IEEE International Conference on Data Engineering (ICDE 2009),
page 604-615, March, 29- April, 3, 2009, Shanghai, China

Space-Constrained Gram-Based Indexing for Efficient Approximate String Search

Alexander Behm[1] Shengyue Ji[1] Chen Li[1] Jiaheng Lu[2]

[1] Department of Computer Science, University of California, Irvine, CA 92697
[2] Key Laboratory of Data Engineering and Knowledge Engineering, Renmin University of China, China
{abehm,shengyuj,chenli}@ics.uci.edu, jiahenglu@gmail.com

***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, 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 5 strings and the corresponding inverted lists of their 2-grams.

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

(a) Strings.

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

(b) Inverted lists.

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

The algorithms answer a 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 50 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

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.32

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 [31]. 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 decompressed 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 needs to occur at least 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 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], [11] 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. The 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.

Due to space limitations, we leave more results in the 18-page full version of this paper [4].

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], [30], [7], [9] 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], [5], [6], [10], [27], [28], especially in the context of record linkage. Some of them are proposed in the context of relational DBMS systems. Several recent papers focused on approximate *selection* (or search) queries [11], [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. [13] proposed a gram-selection technique for indexing text data under space constraints, mainly considering SQL LIKE queries. Other related studies include [17], [26]. There

are recent studies on the problem of estimating the selectivity of SQL LIKE substring queries [15], [18], and approximate string queries [22], [16], [19], [12].

Recently a new technique called VGRAM [21], [29] was proposed to use variable-length grams to improve approximate-string query performance and reduce the 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 can outperform VGRAM, and potentially they can be integrated with VGRAM to further reduce the index size (Section VI-E).

II. Preliminaries

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 $ed(s, q) \leq k$.

Let Σ be an alphabet. For a string s of the characters in Σ, we use "$|s|$" to denote the length of s. 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. (The results in the paper extend naturally to the case where we do not extend a string to produce grams.) 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'. 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 α =#, β = \$, $q = 3$, 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.

We construct an index as follows. 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). 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:

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

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 [10], [20]. To summarize, the following are the pieces of time needed to answer a query:

- If the lower bound T (called "merging threshold") is positive, the time includes the time to traverse the lists of the query grams to find candidates (called "merging time") and the time to remove the false positives (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 mainly focus on the edit distance function, and the results are extended for other functions as well.

III. Adopting Existing Compression Techniques

There are many techniques [31], [7], [9] on list compression, which mainly study the problem of representing integers on inverted lists efficiently to save storage space. In this section we study how to adopt these techniques to solve our problem and discuss their limitations.

Most of these techniques exploit the fact that ids on an inverted list are monotonically increasing integers. For example, suppose we have a list $l = (id_1, id_2, \ldots, id_n)$, $id_i < id_{i+1}$ for $1 \leq i < n$. If we take the differences of adjacent integers to construct a new list $l' = (id_1, id_2 - id_1, id_3 - id_2, \ldots, id_n - id_{n-1})$ (called the gapped list of l), the new integers tend to be smaller than the original ids. Many integer-compression techniques such as gamma codes, delta codes [7], and Golomb codes [9] can efficiently encode the gapped lists by using shorter representations for small integers. As an example, we study how to adopt one of the recent techniques called Carryover-12 [1].

An issue arises when using the encoded, gapped representation of a list. Many efficient list-merging algorithms in our setting [20] rely heavily on binary search on the inverted lists. 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. This problem can be solved

by segmenting the list and introducing *synchronization points* [24]. Each segment is associated with a synchronization point. Decoding can start from any synchronization point, so that only one segment needs to be decompressed in order to read a specific integer. We can make each segment contain the same number of integers. Since different encoded segments could have different sizes, we can 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.

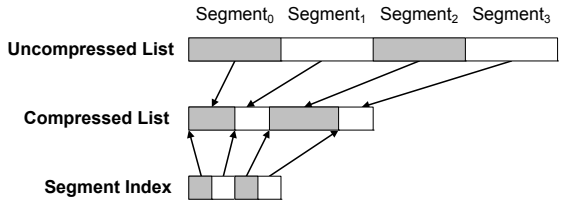


Fig. 2. Inverted-list compression with segmenting and indexing

One way to access elements is by decoding the corresponding segment for each random access. If multiple integers within the same segment are requested, the segment might be decompressed multiple times. The repeated efforts can be alleviated using caching. We allocate a global cache pool for all inverted lists. 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.

Limitations: 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 techniques can still improve query performance since disk I/Os are usually the major cost. When the inverted lists are in memory, these techniques require additional decompression operations, compared to non-compressed indexes. Thus, the query performance can only decrease. These approaches have limited flexibility in trading query performance with space savings. Next we propose two novel methods that do not have these limitations.

IV. Discarding Inverted Lists

In this section we study how to reduce the size of an inverted-list index by discarding some of its 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. 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 (Section IV-B).

A. Effects of Hole Grams on a Query

1) Merging Threshold: Consider a string r in 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 r could have appeared on some of the hole lists. But we do not know on how many hole lists 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 hole grams `irv` and `ine` as shown in Figure 3. Using the formula above, an answer string should share at least 0 nonhole grams with string s, meaning the query can only be answered by a scan. This formula assumes that a single edit operation could potentially destroy 3 grams, and two operations could potentially destroy 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 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. 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 [29] in the context of the VGRAM technique [21]). Our experiments have shown that this algorithm can increase query

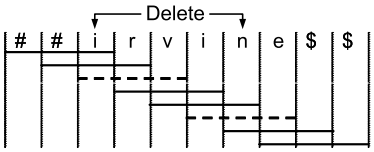


Fig. 3. A query string `irvine` with two hole grams. A solid horizontal line denotes a nonhole gram, and a dashed line denotes a hole gram. The arrows denote character deletions.

performance by tightening the bound. More details about the algorithm and experiments are in [4].

2) List-Merging Time: 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, their running time can only decrease by discarding some lists. Other merging algorithms (e.g., MergeOpt, DivideSkip [20]) separate the inverted lists into a group of long lists and a group of short lists, and process them separately. The performance of these algorithms depends on how the two groups are formed, which is related to T. Thus their performance is sensitive to changes in T. Another class of algorithms such MergeSkip and DivideSkip [20] utilize T to skip irrelevant elements on the lists. Decreasing T by discarding some lists might negatively affect their performance. Meanwhile, we might have fewer lists to process, possibly resulting in an improvement of the query performance.

3) Post-Processing Time: For a given query, introducing hole grams may only increase the number of candidates to post-process if we use Equation 2. Surprisingly, if we use the dynamic programming algorithm to derive a tighter T', then the number of candidates for post-processing might even decrease [4]. 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, as discussed earlier, the new threshold $T' = 2$. After discarding the lists, the string i is not a candidate anymore, since all the lists containing it have been discarded. Thus we can reduce the post-processing cost. Note that any string id which appears only on `irv` and `ine` cannot be an answer to the query and would have been removed from the results during post-processing.

4) Estimating Time Effects on a Query: Since we are evaluating whether it is a wise choice to discard a specific list l_i, we want to know, by discarding list l_i, how the performance of a single query Q will be affected using the indexing structure. We now quantify these effects discussed above by estimating the running time of a query with hole grams. In [4] we discuss how to estimate the merging time and scan time. We focus on 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 the number of candidates multiplied by the 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 [22], [16], [19]. 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 *repeatedly* when choosing lists to discard, and therefore needs to be very 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." 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 many times it occurs on all the inverted lists of the grams in the query and store them in an array C. 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'. We find the new candidates by scanning the array C and recording those positions (corresponding to string ids) whose value is at least T'.

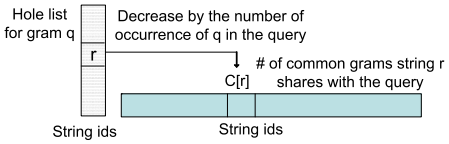


Fig. 4. Intuition behind the Incremental-Scan-Count (ISC) algorithm.

For instance, in Fig. 5, the hole list includes string ids 0, 2, 5, and 9. For each of them, we decrease the corresponding value in the array by 1 (assuming the hole gram occurs 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.

B. Choosing Inverted-Lists to Discard

We now study how to wisely choose lists to discard in order to satisfy a given space constraint. The following are several simple approaches: choosing the longest lists to discard (LongList), choosing the shortest lists to discard (ShortList), or choosing random lists to discard (RandomList). These naive approaches blindly discard lists without considering the

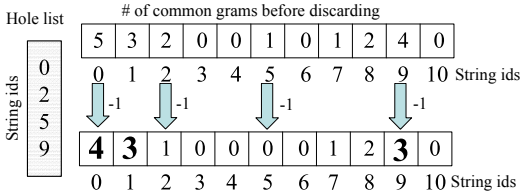


Fig. 5. Running the ISC algorithm ($T' = 3$).

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. 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, ..., l_n}
       Constraint B on the total list size
       Query workload Q = {Q_1, ..., Q_m}
Output: A set D of lists in L that are discarded
Method:
1.  D = ∅;
2.  WHILE (B < (total list size of L)) {
3.      FOR (each list l_i ∈ L) {
4.          Compute size reduction Δ^i_size if discarding l_i
5.          Compute difference of average query time Δ^i_time
                for queries in Q if discarding l_i
        }
6.      Use Δ^i_size's and Δ^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. (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.) In each iteration (line 6), 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) We run the algorithm on a random subset of the strings. As a consequence, (i) we can make sure the entire inverted lists of these sample strings can fit into a given amount of memory. (ii) We can reduce the array size in the ISC algorithm, as well as its scan time to find candidates. (iii) 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 original 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. If we want to use the positional filtering technique (mainly for the edit distance function) described in [10], [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). We propose an algorithm for choosing lists to combine in the presence of space constraints (Section V-C).

A. Data Structures for Combining Lists

In the original inverted-list structure, different grams have different lists. Combining two lists l_1 and l_2 will produce a new list $l_{new} = l_1 \cup l_2$. The size reduction of combining two lists l_1 and l_2 can be computed as

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

All grams that shared l_1 and l_2 (there could be several grams due to earlier combining operations) will now share list l_{new}. In this fashion we can support combining more than two lists iteratively. We use a data structure called Disjoint-Set with the algorithm Union-Find [8] to efficiently combine more than two lists, as illustrated in Figure 7. More details are in [4].

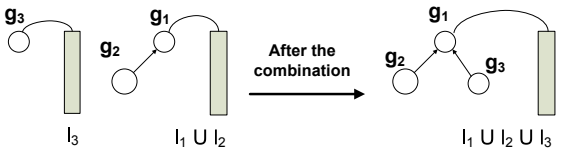


Fig. 7. Combining list of g_2 with list of g_3 using Union-Find

B. Effects of Combining Lists on Query Performance

We study how combining lists affects query performance. 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. (1) 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. (2) The lists will become longer. As a consequence, it will take more time to traverse these lists to find candidates during list merging, and more false positives may be produced to be post-processed.

1) List-Merging Time: As inverted lists get combined, some of them will become longer. 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. Thus 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.[1]

2) Post-processing Time: We want to compute the number of candidates generated from the list-merging algorithm. Before combining any lists, the candidate set generated from a list-merging algorithm contains all correct answers and some false positives. We are particularly 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.

In the algorithm, a ScanCount vector is maintained for a query Q to store the number of grams Q shares with each string id in the collection. 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. Every gram 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.

Figure 8 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 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. Choosing Lists to Combine

We use two steps to combine lists: discovering candidate gram pairs, and selecting some of them to combine.

[1]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%.

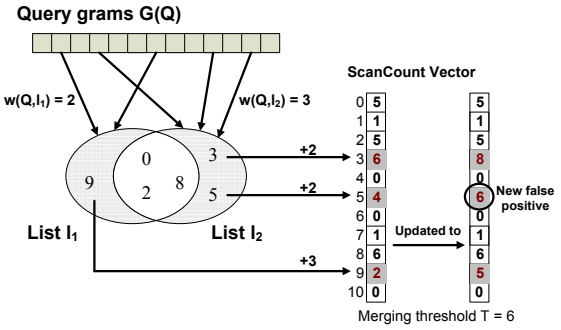


Fig. 8. Example of ISC for computing new false positives after combining lists l_1 and l_2.

Step 1: Discovering Candidate Gram Pairs: We are only interested in combining *correlated* lists. We can use Jaccard similarity to measure the correlation of two lists, defined as $jaccard(l_1, l_2) = \frac{|l_1 \cap l_2|}{|l_1 \cup l_2|}$. Two lists are considered to be combined only if their correlation is greater than a threshold. Clearly it is computationally prohibitive to consider all pairs of grams. There are different ways for generating such pairs. One way is 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 is greater than a predefined threshold. One limitation of this approach is that it 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) [14]. 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. One basic algorithm is the following. 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 does not consider the effect of combining two lists on the overall query performance. We propose a cost-based algorithm to wisely choose lists to combine in the second step. Figure 9 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 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. 9. Cost-based algorithm to select gram pairs to combine.

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

VI. Experiments

We used three real data sets. (1) *IMDB Actor Names*: It consists of the actor names downloaded from the IMDB website (http://www.imdb.com). There were 1,199,299 names. The average string-length was 17 characters. (2) *WEB Corpus Word Grams*: This data set (http://www.ldc.upenn.edu/Catalog, number LDC2006T13) 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. (3) *DBLP Paper Titles*: It includes paper titles downloaded from the DBLP Bibliography site (http://www.informatik.uni-trier.de/∼ley/db). It had 274,788 paper titles. The average string-length was 65.

For all experiments the gram length q was 3, and we applied length filtering [10]. 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 VI-D. 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 OS.

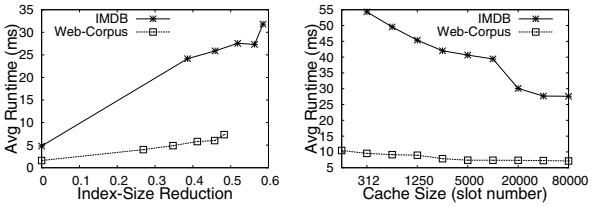
A. Evaluating the Carryover-12 Compression Technique

We adopted the Carryover-12 compression technique as discussed in Section III into our problem setting. We varied the segment size to achieve different index-size reduction ratios. We measured the corresponding query performance. Figure 10(a) shows the results for the IMDB and Web Corpus datasets as the reduction ratio increased. (Notice that the two data sets used different reduction ratios because of the limitation of the technique.) Consider the Web Corpus dataset. The original average query running time (without compression) was about 1.6ms. After compressing the index, the query time increased significantly. For example, when the reduction ratio was about 41%, the query time increased to 5.7ms. The time kept increasing as we compressed the index further.

Figure 10(b) shows how the query time was affected as we increased the cache size. (The cache size was significantly smaller than the compressed index size.) 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 too much. As we allocated more cache to the compressed index, the query time did decrease. Notice that if we allocate enough cache for the entire compressed index, the performance can become almost the same as that of the original index (without considering the cache lookup overhead). As the cache size is typically much smaller than the original index size, the performance should always be worse than the original case due to the online decompression cost.

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. (1) PanicCost: In each iteration we discard the list with the smallest ratio between the list size and the number of additional panic cases. Another similar approach, called PanicCost$^+$, discards the list with the smallest number of additional panic cases, disregarding the list length. (2) 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.



(a) Effect of index compression (b) Effect of caching

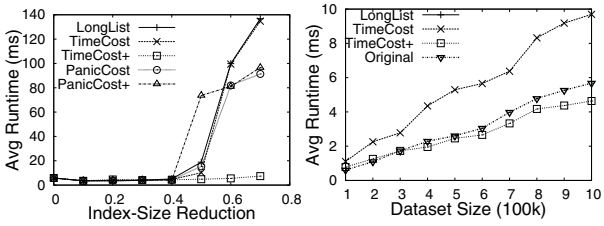
Fig. 10. Carryover-12 compression.

The index-construction time 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 RandomList 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.

Different Methods to Choose Lists to Discard: We first considered the three simple methods, namely LongList, ShortList, RandomList. Experiments [4] showed that in most cases, the LongList method gave us the best query performance, while the RandomList method was the best for high reduction ratios. The ShortList was always the worst.

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. Figure 11(b) shows the benefits of employing the cost-based methods to select lists to discard. Most noteworthy of which is the TimeCost$^+$ method, which consistently delivered good query performance. As shown in Fig. 11(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. TimeCost$^+$ over-topped all the other methods because it can balance the merging time, post-processing time, and scan time.

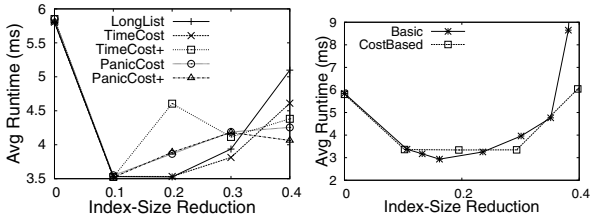
Surprising Improvement on Performance: Figs. 12(a) shows more details when the reduction ratio was smaller (less than 40%). A surprising finding is that, for low to moderate reduction ratios, discarding lists could even improve the query performance! All the methods reduced the average query time from the original 5.8ms to 3.5ms for a 10% reduction ratio. The main reason of the performance improvement is that



(a) Choosing lists to discard (b) Scalability

Fig. 11. Reducing index size by discarding lists (IMDB).

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.



(a) DiscardLists (b) CombineLists

Fig. 12. Improving query performance using two new approaches (IMDB).

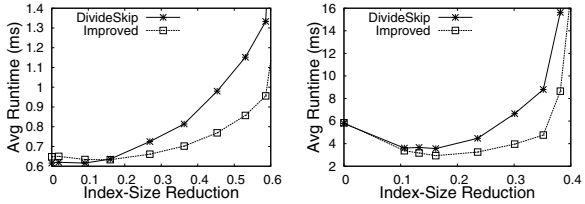
Scalability: For each data set, we increased its number of strings, and used 50% as the index-size-reduction ratio. Fig. 11(d) shows that the TimeCost$^+$ method performed consistently well, even outperforming the corresponding uncompressed indexes (indicated by "original"). At 100,000 data strings, the average query time increased from 0.61ms to 0.78ms for TimeCost$^+$. As the data size increased, TimeCost$^+$ began outperforming the uncompressed index.

C. Evaluating the CombineLists Algorithm

We evaluated the performance of the CombineLists algorithm on the same three data sets. In step 1, we generated candidate list pairs by using both $(q+1)$-grams and LSH. In step 2, we implemented both the basic and the cost-based algorithms for iteratively selecting list pairs to combine.

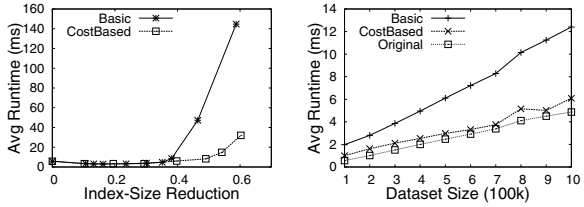
Benefits of Improved List-Merging Algorithms: We first evaluated the benefits of using the improved list-merging algorithms to solve the T-occurrence problem for queries on combined inverted lists, as described in Section V-B. As an example, we compared the DivideSkip algorithm in [20] and its improved version that considers duplicated inverted lists in a query. We used the basic algorithm to select lists to combine. Figure 13 shows the average running time for the algorithm and its improved version (marked as "Improved"). When the reduction ratio increased, more lists were combined, and the improved algorithm did reduce the average query time.

Choosing Lists to Combine: We compared the basic algorithm with the cost-based algorithm for choosing lists to combine, and the results are shown in Figure 14(a). The average query time was plotted over different reduction ratios



(a) DBLP titles (b) IMDB actors

Fig. 13. Reducing query time using improved list-merging algorithm.



(a) Choosing lists to combine (b) Scalability

Fig. 14. Reducing index size by combining lists (IMDB).

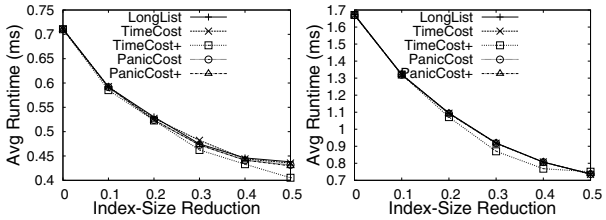
for both algorithms. We observe that on all three data sets, the query running time for both algorithms increased very slowly as we increased the index size reduction ratio, until about 40% to 50%. That means, this technique can reduce the index size without increasing the query time! As we further increased the index size reduction, the query time started to increase. For the cost-based 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 basic algorithm blindly chose lists to combine.

Figure 12(b) shows 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. Figure 14(b) shows how the algorithms of combining lists affected query performance as we increased the data size, for a reduction ratio of 40%.

D. Extension to Other Similarity Functions

For simplicity, our discussion so far mainly focused on the edit distance metric. We can generalize the results to commonly used similarity measures such as Jaccard and cosine. 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 15 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. 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.



(a) Jaccard function (b) Cosine function

Fig. 15. Jaccard and Cosine functions (DiscardLists, DBLP titles)

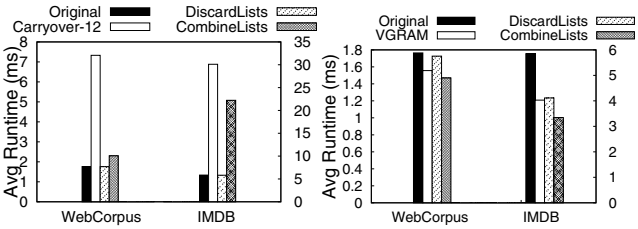
The performance started degrading at a 50% reduction ratio and increased rapidly at a ratio higher than 60%. For a 70% ratio, the time for the cosine and jaccard functions increased to 150ms and 115ms for LongList. For high reduction ratios the TimeCost and TimeCost$^+$ methods became worse than the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

E. Comparing Different Compression Techniques

We implemented the compression techniques discussed so far as well as the VGRAM technique. 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 and computed their compression ratio. Then we compressed the index using DiscardLists and CombineLists separately to achieve the same compression ratio.

Comparison with Carryover-12: Figure 16(a) compares the performance of the two new techniques with Carryover-12. For Carryover-12, to achieve a good balance between the query performance and the index size, we used fixed-size segments of 128 4-byte integers and a synchronization point for each segment. The cache contained 20,000 segment slots (approximately 10MB). It achieved a compression ratio of 58% for the IMDB dataset and 48% for the WebCorpus dataset. We see that its online decompression has a profound impact on the 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 method performed significantly better at 22.15ms for the IMDB dataset and 2.3ms for the WebCorpus dataset. The DiscardLists method could even slightly decrease the running time compared to the original index to 5.81ms and 1.75ms for the IMDB and WebCorpus datasets, respectively.

Comparison with VGRAM: Figure 16(b) compares the performance of two new techniques with VGRAM. We set its q_{min} parameter to 4. We did not take into account the memory requirement for the dictionary trie structure because it was negligible. The compression ratio was 30% for the IMDB dataset and 27% for the WebCorpus dataset. Interestingly,



(a) Carryover-12 (b) VGRAM

Fig. 16. Comparing DiscardLists and CombineLists with existing techniques at the same reduction ratio. In each figure, the left scale corresponds to the WebCorpus data set, and the right scale corresponds to the IMDB data set.

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 reduced the time from an original 5.85ms to 4.02ms, and for the WebCorpus dataset from 1.76ms 1.55ms. Surprisingly, the CombineLists method reduced the running time even more than VGRAM to 3.34ms for the IMDB dataset and to 1.47ms for the WebCorpus dataset. The DiscardLists method performed competitively for the IMDB dataset at 3.93ms and slightly 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 reductions 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.

F. Integrating Several Approaches

The methods studied in this paper are indeed orthogonal, thus we could even use their combinations 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 17. 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% ratio (4th point on the curve) we could achieve a higher reduction ratio (61%) at a lower query time (6.34ms).

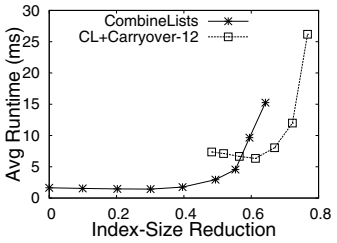


Fig. 17. Reducing index size using CombineLists with Carryover-12.

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, 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.

Additional Experiments: In [4] we included many additional experimental results, including experiments on more data sets, performance of different methods to choose candidate pairs to combine, and how the techniques perform in the presence of query-workload changes. We also discuss how to utilize filtering techniques for compression.

VII. 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.

Acknowledgements: The work was partially supported by the National Science Foundation of China under Grant No. 60828004.

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] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search (full version). Technical report, Department of Computer Science, UC Irvine, June 2008.

[5] 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.

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

[7] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, Mar 1975.

[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] S. Golomb. Run-length encodings (corresp.). *Information Theory, IEEE Transactions on*, 12(3):399–401, Jul 1966.

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

[11] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

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

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

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

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

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

[17] 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.

[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*, pages 257–266, 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):31–88, 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, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[29] 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.

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

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