B^{ed}-Tree: An All-Purpose Index Structure for String Similarity Search Based on Edit Distance

Zhenjie Zhang[1] Marios Hadjieleftheriou[2] Beng Chin Ooi[1] Divesh Srivastava[2]
[1]School of Computing, National University of Singapore, Singapore
{zhenjie,ooibc}@comp.nus.edu.sg
[2]AT&T Labs - Research, Florham Park, NJ, USA
{marioh,divesh}@research.att.com

ABSTRACT

Strings are ubiquitous in computer systems and hence string processing has attracted extensive research effort from computer scientists in diverse areas. One of the most important problems in string processing is to efficiently evaluate the similarity between two strings based on a specified similarity measure. String similarity search is a fundamental problem in information retrieval, database cleaning, biological sequence analysis, and more. While a large number of dissimilarity measures on strings have been proposed, edit distance is the most popular choice in a wide spectrum of applications. Existing indexing techniques for similarity search queries based on edit distance, e.g., approximate selection and join queries, rely mostly on n-gram signatures coupled with inverted list structures. These techniques are tailored for specific query types only, and their performance remains unsatisfactory especially in scenarios with strict memory constraints or frequent data updates. In this paper we propose the B^{ed}-tree, a B$^+$-tree based index structure for evaluating all types of similarity queries on edit distance and normalized edit distance. We identify the necessary properties of a mapping from the string space to the integer space for supporting searching and pruning for these queries. Three transformations are proposed that capture different aspects of information inherent in strings, enabling efficient pruning during the search process on the tree. Compared to state-of-the-art methods on string similarity search, the B^{ed}-tree is a complete solution that meets the requirements of all applications, providing high scalability and fast response time.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database applications

General Terms

Algorithms, Performance

Keywords

String, Edit Distance, Similarity Search, Range Query, Top-k Query, Approximate Join

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$5.00.

1. INTRODUCTION

Strings are ubiquitous in computer systems and are used in a wide spectrum of applications, e.g., for representing addresses, text documents, and biological sequences. As a fundamental data type, strings and their properties have been extensively studied since the inception of computer science, but only recently has there been interest in designing efficient index structures for evaluating various queries on strings.

One of the most important research topics for strings is similarity search, i.e., the discovery of similar strings with respect to a given distance measure. Generally speaking, a string is a sequence of letters drawn from a given alphabet. Given two strings from the given string domain, a distance measure between strings returns a non-negative real value indicating the similarity between these strings. String similarity covers a diverse set of applications that can be formulated with different query predicates. For example, to ensure the consistency of customer information it is important to locate addresses in a customer database within a certain range of similarity thresholds. In information retrieval systems, such as search engines, popular queries are usually displayed on the screen which are most similar to the query entered by the user. In biological databases all pairs of similar proteins or genes within a certain similarity threshold are retrieved to identify biological clusters. In all these applications it is crucial to provide an efficient mechanism for evaluating a diverse set of query types, namely, range and top-k *selection* queries and all-pairs *join* queries, based on a predetermined string similarity measure.

There is a long stream of research on defining string similarity measures, depending on the application domain. These measures can be divided into two major categories: *operation-based* and *token-based*. Edit distance is the representative distance measure in the first category, and is defined as the minimum number of primitive operations (insertions, deletions, and substitutions) needed to transform one string into another. Several variations of edit distance have been proposed in the past, e.g., generalizing edit operations to allow block movements [10] or assigning different weights to each operation [16]. In the second category of string similarity measures, each string is represented as a set of tokens (e.g., tokens can be words or n-grams; n-grams are overlapping substrings of the original string of fixed length n). The distance between two strings is thus calculated based on the similarity between the two token sets. A variety of set similarity functions have been proposed using a gamut of token weighting schemes (e.g, equal weighting [2,11] and idf-based weighting [5,12]).

Similarity search based on edit distance is a direct, value based search without the inherent problems of language understanding or the need to use weighting schemes. Nevertheless, it is not well supported in existing database systems due to the high dimensionality of the string domain. To overcome the curse of dimensionality, existing studies resort to the use of n-gram signatures coupled with inverted list structures to handle edit distance [2, 5, 15, 18, 20, 21] (edit distance can be lower bounded using a set intersection constraint on n-grams [11]). These techniques yield impressive performance improvements on all types of selection and join queries. However, these solutions suffer from the following weaknesses. First, inverted list based indexes cannot handle data updates efficiently. This problem is difficult to resolve unless if each inverted list is stored as an individual B$^+$-tree [8]. This approach incurs high maintenance cost and space overhead, since, in practice, it necessitates the storage and maintenance of B$^+$-trees for several hundred thousand n-grams (in which case the storage overhead for the index level of the trees becomes a dominant factor) and also voids the most important benefit of using inverted lists — taking advantage of sequential I/Os. Second, there is no single index structure that can simultaneously support all types of string similarity queries. To answer these queries, specialized indexes are typically constructed and maintained independently for each query type, leading to resource contention during updates and an unnecessary increase in storage space. Third, the efficiency of inverted list based approaches diminishes as the edit distance threshold increases. Large edit thresholds are necessary for handling top-k queries (where a maximum threshold cannot be determined in advance) and for very long string data (e.g., biological sequences). Inverted list based methods tend to generate a large number of candidate strings due to the fast expansion on the set of similar n-grams as the distance threshold increases.

This work presents a new index scheme, called the B^{ed}-tree, for answering selection and join queries efficiently using a single tree structure. The B^{ed}-tree can support incremental updates efficiently, as opposed to previous techniques. It can handle arbitrary edit distance thresholds without the need to specify a minimum threshold at construction time. Moreover, it is the first indexing scheme to support normalized edit distance for all query types. The underlying structure guarantees good performance for large distance thresholds, long strings, and large datasets. Extensive experiments on real string datasets show that the B^{ed}-tree achieves almost the same querying performance as the state-of-the-art solutions for selection and join queries with small edit distance, and is superior for large distance threshold. Finally, the B^{ed}-tree can be easily implemented on top of existing B$^+$-trees, which are supported by virtually all modern database systems.

The B^{ed}-tree applies specialized transformations from the string domain to the integer space. We identify three properties that a transformation should have in order to support all query types, namely, *string order, comparability, and lower bounding*. We also introduce three different transformation functions that capture various aspects of the abundant information contained in strings, namely, *string prefixes, n-gram counts and n-gram locations.*

The contributions of this work are:

1. We propose an all-purpose edit distance based indexing scheme using the standard B$^+$-tree structure.

2. We show that a carefully designed transformation, from the string domain to the integer domain, enables answering range and top-k selection queries and all-pairs join queries using a single index structure, both for standard and normalized edit distance.

3. We present three different string transformations that capture useful information from different aspects of strings.

4. We evaluate the B^{ed}-tree on real string datasets against the state-of-the art solutions.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 discusses the necessary background and gives a formal problem definition. Section 4 presents the basic principles of the B^{ed}-tree. Section 5 presents the three proposed string transformations. Section 6 presents a comprehensive evaluation of the proposed techniques and Section 7 concludes the paper.

2. RELATED WORK

Edit distance is one of the earliest research topics on string processing. Wagner and Fischer [19] introduced the first algorithm for computing edit distance with time and space complexity of $O(|s_1||s_2|)$, where $|s_1|$ and $|s_2|$ are the lengths of the two strings. Cormen et al. [9] presented a space-efficient algorithm with space complexity $O(\max\{|s_1|, |s_2|\})$. To the best of our knowledge, the fastest edit distance algorithm, proposed by Masek and Patterson [16], requires $O(|s|^2/\log|s|)$ time using a split-and-merge strategy to partition the problem. To capture better semantic meanings on the edit operations for string transformations, some variants on edit distance have been proposed. For example, Cormode and Muthukrishnan [10] propose a new edit distance definition allowing sub-strings to be moved with smaller cost.

Besides operation-based distance measures on strings, another important distance category consists of token-based functions. These functions transform the string similarity problem into a set-similarity problem, by representing strings as sets of tokens. Inverted lists is the most popular index structure for token-based distance functions. Gravano et al. [11] presented some simple filtering techniques to bridge the gap between token-based distances and edit distance by lower bounding the edit distance using set intersection on n-grams. Sarawagi and Kirpal [18] introduced the heap-based list merging algorithm for the set-similarity search problem. The heap-based indexing scheme is greatly improved in [4,12,15] by exploiting the lengths of the inverted lists. Other work uses various n-gram based signatures (e.g., prefixes [6], mismatch filters [20], hash-based signatures [2]) to evaluate join queries, but these approaches can easily be adapted for selection queries using inverted lists on the signatures themselves. All-pair join and self-join queries are also well studied in the literature of string similarity search. Sarawagi and Kirpal [18] present an algorithm for discovering similar pairs using an inverted list structure with the same pruning methods used for selection queries. Bayardo et al. [3] present a general self-join scheme which merges

Notation	Explanation
Σ	the alphabet
s	a string from Σ^*
$\|s\|$	length of string s
$s[i]$	the ith letter of s
$I(s,x,i)$	insert operation
$R(s,i)$	delete operation
$S(s,x,i)$	substitute operation
D	string dataset
q	query string q
θ	edit distance threshold
k	number of strings to return in top-k query
ϕ	mapping function from string domain to integers
$[s_i, s_j]$	all strings corresponding to integers in range $[\phi(s_i), \phi(s_j)]$

Table 1: Table of notations.

the index construction process and similar pair searching process. This idea is further exploited in [20], which utilizes some stronger pruning strategies to eliminate candidate string pairs. Xiao et al. [21] extend the same ideas for top-k self-join queries. All of the aforementioned approaches suffer from: inefficient updates, specialized index structures need to be build for each query type, performance degrades for large edit distance thresholds, they do not support normalized edit distance. Work for enabling efficient batch updates on inverted lists appeared in [13], with the drawback of the increased storage requirements for maintaining one B$^+$-tree per inverted list.

In information retrieval, query auto-completion is also an important topic related to string similarity search. While traditional auto-completion functionality only supports prefix search, some new studies have enabled auto-completion based on edit distance by using tries [7,14].

Approximate edit distance search is another research direction in the literature of computer algorithms. Ostrovsky and Rabani [17] proved that edit distance can be probabilistically well preserved by Hamming distance after projecting the strings onto randomly selected subspaces. Andoni and Onak [1] extended this idea by reducing both the space and time complexities.

3. PRELIMINARIES

In this section we present some preliminary knowledge regarding string processing as well as the basic problem definition. In the rest of the paper we assume the existence of a finite alphabet Σ. Given Σ, a string s is a sequence of l letters drawn from Σ, i.e., $s = s[1]s[2]\dots s[l]$ s.t. $s[i] \in \Sigma$ for $1 \leq i \leq l$. We summarize the notations used in the rest of the paper in Table 1.

Given a string s, there are three primitive operations on s: insert, delete and substitute. An insert operation $I(s,x,i)$ adds one letter $x \in \Sigma$ at position i, forming a new string of length $l+1$ $s' = s[1]\dots s[i-1]xs[i]\dots s[l]$. A delete operation $R(s,i)$ removes the letter at position i, i.e., $s' = s[1]\dots s[i-1]s[i+1]\dots s[l]$. Finally, a substitute operation $S(s,x,i)$ replaces the letter at position i with the new letter x, i.e., $s' = s[1]\dots s[i-1]xs[i+1]\dots s[l]$.

String ID	String Content
s_1	Jim Gray
s_2	Jim Grey
s_3	Michael Stones
s_4	Mike Stone
s_5	Mike Stones

Table 2: An example string dataset.

	$\emptyset$	J	i	m		G	r	**e**	y
$\emptyset$	0	1							
J	1	0	1						
i		1	0	1					
m			1	0	1				
				1	0	1			
G					1	0	1		
r						1	0	1	
a							1	**1**	2
y								2	1

Table 3: Running example of Algorithm 1.

Definition 1 (Edit Distance). *Given two strings s_i and s_j, the edit distance between s_i and s_j is defined as the minimum number of primitive operations needed to transform s_i to s_j, denoted by $d(s_i, s_j)$.*

Definition 2 (Normalized Edit Distance). *Given two strings s_i and s_j, the normalized edit distance between s_i and s_j is defined as*

$$d'(s_i, s_j) = \frac{d(s_i, s_j)}{\max\{|s_i|, |s_j|\}}.$$

Table 2 shows a sample dataset containing five distinct strings. The edit distance between s_1 and s_2 is 1, since s_1 is transformed to s_2 with a substitute operation $S(s_1, 7, e)$. Similarly, the edit distance between s_4 and s_5 is also 1, since $I(s_4, 11, s)$ transforms s_4 to s_5. While computing the edit distance between the two pairs above is straightforward, in general the computation of edit distance is not trivial. The fastest edit distance algorithm known so far runs in $O(|s|^2/\log|s|)$ time for strings of length $|s|$, based on a standard dynamic programming method. However, if we are only interested in testing if the edit distance is within some threshold θ, there is a verification algorithm in $O(\theta|s|)$ time and $O(\theta)$ space, which is much more efficient than the standard one when θ is a small constant. In Algorithm 1, we give the details on the method (we present a version that uses $O(|s|)$ space, for simplicity). In Table 3, the basic idea of the algorithm is illustrated with a running example on the edit distance between s_1 and s_2 with threshold $\theta = 1$. Generally speaking, the algorithm only tests the entries on the dynamic programming table on the diagonal with offset no larger than θ. This is motivated by the simple observation that any matching of letters with position offset larger than θ leads to distance at least $\theta + 1$. All techniques proposed in the rest of the paper assume the employment of Algorithm 1 for performing the final edit distance verification between two candidate strings.

Next, we give the formal definitions of string similarity queries with respect to edit distance.

Algorithm 1 VerifyED (string s_1, string s_2, distance threshold θ)

```
if $||s_1| - |s_2|| > \theta$ return **FALSE**
Construct a table $T$ of 2 rows and $|s_2| + 1$ columns
for $j = 1$ to $\min(|s_2| + 1, 1 + \theta)$ do $T[1][j] = j - 1$
Set $m = \theta + 1$
for $i = 2$ to $|s_1| + 1$ do
    for $j = \min(1, i - \theta)$ to $\min(|s_2| + 1, i + \theta)$ do
        $d_1 = (j < i + \theta)$ ? $T[1][j] + 1$ : $\theta + 1$
        $d_2 = (j > 1)$ ? $T[2][j-1] + 1$ : $\theta + 1$
        $d_3 = (j > 1)$ ? $T[1][j-1] + (s_1[i-1] = s_2[j-1])$ ? 0 : 1) : $\theta + 1$
        $T[2][j] = \min(d_1, d_2, d_3)$
        $m = \min(m, T[2][j])$
    if $m > \theta$ return **FALSE**
    for $j = 0$ to $|s_2| + 1$ do T[1][j] = T[2][j]
return **TRUE**
```

Definition 3 (Range Selection Query). *Given a query string q and a string set $D = \{s_1, s_2, \ldots, s_{|D|}\}$, find all strings in D with edit distance no larger than θ, i.e., $D' = \{s_i \in D \mid d(s_i, q) \le \theta\}$.*

Given the sample string set D in Table 2, if a range query $q =$"Jim Grey" and $\theta = 1$ is issued, two strings s_1 and s_2 will be returned since their distances to q are 1 and 0, respectively. Range selection queries are often used in data cleaning tasks to find out potential noise due to spelling mistakes and other data inconsistencies.

Definition 4 (Top-k Selection Query). *Given a query string q and a string set $D = \{s_1, s_2, \ldots, s_{|D|}\}$, find k strings in D with edit distance no larger than any other string in D.*

A top-k selection query $q =$"Michael Stone" with $k = 2$ returns strings s_3 and s_4, which are more similar to q than any other string in D. While top-k string similarity queries have not been extensively addressed in previous work, we argue that they provide more meaningful results than range queries in some applications, potentially involving strings with large edit distance from the query string. If, for example, we issue the top-k query "M. Stone", expecting to find all users with first names starting with 'M', there is no result in D within distance smaller than 3. In such cases, top-k queries are more natural for users than range queries with large distance thresholds. Given query $q =$"M. Stone" the top-3 similar results are s_4, s_5 and s_3.

Definition 5 (All-pairs Join Query). *Given two string sets D_1, D_2 and a distance threshold θ, find all string pairs $\{s_i, s_j\}$ s.t. $s_i \in D_1$, $s_j \in D_2$ and $d(s_i, s_j) \le \theta$.*

Given two relational tables containing string attributes, the join operator discovers joinable records with some tolerance on string matching. With threshold $\theta = 1$, a self-join query on the sample dataset in Table 2 will return two pairs $\{s_1, s_2\}$ and $\{s_4, s_5\}$, both of which are within edit distance $\theta = 1$.

Algorithm 2 FindNode (string q, B$^+$-tree node N)

```
if $N$ is the leaf node then
    Return $N$
for each splitting string $s_j \in N$ do
    if $\phi(q) \le \phi(s_j)$ then
        Return **FindNode**$(q, N_j)$
Return **FindNode**$(q, N_{m+1})$
```

4. GENERAL INDEX STRUCTURE

This section introduces the general all-purpose indexing scheme for string processing on edit distance, and discusses the applicability of the indexing scheme with respect to the properties of the transformation function used.

To index the strings with the B$^+$-tree, it is necessary to construct a mapping from the string domain to integer space. Formally:

Definition 6 (String Order). *Given the string domain Σ^*, a string order is a mapping function $\phi : \Sigma^* \to \mathbb{N}$, mapping each string to an integer value.*

The definition above implies that the mapping function ϕ uniquely decides the string order. Therefore, we abuse notation to represent with ϕ both the actual string order imposed as well as the mapping function itself. Note that some strings might be mapped to the same integer by the function ϕ. In many cases, the use of a concrete mapping function ϕ is ineffective on both computation and storage. To alleviate this problem, it is better if we do not construct the mapping explicitly. This requirement can be fulfilled if the string order ϕ satisfies the following desirable property on comparability:

Property 1 (Comparability). *A string order ϕ is efficiently comparable if it takes linear time to verify if $\phi(s_i)$ is larger than $\phi(s_j)$ for any string pair s_i and s_j.*

Here, the verification method is supposed to take linear time with respect to the lengths of the strings s_i and s_j. It is easy to see that the insertion and deletion operations on the B$^+$-tree rely only on the comparability of the string order. Therefore, any string order having Property 1 can be used to index strings on the B$^+$-tree. In Algorithm 2 we present the algorithm for locating the first leaf node of the tree potentially containing a given target string. Each intermediate node N in the B$^+$-tree contains m splitting strings $\{s_1, \ldots, s_m\}$ and $m + 1$ pointers to children nodes $\{N_1, \ldots, N_{m+1}\}$. The algorithm identifies the first splitting string with mapping value larger than that of the query string and iteratively searches the subtree from the corresponding pointer all they way to the leaf level of the tree. The implementation of insertion, deletion and split operations follow similar strategies by using the new comparison oracle between $\phi(s_i)$ and $\phi(s_j)$.

From Algorithm 2, we can see that all strings stored in the sub-tree rooted at N_j have mapping values in $[\phi(s_{j-1}), \phi(s_j)]$ by construction. To simplify notation, we say that $s \in [s_i, s_j]$ if $\phi(s_i) \le \phi(s) \le \phi(s_j)$.

The following property enables the B$^+$-tree structure to handle range queries based on edit distance:

Property 2 (Lower Bounding). *A string order ϕ is lower bounding if it efficiently returns the minimal edit distance between string q and any $s_l \in [s_i, s_j]$.*

Algorithm 3 RangeQuery (string q, B$^+$-tree node N, threshold θ, minimal string $s_{\min}$, maximal string $s_{\max}$)

```
if $N$ is the leaf node then
    for each $s_j \in N$ do
        if VerifyED$(q, s_j, \theta)$ then
            Include $s_j$ in query result
else
    if $LB(q, [s_{\min}, s_1]) \leq \theta$ then
        RangeQuery$(q, N_1, \theta, s_{\min}, s_1)$
    for $j = 2$ to $m$ do
        if $LB(q, [s_{j-1}, s_j]) \leq \theta$ then
            RangeQuery$(q, N_j, \theta, s_{j-1}, s_j)$
    if $LB(q, [s_m, s_{\max}]) \leq \theta$ then
        RangeQuery$(q, N_{m+1}, s_m, s_{\max})$
```

Algorithm 4 TopKQuery (string q, B$^+$-tree node N, threshold θ, minimal string $s_{\min}$, maximal string $s_{\max}$, result heap H)

```
if $N$ is the leaf node then
    for each $s_j \in N$ do
        if VerifyED$(q, s_j, \theta)$ then
            Insert $s_j$ into $H$
            if $|H| > k$ then pop top entry
            Update the global threshold $\theta$
else
    if $LB(q, [s_{\min}, s_1]) \leq \theta$ then
        TopKQuery$(q, N_1, \theta, s_{\min}, s_1, H)$
    for $j = 2$ to $m$ do
        if $LB(q, [s_{j-1}, s_j]) \leq \theta$ then
            TopKQuery$(q, N_j, \theta, s_{j-1}, s_j, H)$
    if $LB(q, [s_m, s_{\max}]) \leq \theta$ then
        TopKQuery$(q, N_{m+1}, s_m, s_{\max}, H)$
```

With Property 2 the B$^+$-tree can handle range queries using Algorithm 3. In the algorithm we use $LB(s_i, [s_{j-1}, s_j])$ to denote the lower bound on the edit distance between s_i and any string $s_l \in [s_{j-1}, s_j]$. Algorithm 3 iteratively visits the nodes with lower bound edit distance no larger than θ and verifies the strings found at the leaf level of the tree using Algorithm 1. Notice that the algorithm might have to traverse multiple paths down the tree (as opposed to the standard B$^+$-tree traversal algorithm). The minimal and maximal strings $s_{\min}$ and $s_{\max}$ indicate the boundaries of any string in a given subtree with respect to the string order ϕ. This information can be retrieved from the parent node, as the algorithm implies. It is easy to verify that this algorithm accurately returns all strings within distance θ from query string q if the lower bounding property holds. The efficiency of the algorithm depends on the tightness of the lower bound. We discuss concrete string orders that satisfy Property 2 in Section 5.

If a string order ϕ supports range queries, it also directly supports top-k selection queries on the B$^+$-tree structure. We simply use a min-heap to keep the current top-k similar strings and update the threshold θ with the distance value of the top element in the heap. The detailed algorithm is shown in Algorithm 4.

The standard all-pairs join algorithm on B$^+$-trees for traditional one-dimensional data discovers all node pairs $\{N_1, N_2\}$ on the same level of the tree, with non-empty value overlap on their ranges. Expansions are conducted by adding join candidates after testing every pair of children drawn from N_1 and N_2 respectively. For string join queries with threshold θ the standard algorithm is applicable if the following property holds:

Algorithm 5 JoinQuery (B$^+$-tree node N_1, B$^+$-tree Node N_2, threshold θ)

```
if $N_1$ and $N_2$ are leaf nodes then
    for each $s_i \in N_1$ and $s_j \in N_2$ do
        if VerifyED$(s_i, s_j, \theta)$ then
            Insert $(s_i, s_j)$ into the result
else
    for child node $N_i$ of $N_1$ and child node $N_j$ of $N_2$ do
        Find out the string interval $[s_i^l, s_i^u]$ and$[s_j^l, s_j^u]$ for
        $N_i$ and $N_j$
        if $LB([s_i^l, s_i^u], [s_j^l, s_j^u]) \leq \theta$ then
            JoinQuery$(N_i, N_j, \theta)$
```

	Range	Top-k	All-pairs Join
Edit Distance	P1,P2	P1,P2	P1,P3
Normalized E.D.	P1,P2,P4	P1,P2,P4	P1,P3,P4

Table 4: Necessary properties with respect to query type and distance function.

Property 3 (Pairwise Lower Bounding). *Given two string intervals $[s_1^l, s_1^u]$ and $[s_2^l, s_2^u]$, the string order ϕ is pairwise lower bounding if it returns the lower bound on the distance between any $s_i \in [s_1^l, s_1^u]$ and any $s_j \in [s_2^l, s_2^u]$.*

We use $LB([s_1^l, s_1^u], [s_2^l, s_2^u])$ to denote the lower bound edit distance between the two string intervals. This property allows the direct adoption of the standard join algorithm, as is shown in Algorithm 5. The algorithm recursively expands the nodes in depth-first order, to give a clear idea on the joining process. In our implementation, we use a heap to store the node pairs and pop out the next candidate to join if it satisfies the minimal distance lower bound.

Finally, our general index scheme is also capable of handling normalized edit distance. This is achievable if the maximal length of the string can be estimated by the string order:

Property 4 (Length Bounding). *Given any string interval $[s_i, s_j]$, the string order ϕ is length bounding if it efficiently returns an upper bound on the length of any string $s_l \in [s_i, s_j]$.*

The length bounding property can be combined with any of the query processing algorithms, by dividing the lower bound edit distance with the maximal length of the string intervals. Therefore, our index structure seamlessly supports both edit distance and normalized edit distance, if the underlying string order is consistent with these properties. We summarize the necessary properties the mapping function ϕ needs to have in order to be able to support the three types of string similarity queries based on edit distance and normalized edit distance in Table 4.

5. STRING ORDERS

In the last section we presented the general theory on string indexing using B$^+$-trees, and identified the necessary properties that a mapping from strings to integers should have in order to preserve correctness for different types of queries. In this section we discuss three concrete string orders that exploit different aspects of useful information inherent in strings, and that satisfy all of these properties.

5.1 Dictionary Order

Dictionary Order is the most straightforward choice for the string order. Next, we show that dictionary order obeys comparability, lower bounding, pairwise lower bounding, and length bounding. Therefore, it can be used to index on edit distance and normalized edit distance for range, top-k and all-pair join queries.

Given an alphabet Σ, there is a pre-defined order on all letters in Σ, i.e., $\{x_1, x_2, \dots, x_{|\Sigma|}\}$. We simply assume that the index of x_i in Σ can be calculated by the permutation function $\pi(x_i)$. Then, we map the string domain to an integer space where strings are first sorted by length and within each length group, they are sorted in dictionary order. Intuitively, such a sorting counts the total number of strings with length smaller than $|s|$ plus the number of string with length equal to $|s|$ preceding s in dictionary order. We denote this length sorted dictionary order with ϕ_d.

It is obvious that string order follows the property of comparability. Given two strings s_i and s_j, it is sufficient to find the most significant position p where the two string differ. If $\pi(s_i[p]) < \pi(s_j[p])$, we can assert that s_i precedes s_j in dictionary order ϕ_d, and vice versa. This comparison can be done in linear time with respect to the length of the strings — we do not need to actually instantiate order ϕ_d.

Dictionary order is also consistent with the property of lower bounding. Given a string interval $[\phi_d(s_i), \phi_d(s_j)]$ (or for simplicity $[s_i, s_j]$) in dictionary order, we know that all strings in this interval must share the longest common prefix of s_i and s_j, i.e., $LCP(s_i, s_j)$. To be more precise, if $s \in [s_i, s_j]$, we have:

$$\forall p \in [1, |LCP(s_i, s_j)|], \quad s[p] = s_i[p] = s_j[p]. \qquad (1)$$

Let $p = |LCP(s_i, s_j)|$. In fact, we can actually use letter $s[p+1]$ to refine the lower bound even further:

$$s_i[p+1] \le s[p+1] \le s_j[p+1], \quad p = |LCP(s_i, s_j)|. \qquad (2)$$

Recall the example dataset in Table 2. Consider the string interval $[s_1, s_2]$. Any string within interval $[s_1, s_2]$ must have the prefix "Jim Gr" and the 7th character must be between 'a' and 'e'. The suffix after the 7th character can be any valid string on the alphabet of length 1. Notice here that this does not imply that all of these strings with unknown arbitrary suffixes are actually contained in interval $[s_1, s_2]$. We simply return a super-set of the strings in the interval which is sufficient for computing a correct (albeit) loose lower bound. Notice that Equation (2) is valid only if $|s_i| > |LCP(s_i, s_j)|$. If $|s_i| \le |LCP(s_i, s_j)|$ (i.e., s_i is completely covered by s_j), then we can transform interval $[s_i, s_j]$ to the equivalent interval $[s_i', s_j]$, where $s_i' = s_i + min\Sigma$ (i.e., we extend s_i by one character, the minimal character in Σ), to satisfy Equation (2).

	∅	J	a	m		G	r	a	y
∅	0	1	2	3	4	5	6	7	8
J	1	0	1						
i	2	1	1	2					
m	3		2	1	2				
	4			2	1	2			
G	5				2	1	2		
r	6					2	1	2	
{a,b,c,d,e}	7						2	1	2

Table 5: Edit distance lower bound estimation between a string and a string interval for dictionary order.

	∅	M	i	{c,. . . ,k}
∅	0	1	2	3
J	1	2	2	
i	2	2	1	2
m	3		2	2
	4			
G	5			
r	6			
{a,b,c,d,e}	7			

Table 6: Edit distance lower bound estimation between two string intervals for dictionary order.

Given Equations (1) and (2), we can now derive an efficient lower bound computation between a query string q and any string $s \in [s_i, s_j]$, based on the edit distance verification Algorithm 1. In Table 5, a running example is presented of the computation of the lower bound on the edit distance between query "Jam Gray" and interval ["Jim Gray", "Jim Grey"], with distance threshold 1. Notice that the 8th row of the table uses a candidate letter set $\{a, b, c, d, e\}$ to represent the string interval. A query letter will match the 8th row if and only if that letter is contained in the respective set of letters. Since the given interval provides information only on the eight first characters of the strings within the interval, the algorithm stops on the 8th row and estimates the lower bound on the edit distance as the smallest value on the final row (assuming a best case scenario where a string exists within the interval matching the suffix of the query exactly). In this case the algorithm returns 1 as the estimated lower bound between the given query string and the string interval. We skip the details of the algorithm which can be easily implemented by modifying Algorithm 1.

Similarly, we can compute a lower bound edit distance between two string intervals $[s_1^l, s_1^u]$ and $[s_2^l, s_2^u]$, by combining the prefixes of the boundary strings from these two intervals. Consider two string intervals $[s_1, s_2]$ and $[s_3, s_4]$, all drawn from the example in Table 2. We construct Table 6 to compute the minimal distance with threshold $\theta = 1$. Note that there is a match between 'i' and $\{c, \dots, k\}$ in row 3, since the letter 'i' is included in the letter set. However, the algorithm stops on the 4th row, since the minimal distance on the row is already larger than the threshold $\theta = 1$.

Notice that computing $\phi_d(s)$ requires infinite precision arithmetic which is impractical. As already mentioned it is not necessary to instantiate the mapping, since it can be efficiently verified in linear time. In our implementation we store the actual keys inside each B$^+$-tree node instead of the mapping, which of course increases the storage requirements of this structure (as usually is the case for string B-trees).

Clearly, dictionary order also satisfies the length bounding property, since the length of any string $s_l \in [s_i, s_j]$ has to

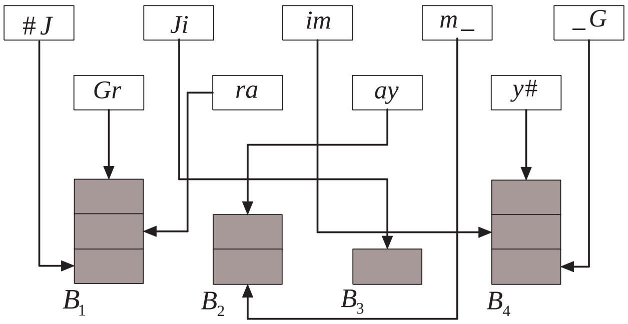


Figure 1: Example of a 2-gram counting hash.

satisfy $|s_i| \le |s_l| \le |s_j|$. This allows us to correctly answer queries using normalized edit distance.

5.2 Gram Counting Order

The dictionary order collects useful information only on the prefixes of the strings. In many cases, unfortunately, the discriminative information of the strings scatters in different positions. This motivates the use of n-grams instead of prefixes to summarize the string set. In this section, we design a string order based on counting the number of n-grams within a string. We use a hash function to map n-grams to a set of buckets of fixed cardinality. We show that the *Gram Counting Order* has the properties of comparability, lower bounding, pairwise lower bounding and length bounding. Therefore, it can be used to index on edit distance and normalized edit distance for range and top-k selection queries, and join queries.

An n-gram is a contiguous sequence of n characters from string s. Given string s there exist $|s| + n - 1$ overlapping n-grams. Considering string s_1 =“Jim Gray”, for $n = 2$, the n-gram set $Q(s_1)$ contains nine 2-grams: “#J”, “Ji”, “im”, “i_”, “_G”, “Gr”, “ra”, “ay” and “y#”. A n-gram set can be intuitively represented as a vector in a high dimensional space where each dimension corresponds to a distinct n-gram. This solution, however, incurs high storage cost. To compress the information on the vector space, we use a hash function to map each n-gram to a set of L buckets, and count the number of n-grams in each bucket. Thus, the n-gram set is transformed into a vector of L non-negative integers. In Figure 1, we hash the nine 2-grams of string s_1 to four buckets. After the mapping, the string is represented by a 4-dimensional vector $v_1 = \langle 3, 2, 1, 3\rangle$ in the gram counting space.

Previous studies have proved the strong connection between edit distance and similarity of n-gram vectors. In particular, the edit distance between two strings s_i and s_j is no smaller than

$$\max\left(\frac{|Q(s_i) \setminus Q(s_j)|}{n}, \frac{|Q(s_j) \setminus Q(s_i)|}{n}\right). \tag{3}$$

Here, $Q(s_i) \setminus Q(s_j)$ is the set of n-grams in $Q(s_i)$ and not $Q(s_j)$, and vice versa. After mapping strings from gram space to the bucket space, a new lower bound holds. If v_i and v_j are the L-dimensional bucket vector representations of s_i and s_j respectively, the edit distance between s_i and s_j is no smaller than

$$\max\left(\sum_{v_i[l]>v_j[l]} \frac{v_i[l] - v_j[l]}{n}, \sum_{v_j[l]>v_i[l]} \frac{v_j[l] - v_i[l]}{n}\right), \tag{4}$$

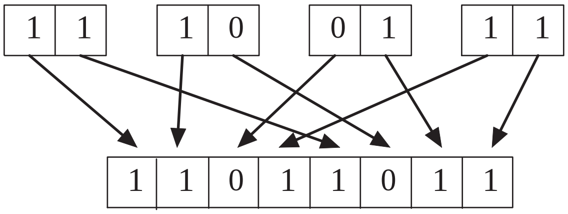


Figure 2: Example of z-order on n-gram vectors.

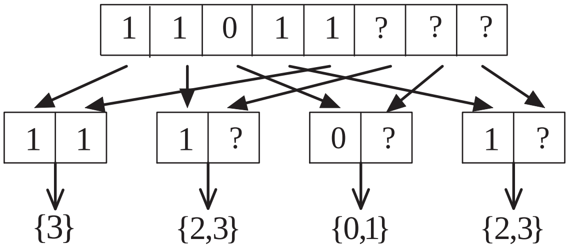


Figure 3: Example of bounding the values in buckets on z-order.

for $1 \le l \le L$. Stated simply, the edit distance should be at least as large as the largest difference in the n-gram counts between any pair of corresponding buckets in the bucket representation (correcting for the fact that one edit operation can change up to n n-grams). To achieve a tighter lower bound we apply z-order on the n-gram counting vectors to cluster strings with similar vectors as best as possible in the one-dimensional B$^+$-tree space.

Given a vector v_i, z-order interleaves the bits from all vector components in a round robin fashion. Using the example in Figure 1 with n-gram counting vector $\langle 3, 2, 1, 3\rangle$, the binary values of the vector components are “11”, “10”, “01”, “11”. Thus, the z-order value of the vector is “11011011” (see Figure 2). Therefore, each string is indexed in the B$^+$-tree according to the z-order value of its n-gram counting vector. Formally, the gram counting order ϕ_{gc} for n-gram counting vector v is defined as:

$$\phi_{gc}(s_i) = zorder(v_i) \tag{5}$$

The property of comparability is straightforwardly satisfied since it only requires to compare the binary representations of the z-order values on string s_i and s_j to verify their order in the B$^+$-tree.

Next, we analyze the property of lower bounding. For a string interval $[s_i, s_j]$ in z-order representation, a lower bound and upper bound on the number of n-grams in each bucket for all strings in the interval can be derived. This is easy to see with an example. Let s_i and s_j have binary z-order values “11011011” and “11011110” respectively, any string between them must have the prefix “11011”, while the remaining bits can be either 0 or 1. In Figure 3 it can be seen that some accurate estimation on the bucket values can be recovered if the common prefix is long enough. Specifically, the first bucket B_1 is clearly 3, since both bits are deterministicaly decided. For the rest of the buckets, the set of possible values can also be calculated with the confirmed prefix bits.

Assume that for a given string interval $[s_i, s_j]$ the lower and upper bound values on bucket B_l are $lb[l]$ and $ub[l]$ (for $1 \leq l \leq L$). After transforming the query q to vector v_q in gram counting order, we apply Equation (4) with $lb[l]$ and $ub[l]$, to reach some new lower bound on the edit distance from q to any string contained in the interval. The pairwise lower bounding property can be achieved similarly by retrieving the bound pairs $(lb_1[l], ub_1[l])$ and $(lb_2[l], ub_2[l])$ for $[s_1^l, s_1^u]$ and $[s_2^l, s_2^u]$ respectively.

The gram counting order has the length bounding property as well. Given vector v for string s in gram counting order, the length of s is $\sum_{l=1}^{L} v[l] - n + 1$, by the definition of n-grams. This implies that the length of the strings in string interval $[s_i, s_j]$ is bounded in the interval $[\sum_{l=1}^{L} lb[l] - n + 1, \sum_{l=1}^{L} ub[l] - n + 1]$. This allows us to correctly answer queries based on normalized, as well as standard edit distance.

5.3 Gram Location Order

In gram counting order ϕ_{gc}, the positional information of the n-grams is simply discarded. Motivated by the success of positional n-grams on join queries [20], we introduce another string order, called *Gram Location Order*, exploiting the positions of the n-grams to improve edit distance based pruning. The gram location order satisfies comparability, lower bounding, and pairwise lower bounding, and hence supports all types of queries but not normalized edit distance.

To conduct the transformation, the n-grams are extracted along with their actual positions in the string. For string “Jim Gray”, the extracted positional 2-grams are: (“#J”,0), (“Ji”,1), (“im”,2), (“i_”,3), (“_G”,4), (“Gr”,5), (“ra”,6), (“ay”,7) and (“y#”,8). By hashing the n-grams to integers, each positional n-gram is represented by a vector of two entries, the hash value of the n-gram and the position of the n-gram.[1] Using the same hash function shown in Figure 1, the positional n-grams of string s_1 are:

$$\{(1,0), (3,1), (4,2), (2,3), (4,4), (1,5), (1,6), (2,7), (4,8)\}.$$

Similar to the problem faced in gram counting order, the actual size of the positional n-gram set can be very large. To avoid storing long sets in the intermediate B$^+$-tree nodes, we sort the set elements based on the increasing 2-dimensional z-order value of each element (i.e., the z-order value of pair (h_s, p_s), where $h_s = h(s)$). Then, for each set we preserve only the first L elements in the z-order, and finally sort the elements in increasing order of positions. In the rest we use Z_i to denote the top-L positional n-grams for string s_i, i.e., $Z_i = \{(h_{i1}, p_{i1}), \ldots, (h_{iL}, p_{iL})\}$, in which h_{il} is the n-gram hash value, p_{il} is the corresponding position of the n-gram in the original string s_i, and $p_{i1} \leq \ldots \leq p_{iL}$.

Simply stated, we preserve only a pseudo-random subset of positional n-grams per string, and approximately compare strings based on these subsets only (the pseudo-random ordering is imposed by the hash function h and the z-order used for each vector component, which is used in order to prune both based on the n-gram hashes and the positions simultaneously). For $L = 4$ the remaining positional n-grams for string s_1, after the z-value has been computed for each set element are:

$$\{(4,2), (4,4), (2,7), (4,8)\}.$$

The mapping function of gram location order is formally defined as the dictionary order on the positional n-gram set:

$$\phi_{gl}(s_i) = \phi_d(Z_i). \tag{6}$$

After transforming every string to the top-L positional n-gram sets, the property of comparability is guaranteed due to the property of dictionary order. In the following, we analyze the property of lower bounding and pairwise lower bounding.

Similar to dictionary order, given a string interval $[s_i, s_j]$ in gram location order, all strings in this interval share a common prefix $LCP(Z_i, Z_j)$ of positional n-grams. If a query string q is also represented by a positional n-gram set Z_q, the lower bound on the edit distance from q to any string $s \in [s_i, s_j]$ can be estimated by counting the number of mismatched positional n-grams between Z_q and $LCP(Z_i, Z_j)$. A mismatch with respect to distance threshold θ was originally proposed in [20] and is formally defined below:

Definition 7 (Positional Mismatch [20]). *Given a distance threshold θ and two positional n-gram sets Z_q and Z_s, a positional n-gram $(h_{ql}, p_{ql}) \in Z_q$ incurs a mismatch if: 1. No $(h_{st}, p_{st}) \in Z_s$ exists s.t. $h_{st} = h_{ql}$ and $|p_{st} - p_{ql}| \leq \theta$ for any t; 2. No mismatching (h_{qt}, p_{qt}) has been already found s.t. $p_{qt} \geq p_{ql} - \theta$ for any $t < l$.*

The first condition checks for potential matches between the same n-grams in q and s within distance θ. The second condition checks whether there exists a mismatching n-gram preceding the current n-gram within distance θ from the position of the current n-gram. If such a mismatch exists, then it might be possible to correct both n-grams with one edit operation simultaneously (since they overlap), and thus we cannot count both n-grams as mismatches. Note that the definition above assumes that each string is long enough to contain enough positional n-grams. It is easy to modify the definition in case that the number of n-grams is not large enough, but we omit the details here due to lack of space. We can state the following lemma:

Lemma 5.1. *Given a query string q and a string interval $[s_i, s_j]$, the edit distance between q and any string $s \in [s_i, s_j]$ is lower bounded by the number of mismatches from Z_q to $LCP(Z_i, Z_j)$.*

An example of applying the mismatch conditions between query $q =$“Jim Grey” with positional n-gram set

$$\{(4,2), (4,4), (2,7), (4,8)\}$$

and an interval $[s_i, s_j]$ with $LCP(Z_i, Z_j) = \{(4,3), (2,5)\}$ is discussed next. Let the distance threshold be 1. Then $(4,2)$ is a match since there exists positional n-gram $(4,3)$ that matches on the n-gram hash value and is within distance 1. The same holds for $(4,4)$. Q-gram $(2,7)$ is a mismatch, since there is no n-gram in $LCP(Z_i, Z_j)$ close enough on position. Although there is no matching n-gram for $(4,8)$ for the same reason, this is not counted as a mismatch, since in the best case one edit operation on $(2,7)$, which is only one position away, might also fix $(4,8)$. In summary, only one mismatched positional n-gram can be identified, leading to the possibility of the existence of strings within $[s_i, s_j]$ with edit distance to q equal to the threshold.

[1] We use a hash value instead of the idf [20] to avoid the updating issues related with computing idfs when strings are added or deleted from the index.

Dataset	# of Strings	Max. Length	Avg. Length
Author	2,948,929	56	22
Title	1,158,648	675	74
Actor	1,213,391	80	23
Movie	1,568,893	247	26
Protein	508,038	1,999	347

Table 7: Dataset statistics.

To get a tighter lower bound on the edit distance estimation we can also reverse the process by counting the mismatched positional n-grams from $LCP(Z_i, Z_j)$ to Z_q. After counting the mismatches on both directions, the larger one will be returned as the lower bound value. The algorithm takes linear time with respect to the string lengths, since all positional n-grams are sorted on positions.

On the property of pairwise lower bounding, similar techniques can be applied by constructing the longest common prefix for both string intervals $[s_1^l, s_1^u]$ and $[s_2^l, s_2^u]$. Again, the number of mismatches from both positional n-gram sets are counted, the larger of which is the estimated lower bound.

6. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of the B^{ed}-tree with five real datasets with respect to range, top-k and join similarity queries on edit distance and normalized edit distance.

6.1 Setup

We use five real string datasets from different domains: *Author*, *Title*, *Actor*, *Movie* and *Protein*. Both *Author* and *Title* are extracted from DBLP,[2] containing the names of authors and paper titles respectively. The *Actor* and *Movie* datasets are taken from IMDB,[3] including strings on actor names and movie names respectively. Finally, the *Protein* dataset consists of protein sequences in flat text format from UNIPROT.[4] (We prune some extremely long strings from the original protein sequences to avoid problems of overflow on storage pages.) For each of the datasets above, we generate 100 query strings by random sampling. The detailed statistics of all five datasets are summarized in Table 7. The statistics show that the name strings in *Author* and *Actor* are usually shorter than other datasets. On the contrary, *Title* and *Protein* consist of relatively long strings. *Movie* has the largest standard deviation on string lengths, compared with other string datasets.

In the rest we use **BD**, **BGC** and **BGL** to denote our B^{ed}-tree index with dictionary order ϕ_d, gram counting order ϕ_{gc} and gram location order ϕ_{gl} respectively. Generally speaking, each group of experimental studies is partitioned into two parts, main memory and external memory based experiments. Since all state-of-the-art string indexes were designed for main memory or do not support incremental updates, we only compare against them assuming static datasets and main memory indexes. In particular, for range queries we employ **Flamingo** [15], and **Mismatch** [20] as our competitors. For top-k queries, we modify the original implementation of **Flamingo** to support top-k queries by issuing range queries with successively increasing thresholds until more than k results are returned. For join queries, **Mismatch** is used to evaluate the effectiveness and efficiency of our proposal.

Parameter	Range
Distance threshold θ	1,2,**4**,8,16
Top-k threshold k	1,2,**4**,8,16
Buffer size MB	8, 16, **32**, 64, 128
Gram size n	**2**,3,4,5
Buckets/Prefix size	2,3,**4**,5,6
Page size (KB)	1,2,**4**,8,16

Table 8: Parameters tested in our experiments.

	BD	BGC	BGL	Flamingo	Mismatch
Author	106	42	43	46	211
Title	20	24	24	67	151
Actor	38	21	21	20	93
Movie	46	27	28	21	104
Protein	211	17	23	71	233

Table 9: Index construction time (seconds) with distance threshold $\theta = 4$ and no memory constraints.

In the experiments for main memory, performance measures include index construction cost and total and average query processing. In experiments for external memory we also report I/O time for index construction and query processing, as well as the number of candidates for final edit distance verification. In Table 8 we list the testing parameters and their corresponding ranges, in which the default value is marked in bold font.

We compile all the programs in Red Hat Linux using G++ 4.5.1. The experiments are run on a Quad-Core AMD Opteron(tm) Processor 8356 with 128 GB main memory.

6.2 Range Queries

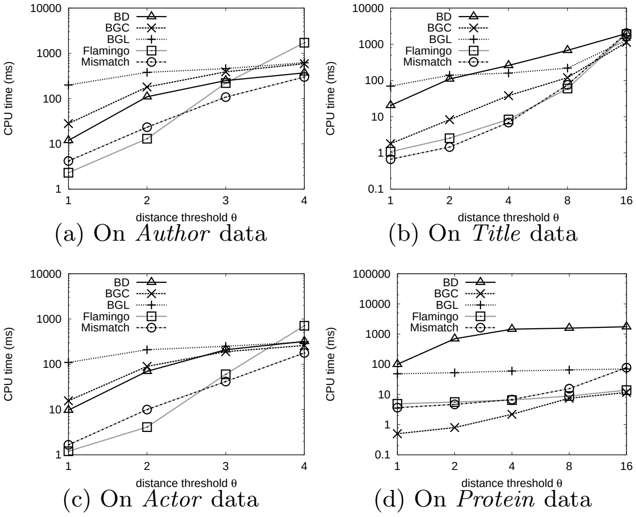
We first test the index construction time when all of the methods are allowed to use arbitrarily large main memory. In Table 9 we list the construction time in seconds spent by all five methods when the distance threshold is set at $\theta = 4$. Note that the index construction time of any B^{ed}-tree is not affected by the threshold θ. On the other hand some methods, e.g., Mismatch, have to be given a minimum querying threshold θ before-hand, and cannot correctly answer queries with thresholds smaller than the construction threshold. The results in the table imply that BGC and BGL with gram counting order and gram location order scale better than the other three solutions, especially when dealing with longer strings such as *Title* and *Protein*. The efficiency of BD is not to par since it has to maintain the complete keys in intermediate nodes of the B^{ed}-tree, while the other two orders only store compressed string representations of fixed length.

In Figure 4 we compare the average query processing time on different datasets in main memory. On *Author*, *Title* and *Actor* data, Flamingo and Mismatch outperform the B^{ed}-tree methods for small thresholds. Nevertheless, the performance gap shrinks quickly for larger thresholds. Within the three orders on the B^{ed}-tree, BGC shows advantages on query processing time over the other two solutions. It

[2] http://www.informatik.uni-trier.de/~ley/db

[3] http://www.imdb.com

[4] http://www.uniprot.org



(a) On *Author* data

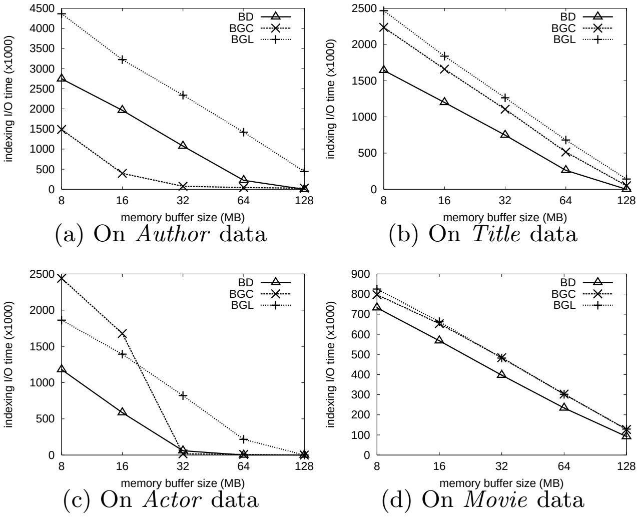
(b) On *Title* data

(c) On *Actor* data

(d) On *Protein* data

Figure 4: Average query processing time for range queries without memory constraints.

achieves almost the same performance as Mismatch for large thresholds. On the *Protein* data, the B^{ed}-tree methods are more efficient on query processing than the competitor approaches, due to the limited pruning power of n-gram inverted lists on long strings. BGC beats all other solutions by at least an order of magnitude when $\theta = 1$. BGL is less efficient than BGC since it is very hard to find common prefixes among a group of strings, which limits the pruning power of BGL only on tree nodes close to the bottom level of the tree. This result shows that mismatch-based filtering is not an effective tool for string matching when using the B^{ed}-tree.



(a) On *Author* data

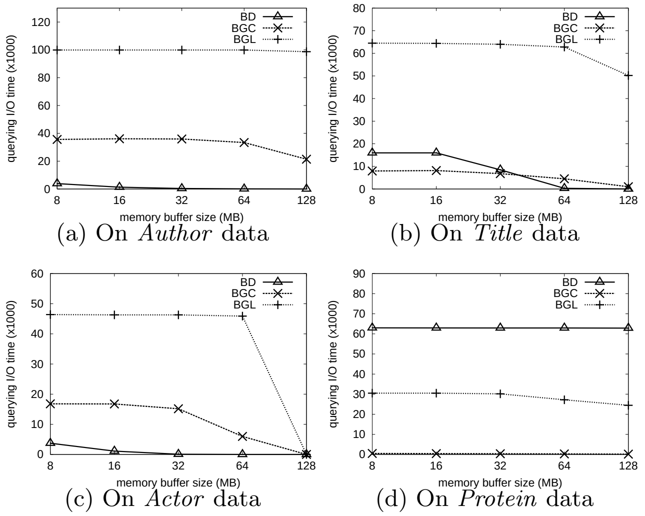
(b) On *Title* data

(c) On *Actor* data

(d) On *Movie* data

Figure 5: Total construction I/O time vs. memory buffer size.

Since all inverted list methods were designed for main memory usage and static datasets (they cannot support incremental updates), we only test the B^{ed}-tree methods in external memory and for incremental updates. (In addition, the existing implementation of Flamingo supports only main memory indexes.) Figure 5 shows the I/O time on index construction when varying the memory buffer size from 8 MB to 128 MB. These results imply that the number of I/O operations reduces sharply for increasing buffer sizes. BD spends the least time for I/O among all three methods compared in this group. When the buffer size is as large as 128 MB, all three methods can accommodate almost the complete index structure in main memory. All these results show that the B^{ed}-tree schemes are very easy to update even when there is limited memory available in the system.



(a) On *Author* data

(b) On *Title* data

(c) On *Actor* data

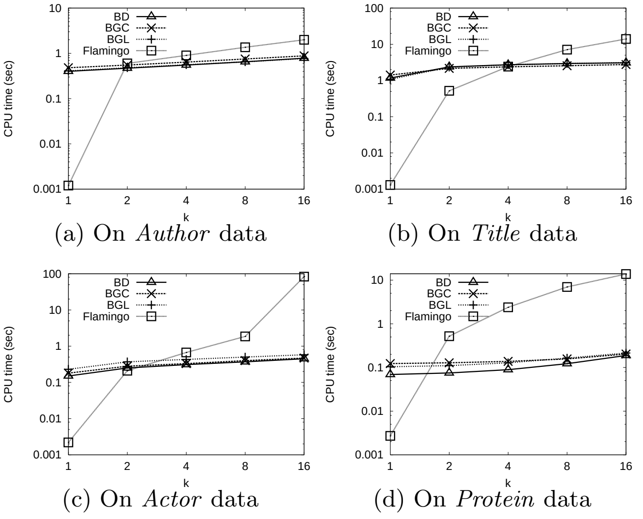
(d) On *Protein* data

Figure 6: Average range query I/O time vs. memory buffer size

In Figure 6, we evaluate the range query processing I/O for the B^{ed}-tree schemes on four string datasets. The total I/O remains almost unaffected until the buffer size is large enough to store the complete index structure. This is due to the fact that most of the I/Os are incurred on the leaf nodes of the B^{ed}-tree which are frequently swapped out when queries are continuously processed. An interesting point to note in these experiments is the low I/O cost of BD on datasets with short strings, e.g., *Author* and *Actor* data. Despite the high CPU cost of BD shown in Figure 4, BD remains a good option when only limited memory is available and the data consists of short strings. The small I/O cost is due to the fact that we store the actual keys in intermediate nodes, which results in tighter pruning. Of course, the penalty is increased CPU cost to process a large number of strings and increased storage requirements.

6.3 Top-k Queries

In many cases, top-k queries are more practical than range queries. However, existing indexing schemes with inverted lists do not naturally support such queries. To illustrate the performance benefits of our proposals, we implemented a simple strategy with Flamingo, by increasing the range query threshold gradually until more than k string results are found. Notice that we use the same B^{ed}-tree structures to support all different types of queries. Thus, we skip the performance comparisons on index construction but focus on query processing efficiency.

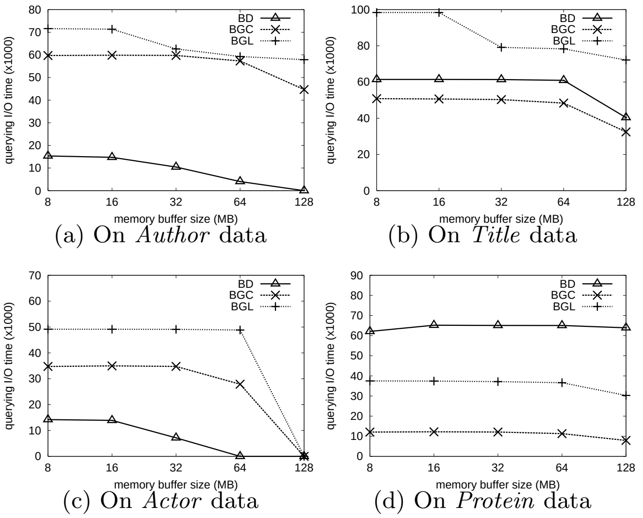


(a) On *Author* data (b) On *Title* data

(c) On *Actor* data (d) On *Protein* data

Figure 7: Average query processing time for top-k queries without memory constraints.

In Figure 7, we report the query processing time of the algorithms when the system allows all methods to load the complete index structure into main memory. From the results in the figures we can conclude that the B^{ed}-tree based index scales better then the competitive techniques for top-k queries. The CPU time for query processing is sub-linear to the number of results k. On the other hand, the variant of Flamingo does not provide an efficient solution for top-k queries, since its efficiency for range queries with large thresholds is poor. On *Actor* data, for example, it takes more than 100 seconds to return the top-16 similar strings, which is far from satisfactory if employed in real systems. On the contrary, our B^{ed}-tree method always outputs the result within 1 second, across the board.



(a) On *Author* data (b) On *Title* data

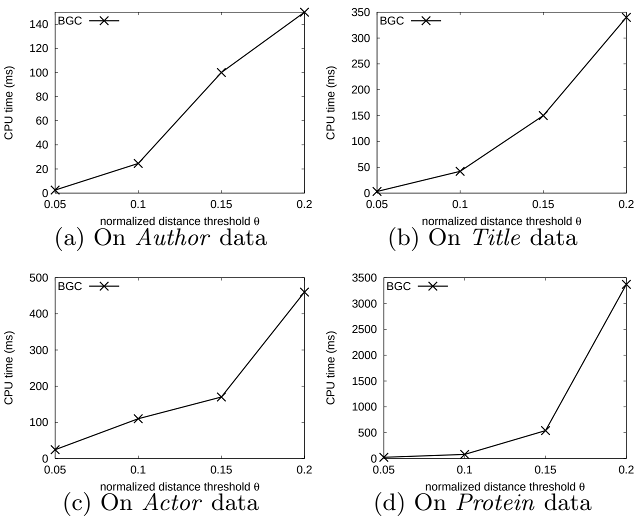
(c) On *Actor* data (d) On *Protein* data

Figure 8: Average query I/O time for top-k queries vs. memory buffer size.

Figure 8 evaluates the impact of buffer size on query processing I/O for the B^{ed}-tree structures. A sharp decrease on I/O time can be observed in the figures, especially on *Actor* data on 128 MB. Again, BD is more I/O efficient than the other two on the *Author* and *Actor* datasets because of the small lengths of strings. However, BD incurs very high I/O cost for the *Protein* data, which contains long sequences.

6.4 Normalized Edit Distance

In this group of experiments we use the B^{ed}-tree structure with gram counting order to query on normalized edit distance. Normalized edit distance is considered more difficult to evaluate over traditional edit distance, because it takes the string length into consideration. To the best of our knowledge, there is no existing systematic solution for the problem of similarity search with normalized edit distance. Therefore, we only report the performance of BGC on both range queries and top-k queries on some of the datasets.

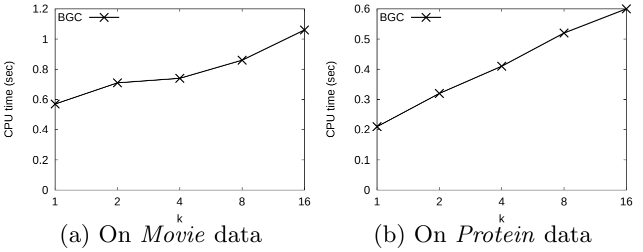


(a) On *Author* data (b) On *Title* data

(c) On *Actor* data (d) On *Protein* data

Figure 9: Average query processing time for range queries vs. distance threshold θ for normalized edit distance.

The results in Figure 9 show that BGC is effective and efficient for handling range queries for normalized edit distance. On the *Author*, *Title* and *Actor* datasets, it reports the results in 0.5 seconds on average. Even on the *Protein* data, BGC is able to return the range query results in 4 seconds when 20% of the letters are subject to change.



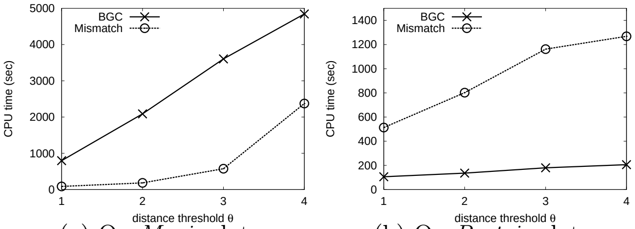
(a) On *Movie* data (b) On *Protein* data

Figure 10: Average query processing for top-k queries vs. k for normalized edit distance.

In Figure 10, we evaluate the performance of BGC on top-k queries for normalized edit distance. Similar to the results on standard edit distance, the CPU time of BGC is linear with respect to k, since the B^{ed}-tree index scheme allows the algorithm to gradually prune branches not containing string candidates. The method is very efficient, outputting the query results within 1 second in almost all cases.

6.5 Join Queries

Join query is the most expensive operator on string datasets since it returns all pairs of strings within a given distance threshold θ. On the *Author* and *Actor* datasets, the joined results contain an overwhelming amount of string pairs even for small distance thresholds. Thus, we only test on the *Movie* and *Protein* data, which output string pairs of reasonable size.



(a) On *Movie* data (b) On *Protein* data

Figure 11: Average query processing time for join queries vs. distance threshold θ.

The results for *Movie* and *Protein* data implies that the query processing time of the B^{ed}-tree based method is linear to the distance threshold θ. On string datasets with small string lengths the Mismatch based method outperforms BGC by a large margin when the threshold is small. However, on the *Protein* data with long strings, the Mismatch method is much less efficient due to the poor pruning effectiveness.

7. CONCLUSION

In this paper we propose a general and simple B^{+}-tree based indexing structure to support a broad class of string similarity queries with respect to edit distance. We prove that the capabilities of the index rely only on the properties of the transformation function used to implicitly map strings into integers. We present three transformation functions, namely, dictionary order, gram counting order and gram location order. The experiments on real datasets show that our indexing scheme achieves comparable performance against state-of-the-art solutions on range, top-k, and join queries. Additionally, our all purpose index can support normalized edit distance and, most importantly, incremental updates. The index can be built once and used with arbitrary distance thresholds and for all query types. All of the above are a departure from previous work. Furthermore, our index scheme can be easily implemented in existing commercial database systems using existing B^{+}-tree structures and has small memory requirements.

8. ACKNOWLEDGMENTS

Zhenjie Zhang and Beng Chin Ooi were supported by Singapore NRF grant R-252-000-376-279.

9. REFERENCES

[1] A. Andoni and K. Onak. Approximating edit distance in near-linear time. In *STOC*, pages 199–204, 2009.

[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. In *ICDE*, pages 604–615, 2009.

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

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–15, 2006.

[7] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD*, pages 707–718, 2009.

[8] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[10] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Transactions on Algorithms*, 3(1), 2007.

[11] 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.

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

[13] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD*, pages 429–440, 2009.

[14] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[15] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[16] W. J. Masek and M. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[17] R. Ostrovsky and Y. Rabani. Low distortion embeddings for edit distance. *Journal of the ACM*, 54(5):23–36, 2007.

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

[19] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[20] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[21] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.