to 18.8ms for the LongList method. For all the data sets, the ShortList method performed significantly worse than the other two. The main reason is that it caused a dramatic increase in the merge-time and post-processing time. For higher reduction ratios, RandomList sometimes outperformed LongList because the former avoided more panics, alleviated in LongList by discarding lists of popular grams.

Based on these results, we chose to omit ShortList and Random in the following experiments.
Figures 13(b) and 14(b) show the benefits of employing the cost-based methods to select lists to discard. Most noteworthy of which is the TimeCost$^+$ method, which consistently delivered the best query performance. As shown in Fig. 13(b), the method achieved a 70% reduction ratio while increasing the query processing time from the original 5.8ms to 7.4ms only. All the other methods increased the time up to at least 96ms for that reduction ratio. Notice that TimeCost$^+$ ignored the list size when selecting a list to discard. The good results may seem counter-intuitive. A deeper analysis suggests that there is really no obvious need to take the list size into account when choosing lists to discard, since the effects of a list on query performance may change significantly in the next iteration, which could be very irrelevant to its size. Thus our main objective should be minimizing the penalty in query performance, regardless of list sizes. We see the TimeCost method behaved similarly to LongList. The reason is that the former preferred discarding long lists over short lists because it tries to maximize the ratio of the list-size and the estimated running time of the query workload. The PanicCost and PanicCost$^+$ methods performed well on the



(a) Simple methods (b) Cost-based methods (c) Improving query performance (d) Scalability

Fig. 13. Reducing index size by discarding lists (IMDB Actors).



(a) Simple methods (b) Cost-based methods (c) Improving query performance (d) Scalability

Fig. 14. Reducing index size by discarding lists (WebCorpus word grams).

DBLP and WebCorpus data sets, because at higher reduction ratios the cost for panics became prevalent. The two panic-based methods performed poorly in Fig. 13(b) because they blindly minimized the number of panics while drastically increasing postprocessing time. TimeCost$^+$ over-topped all the other methods because it can balance the merging time, post-processing time, and scan time.

Surprising Improvement on Performance: Figs. 13(c) and 14(c) show more details in the previous set of figures when the reduction ratio was smaller (less than 40%). A surprising finding is that, for low to moderate reduction ratios, discarding lists could even improve the query performance! Take Fig. 13(c) for an example. All the methods reduced the average query time from the original 5.8ms to 3.5ms for a 10% reduction ratio. The main reason of the performance improvement is that by discarding long lists we can help list-merging algorithms solve the T-occurrence problem more efficiently. We see that significantly reducing the number of total list-elements to process can overcompensate for the decrease in the threshold.

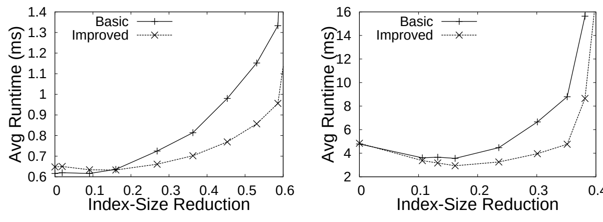
Scalability: For each data set, we increased its number of strings, and used 50% as the index-size-reduction ratio. Figs. 13(d) and 14(d) show that the TimeCost$^+$ method performed consistently well, even outperforming the corresponding uncompressed indexes (indicated by "original"). In the Figure 13(d), in which at 100,000 data strings the average query time increased from 0.61ms to 0.78ms for TimeCost$^+$. As the data size increased, TimeCost$^+$ began outperforming the uncompressed index. For example, at 1 million strings the average query time decreased from 5.7ms (original) to 4.6ms (TimeCost$^+$). This benefit can be attributed to the skewed Zipfian query distribution this algorithm adapts to. Notice that the TimeCost method behaves predictably, albeit badly, whereas the two panic-based methods can behave unpredictably because the merge operation becomes increasingly expensive with a growing data set. Therefore, we excluded the curves for PanicCost and PanicCost$^+$ from 14(d) since they deliver no clear trend.

Summary: (1) Discarding lists can achieve a significant index size reduction without significantly increasing query running time. (2) For small reduction ratios, we can even improve query performance. (3) For most cases the TimeCost$^+$ method can achieve a good query performance.

C. Evaluating the CombineLists Algorithm

We evaluated the performance of the CombineLists algorithm on the same three data sets. In step 1, we implemented three methods to generate candidate list pairs: (1) Adjacent: Consider $(q+1)$-grams. (2) LSH: Use LSH to find strongly correlated pairs (with a high probability). We generated 100 MinHash signatures for each list. (3) Adjacent+LSH: We take the union of the candidate pairs generated by these two methods. In step 2, we implemented both the CombineBasic and the CombineCost algorithms for iteratively selecting list pairs to combine.

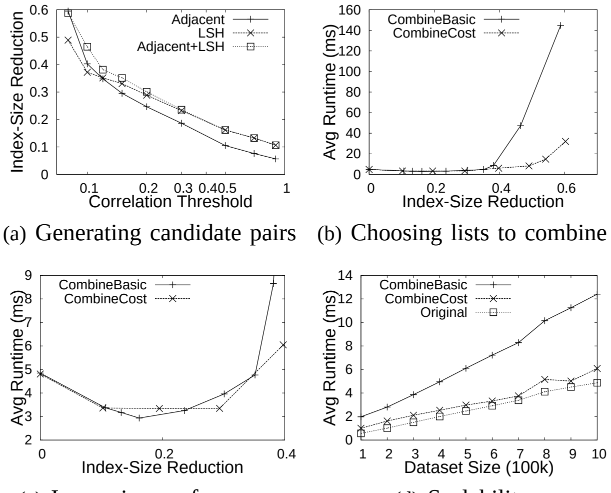
Benefits of Improved List-Merging Algorithms: We first evaluated the benefits of using the improved list-merging algorithms to solve the T-occurrence problem for queries on combined inverted lists, as described in Section V-B. As an example, we compared the DivideSkip algorithm in [20] and its improved version that considers duplicated inverted lists in a query. We used the Adjacent+LSH method to generate candidate list pairs, and the CombineBasic algorithm to select lists to combine. Figure 15 shows the average running time for the basic DivideSkip (marked as “Basic”) and the improved DivideSkip algorithm (marked as “Improved”). We can see that when the reduction ratio increased, more lists were combined, and the improved algorithm did reduce the average query time.



(a) DBLP titles (b) IMDB actors

Fig. 15. Reducing query time using improved list-merging algorithm (DivideSkip).

Generating Candidate List Pairs: Figures 16(a) and 17(a) compare the index size reduction ratio for different candidate-generating methods, over various correlation threshold δ, on the two data sets. We used the CombineBasic method for combining lists. The reduction ratio was plotted in a logarithmic scale. We observed that on the IMDB and DBLP data sets, with a higher correlation threshold, LSH produced more candidate pairs than Adjacent, since the former can find correlated pairs of grams that are not adjacent. The results also show that a significant number of correlated pairs were indeed from adjacent grams. As we decreased the correlation threshold, LSH was less capable of discovering all qualified candidate pairs without using more MinHash signatures. The results of the Adjacent+LSH were better than the two previous methods, since it considered more candidate list pairs.

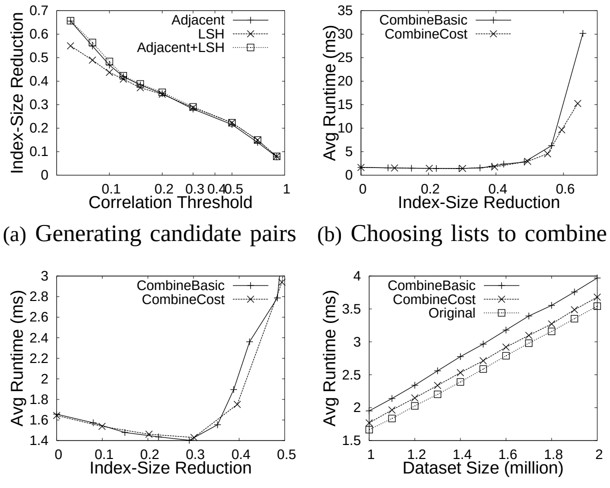


(a) Generating candidate pairs (b) Choosing lists to combine

(c) Improving performance (d) Scalability

Fig. 16. Reducing index size by combining lists (IMDB Actors).

Choosing Lists to Combine: We compared the CombineBasic algorithm with the cost-based CombineCost algorithm for choosing lists to combine, and the results are shown in Figure 17(b) and 16(b). The average query time was plotted



(a) Generating candidate pairs (b) Choosing lists to combine

(c) Improving performance (d) Scalability

Fig. 17. Reducing index size by combining lists (WebCorpus word grams).

over different reduction ratios for both algorithms. We observe that on all three data sets, the query running time for both algorithms increased very slowly as we increased the index size reduction ratio, until about 40% to 50%. That means, this technique can reduce the index size without increasing the query time! As we further increased the index size reduction, the query time started to increase. For the CombineCost algorithm, the time increased slowly, especially on the IMDB data set. The reason is that this cost-based algorithm avoided choosing bad lists to combine, while the CombineBasic algorithm blindly chose lists to combine. The difference between these two algorithms on the IMDB data set was larger than these two data sets, because the CombineBasic algorithm caused queries to spend a lot of post-processing time on this data set.

Figures 16(c) and 17(c) show that when the reduction ratio is less than 40%, the query time even decreased. This improvement is mainly due to the improved list-merging algorithms. Figures 16(d) and 17(d) show how the algorithms of combining lists affected query performance as we increased the data size, for a reduction ratio of 40%.

Summary: (1) Combining lists can also achieve a significant index size reduction without increasing query running time too much. (2) For small reduction ratios, combining lists can also improve query performance, attributed to the improved list-merging algorithms. (3) For most cases the cost-based CombineCost algorithm gives a better index structure than the CombineBasic algorithm.

D. Comparing Different Compression Techniques

We compared the cost-based DiscardLists and CombineLists techniques with Carryover-12 (see Section III) and VGRAM. Since Carryover-12 and VGRAM do not allow explicit control of the compression ratio, for each of them we reduced the size of the inverted-list index to determine their compression ratio. Also, we compressed the inverted-list index using DiscardLists and CombineLists at the same compression ratio.

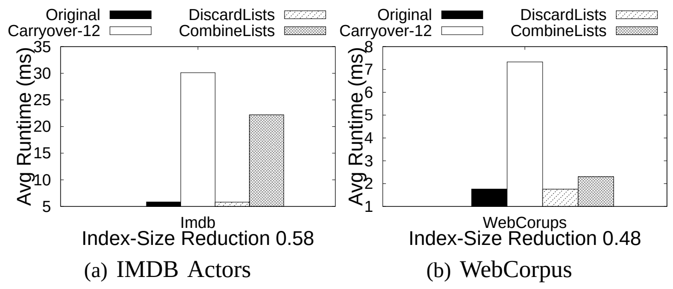


Fig. 18. Comparison of Carryover-12 with DiscardLists and CombineLists at the same reduction ratio.

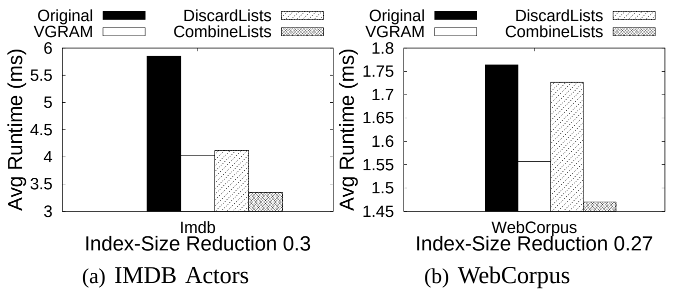


Fig. 19. Comparison of VGRAM with DiscardLists and CombineLists at the same reduction ratio.

Comparison with Carryover-12: Figures 18(a) and 18(b) compare the performance of our techniques with the Carryover-12 compression. For Carryover-12, to achieve a good balance between the query performance and the index-sizen, we use fixed size segments of 128 4-byte integers and a synchronization point for each segment. The cache contained 20,000 segment slots (approximately 10MB). The compression ratio achieved by Carryover-12 was 58% for the IMDB dataset and 48% for the WebCorpus dataset. We see that the online decompression of Carryover-12 has a profound impact on performance. It increased the average running time from an original 5.85ms to 30.1ms for the IMDB dataset, and from an original 1.76ms to 7.32ms for the WebCorpus dataset. The CombineLists performed significantly better at 22.15ms for the IMDB dataset and 2.3ms for the WebCorpus dataset. The DiscardLists could even slightly decrease the running time compared to the original index to 5.81ms and 1.75ms for the IMDB and WebCorpus dataset, respectively.

Comparison with VGRAM: Figures 19(a) and 19(b) compare the performance of our techniques with VGRAM. We set its $qmin$ parameter to 4. We did not take into account the memory requirement for the dictionary trie structures because it was negligible. The compression ratio was 30% for the IMDB dataset and 27% for the WebCorpus dataset. Interestingly, all methods could outperform the original, uncompressed index. As suspected, VGRAM can considerably reduce the running time for both datasets. For the IMDB dataset it could reduce the time from an original 5.85ms to 4.02ms and for the WebCorpus dataset from 1.76ms 1.55ms. Suprisingly, the CombineLists algorithm could reduce the running time even more than VGRAM to 3.34ms for the IMDB dataset and to 1.47ms for the WebCorpus dataset. The DiscardLists performed competitively for the IMDB dataset at 3.93ms and slighlty faster than the original index (1.67ms) on the WebCorpus dataset.

Summary: (1) CombineLists and DiscardLists can significantly outperform Carryover-12 at the same memory reduction ratio because of the online decompression required by Carryover-12. (2) For small compression ratios CombineLists performs best, even outperforming VGRAM. (3) For large compression ratios DiscardLists delivers the best query performance. (4) While Carryover-12 can achieve reuductions up to 60% and VGRAM up to 30%, neither allows explicit control over the reduction ratio. DiscardLists and CombineLists offer this flexibility with good query performance.

E. Changing Query Workloads

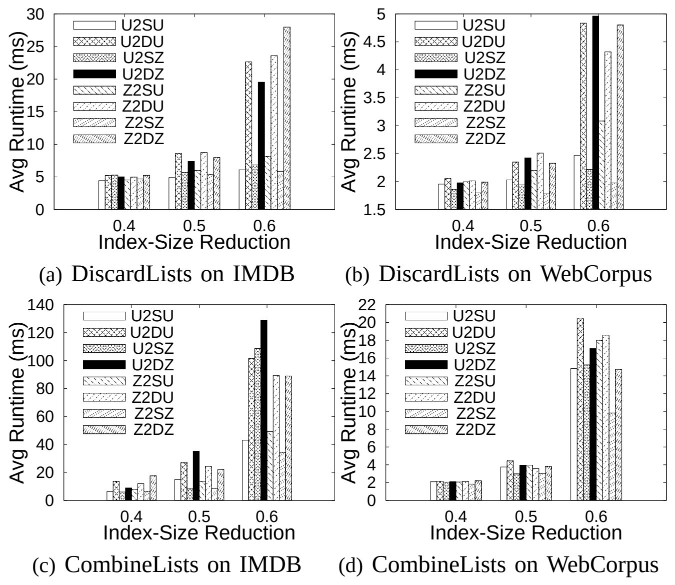


Fig. 20. Performance of DiscardLists and CombineLists on changing workloads

The cost-based methods for discarding and combining lists assume a query workload. We experimentally evaluated their performance in the presence of changing query workloads. That is, we build an inverted-list index assuming one query workload and measured the performance of another query workload. We shall denote the set of queries used to build the index as the *training workload* and the set of queries whose time we measured as the *testing workload*. For the testing workload, we consider that it can differ from the training workload in the distinct query strings, and in the distribution of the distinct queries in the workload. To explain this further, let us examine the legends in Fig. 20: The abbreviation scheme reflects a) the distribution of the training workload, U for uniform and Z for Zipfian, b) whether the running workload consisted of the same distinct queries (S) as the training workload or entirely different distinct queries (D) and c) the distribution of the running workload (the last U or Z). For example "U2SZ" means we used a uniformly distributed query workload as a training workload, and generated a testing workload using the same distinct queries as in the training workload, but in a Zipfian distribution. Consider another example, "U2DZ". This means our training workload was a uniformly distributed query workload and the testing workload was generated from entirely different distinct queries to form a Zipfian distribution. Our query workloads consisted of 10,000 queries generated from 1,000 distinct queries. Figs. 20(a) and 20(b) depict the average query performance of various workloads on indexes compressed by discarding lists. Not surprisingly, as the reduction ratio increases so does the variance in performance among the different workloads. For example, on the IMDB dataset at a 40% reduction ratio, Z2SZ took an average of 4.69ms per query versus a 4.97ms for Z2DU, while at a 60% reduction ratio the former increased to 5.85ms and the latter to 23.6ms. Interestingly, the experiments show that a change in both the distribution and the queries does not always present the worst case for performance. Consider the IMDB dataset at a 60% reduction ratio where one might suspect that U2DZ perform worse than U2DU. The former yielded an average of 7.38ms and the latter 8.56ms. Notice that for both the IMDB and the WebCorpus datasets the worst performance achieved by changing the workloads is still competitive to a Carryover-12 compressed index. Take the WebCorpus dataset, where at a 50% reduction, Z2DU presents the worst change to 2.5ms while Carryover-12 needs an average of 7.32ms at a 48% reduction ratio.

Combining Lists: Figs. 20(a) and 20(b) show the average query performance of various workloads on indexes compressed by combining lists. Again, the variance in performance increases as the compression ratio increases. For example on the WebCorpus dataset, at a 40% reduction ratio U2SU needed 2.09ms per query and "U2DZ" 2.08ms, while at a 60% ratio the former increased to 14.82ms and the latter to 17.06ms. Overall, combining lists does not seem to be as sensitive to changes in the workload as discarding lists. This is because discarding lists can cause panic cases (requiring a scan) while combining lists cannot. Interestingly, on the WebCorpus dataset combining lists can outperform the Carryover-12 technique but not on the IMDB dataset. For the WebCorpus dataset at a 50% reduction ratio, the worst change in workload (U2DU) amounted to 4.43ms per query, while at a 48% reduction ratio Carryover-12 required 7.32ms.

Summary: (1) Not surprisingly, the cost-based methods of combining and discarding lists are sensitive to how representative the chosen workload is. (2) The method of combining lists is less sensitive to changes in workloads than discarding lists, but it performs worse than discarding lists at high compression ratios. (3) Even for the worst changes in workloads that we tested, our techniques can mostly outperform Carryover-12 at similar compression ratios.

VII. Discussion

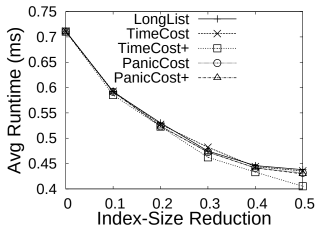
A. Index Size Reduction with Filters

Various filtering techniques have been proposed to prune strings that cannot be similar enough to a query string [9], [6], [20]. In general, a filter can partition strings into different disjoint groups. There are different ways to adopt our size-reduction techniques in the presence of filters. Let $P_1, \ldots, P_n$ be the groups of the strings generated by filters. (1) *Global reduction*: We use the lists of the entire data set to decide what lists to discard or combine, then apply these decisions to the gram lists within each group P_i. (2) *Local reduction*: We first decide how much reduction to allocate to each group P_i in order to achieve a global reduction ratio. Within each group G_i, we decide the lists to discard or combine, and different groups have different decisions. Our cost-based size-reduction methods can benefit from this divide-and-conquer strategy in two ways: (a) Running the DiscardLists or CombineLists algorithm for each group can be faster than running them once on the entire data set. (b) The performance improvement can allow a higher sampling-ratio, potentially enhancing the quality of the results. Notice that the query workload may not be evenly distributed among all the groups. For the local policy, the main problem becomes distributing a space constraint over all the groups such that the average query performance is maximized. The problem of deciding this size allocation needs future research.

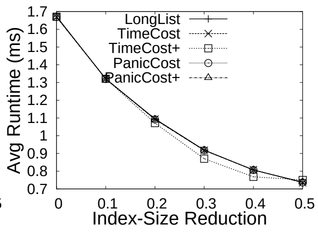
B. Extension to Other Similarity Functions

Our discussion so far mainly focused on the edit distance metric. We now generalize the results to the following commonly used similarity measures: Jaccard similarity and cosine similarity. To reduce the size of inverted lists based on those similarity functions, the main procedure of algorithms DiscardLists and CombineLists remains the same. The only difference is that in DiscardLists, to compute the merging threshold T for a query after discarding some lists, we need to subtract the number of hole lists for the query from the formulas proposed in [20]. In addition, for the estimation of the post-processing time, we also need to replace the estimation of the edit distance time with that of Jaccard and cosine time respectively. Figure 21 shows the average running time for the DBLP data using variants of the TimeCost$^+$ algorithm for these two functions. The results on the other two data sets were similar and not included here. We see that the average running time continuously decreased when the reduction ratio increased to up to 40%. For example, at a 40% reduction ratio for the cosine function, the running time decreased from 1.7ms to 0.8ms.

The performance started degrading at a 50% reduction ratio and increased rapidly at a ratio higher than 60%. For a 70% reduction ratio, the time for the Cosine and Jaccard functions increased to 150ms and 115ms for LongList. Also, for high reduction ratios the TimeCost and TimeCost$^+$ methods became worse than the panic-based methods, due to the inaccuracy in estimating the merging time. Note that the Cosine and Jaccard



(a) Jaccard function



(b) Cosine function

Fig. 21. Reducing index size based on Jaccard and Cosine similarity functions (DBLP titles)

functions are expensive to compute, therefore the punishment (in terms of post-processing time) for inaccurately estimating the merging time can be much more severe than that for the edit distance.

C. Integrating Several Approaches

So far, we have studied traditional list-compression techniques, our new methods based on discarding lists and combining lists, and the VGRAM technique independently to reduce the index size. Since these methods are indeed orthogonal, we could even use any combination of them to further reduce the index size and/or improve query performance. As an example, we integrated CombineLists with Carryover-12.

We first compressed the index using CombineLists approach with a reduction α, and then applied Carryover-12 on the resulting index. We varied α from 0 (no reduction for CombineLists) to 60% in 10% increments. The results of the overall reduction ratio and the average query time are shown in the "CL+Carryover-12" curve in Figure 22. The leftmost point on the curve corresponds to the case where $\alpha = 0$. For comparison purposes, we also plotted the results of using the CombineLists alone shown on the other curve. The results clearly show that using both methods we can achieve high reduction ratios with a better query performance than using CombineLists alone. Consider the first point that only uses Carryover-12. It could achieve a 48% reduction with an average query time of 7.3ms. By first using CombineLists at a 30# reduction ratio (4th point on the curve) we could achieve a higher reduction ratio (61%) at a lower average query time (6.34ms).

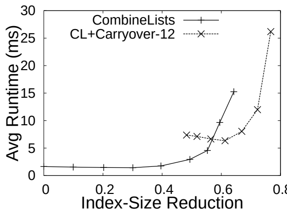


Fig. 22. Reducing index size using CombineLists with Carryover-12.

One way to integrate multiple methods is to distribute the global memory constraint among several methods. Notice since Carryover-12 and VGRAM do not allow explicit control of the index size, it is not easy to use them to satisfy an arbitrary space constraint. Several open challenging problems need more future research. First, similar to the space allocation problem discussed in Section VII-A, we need to decide how to distribute the global memory constraint among different methods. Second, we need to decide in which order to use them. For example, if we use CombineLists first, then we never consider discarding merged lists in DiscardLists. Similarly, if we run DiscardLists first, then we never consider combining any discarded list in CombineLists.

VIII. Conclusions

In this paper, we studied how to reduce the size of inverted-list index structures of string collections to support approximate string queries. We studied how to adopt existing inverted-list compression techniques to achieve the goal, and proposed two novel methods for achieving the goal: one is based on discarding lists, and one based on combining correlated lists. They are both orthogonal to existing compression techniques, exploit a unique property of our setting, and offer new opportunities for improving query performance. We studied technical challenges in each method, and proposed efficient, cost-based algorithms for solving related problems. Our extensive experiments on real data sets show that our approaches provide applications the flexibility in deciding the tradeoff between query performance and indexing size and can outperform existing compression techniques. An interesting and surprising finding is that while we can reduce the index size significantly (up to 60% reduction) with tolerable performance penalties, for 20-40% reductions we can even improve query performance compared to original indexes.

References

[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

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

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[4] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, 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 Trans. Knowl. Data Eng.*, 18(4):433–447, 2006.

[8] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[9] 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.

[10] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[11] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: Selectivity estimators for set similarity selection queries. In *VLDB*, 2008.

[12] B. Hore, H. Hacigümüs, B. R. Iyer, and S. Mehrotra. Indexing text data under space constraints. In *CIKM*, pages 198–207, 2004.

[13] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC Conference*, 1998.

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

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

[16] 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.

[17] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD Conference*, pages 802–803, 2006.

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

[19] 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.

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

[21] 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.

[22] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2):12, 2007.

[23] M. D. McIllroy. Development of a spelling list. *IEEE Transactions on Communications*, 30(1):91–99, 1998.

[24] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[25] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.

[26] S. C. Sahinalp, M. Tasan, J. Macker, and Z. M. Özsoyoglu. Distance based indexing for string proximity search. In *ICDE*, pages 125–, 2003.

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

[28] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[29] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[30] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, 2008.

[31] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[32] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.