# Efficiently Supporting Edit Distance based String Similarity Search Using B$^+$-trees

Wei Lu, Xiaoyong Du, Marios Hadjieleftheriou, Beng Chin Ooi

**Abstract**—Edit distance is widely used for measuring the similarity between two strings. As a primitive operation, edit distance based string similarity search is to find strings in a collection that are similar to a given query string using edit distance. Existing approaches for answering such string similarity queries follow the filter-and-verify framework by using various indexes. Typically, most approaches assume that indexes and datasets are maintained in main memory. To overcome this limitation, in this paper, we propose B$^+$-tree based approaches to answer edit distance based string similarity queries, and hence, our approaches can be easily integrated into existing RDBMSs. In general, we answer string similarity search using pruning techniques employed in the metric space in that edit distance is a metric. First, we split the string collection into partitions according to a set of reference strings. Then, we index strings in all partitions using a single B$^+$-tree based on the distances of these strings to their corresponding reference strings. Finally, we propose two approaches to efficiently answer range and KNN queries, respectively, based on the B$^+$-tree. We prove that the optimal partitioning of the dataset is an NP-hard problem, and therefore propose a heuristic approach for selecting the reference strings greedily and present an optimal partition assignment strategy to minimize the expected number of strings that need to be verified during the query evaluation. Through extensive experiments over a variety of real datasets, we demonstrate that our B$^+$-tree based approaches provide superior performance over state-of-the-art techniques on both range and KNN queries in most cases.

**Index Terms**—Similarity search, string, edit distance, B$^+$-tree

## 1 Introduction

As a primitive operation, string similarity search has a wide variety of applications in data cleaning, data integration, error checking, pattern recognition, biological sequence analysis, and so forth. For example, in error checking, string similarity search can help find good recommendations of similar words from a given dictionary when users make typos.

Thus far, a variety of similarity functions have been proposed to measure the similarity between two strings [1]. While it is widely recognized that there does not exist a similarity function that provides the best quality in all application domains [1], [2], edit distance remains one of the widely accepted similarity measures. Therefore, we employ edit distance to measure the similarity between two strings. Given a query string $q$, string similarity search returns a set of strings from a collection of strings $S$ that are similar to $q$. In general, two types of queries are often considered:

- **Range search:** Given a string $q$, range search returns all strings in $S$ with edit distance to $q$ not greater than a user defined threshold $\theta$.
- **$K$-nearest-neighbor search (KNN):** Given a string $q$, KNN search returns $K$ strings in $S$ with the smallest edit distances to $q$.

Verifying whether the edit distance between two strings with length $|s|$ is smaller than or equal to $\theta$ has complexity $O(|s|\theta)$ which is expensive for large $\theta$ or long strings (*computing*, as opposed to *verifying*, the edit distance between two strings has complexity $O(|s|^2/\log|s|)$; for the purpose of answering range or KNN queries we mostly need to verify edit distances). It is therefore important for the indexing mechanism to filter out as many strings as possible so that verification between string pairs can be reduced to a minimum. Existing techniques for string similarity search use either $n$-gram based inverted indexes [3], [4], [5], [6], [7], [8], [9], [10], Trie based indexes [11], [12] or $B^+$-trees [13].

The main idea behind the $n$-gram based inverted index approaches is to use the inverted lists to identify all candidate strings that have a certain number of $n$-grams in common with the query. This approach can support both range and KNN queries, and is efficient when $\theta$ is small, but *suffers for larger $\theta$ and short strings*. In addition, it incurs very high space overhead, resulting in indexes that are more than five times larger than the size of the original dataset in most cases, due to the decomposition of strings into overlapping $n$-grams. Besides, as we argue in details in Section 2 using an example, this approach in many cases resorts to the linear scan for range queries but misses results for KNN queries. Moreover, it incurs a random I/O for every candidate examined, given that the index is not clustered


- *Wei Lu and Beng Chin Ooi are with the School of Computing, National University of Singapore, Singapore.*
  *E-mail: {luwei1, ooibc}@comp.nus.edu.sg*
- *Xiaoyong Du is with the Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China, and School of Information, Renmin University of China, China.*
  *E-mail: duyong@ruc.edu.cn*
- *Marios Hadjieleftheriou is with the AT&T Labs-Research, 180 Park Ave Bldg, 103 Florham Park, NJ 07932.*
  *E-mail: marioh@research.att.com*

(clustering inverted lists would lead to even larger space explosion, since each string would have to be duplicated as many times as the number of $n$-grams it contains). Trie based indexing approaches are main memory only and hence are not directly comparable. $B^{\text{ed}}$-tree [13], a $B^+$-tree based approach, organizes strings based on a global ordering and exhibits good scalability. However, the range search of this approach degenerates to partially sequential scans of the tree when *the average length of strings in the dataset is short*.

In this paper we introduce a new approach that is based on constructing a clustered B$^+$-tree, which reduces random I/Os to a minimum, making our technique faster than existing alternatives across a wide range of edit distance thresholds and average string lengths in the indexed dataset. Our approach is based on the fact that edit distance satisfies the properties of a metric space. Consequently, the strings can be indexed using a B$^+$-tree based on the concepts of distance transformation and distance indexing of the iDistance method [14], [15]. By selecting a set of reference points, the iDistance method partitions the dataset into clusters such that data points within that cluster can be indexed based on their distances from the reference point. In essence, this technique implicitly partitions the data space into Voronoi cells, thereby achieving efficient indexing in a high-dimensional space by using a B$^+$-tree. When a range query is issued, the ranges formed by the data partitions that intersect with the query range are examined, and data points are pruned based on the metric values indexed by the B$^+$-tree.

The iDistance method provides an efficient way of indexing high-dimensional data points in a one dimensional metric space. However, direct application of iDistance for supporting string similarity search is challenging with respect to three issues:

- iDistance can be applied only in Euclidean space. Partitioning of the string collection in the string domain remains an open problem since the distance between two strings is difficult to express geometrically.
- Reference selection over string domain is not supported in iDistance. In Euclidean space, selection of the reference points depends on the spatial geometry. e.g., the centroid of the cluster. In the high-dimensional string domain, there is no known way of identifying the centroid of a set of strings under edit distance. Furthermore, identifying the median string of a given string collection in terms of edit distance is known to be an NP-hard problem [16].
- It is not easy to find the best partition assignment strategy. For clustering data, a typical objective function to optimize is the total distance of points within a cluster from a reference point. In the high dimensional string domain this assignment strategy might not always lead to the best clustering with respect to answering queries based on edit distance.

In this paper we make the following contributions:

- We propose a partitioning based approach that employs the B$^+$-trees to answer both range and KNN queries. Our approach can be easily integrated into existing RDBMSs.
- By employing our approach, strings that need to be verified are maintained in the continuous leaves of the B$^+$-tree, hence reducing random I/O to a minimum. Furthermore, the size of the index only relies on the cardinality of the dataset, while existing indexing techniques depend on both the cardinality and the length of strings in the dataset.
- Given a set of reference strings, we propose a novel partition assignment for each string that directly minimizes the expected number of strings that need to be verified per query. We demonstrate that selecting the optimal set of reference strings is an NP-hard problem. Hence, we propose a heuristic approach to extract the reference strings greedily.
- We conduct an extensive experimental study to evaluate the performance of the proposed approach in comparison with state-of-the-art techniques.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 states the problem and necessary definitions. Section 4 describes the partitioning based approach using the metric properties of edit distance. Section 5 presents algorithms for choosing reference strings and assigning strings to partitions. Experimental results are presented in Section 6. Finally, Section 7 concludes this paper.

## 2 Related Work

The problem of efficiently answering string similarity queries under edit distance has attracted a lot of interest from different research communities. Given that computing the edit distance between two strings is an expensive operation, filtering algorithms are essential for reducing the cost of evaluating queries. A variety of filtering algorithms have been proposed specifically for join, range and KNN queries [3], [17], [18], [5], [13], [7], [8], [19]. In addition, a variety of indexing techniques have been proposed that make use of these filters to reduce the candidate set size.

The main idea behind the $n$-gram based inverted index approaches is to use the inverted lists to identify all candidate strings that have a certain number of $n$-grams in common with the query. The intuition here is that strings with small edit distance must share many $n$-grams. This is also known as the $T$-occurrence problem [20], where $T$ depends on the length of the query $|q|$, $n$ and $\theta$. Clearly, the cost of this approach for answering range queries depends on the algorithm used to merge inverted lists. Sarawagi and Kirpal [20] propose a heap based merging algorithm. Li et al. [5] further optimize the merging process by identifying elements that can be skipped in the inverted lists. Similar merging techniques can be used to answer KNN queries with few modifications [6].

This technique is efficient for long strings but suffers for short strings since in this case, when $\theta$ is relatively large but is still a meaningful value, the number of candidates explodes using either short $n$-grams (poor selectivity) or long $n$-grams ($T$ becomes small). More importantly, it also suffers from another major problem: $n$-gram based inverted index approaches make an implicit assumption that strings within edit distance $\theta$ have *at least one* $n$-gram in common. However, in practice, it is not always the case. Consider a simple example where $\theta = 2$, $q = abc$ and $s = cba$. Clearly the two strings have no $n$-grams in common yet they are within edit distance 2. String $s$ would not be identified as an answer in this example, given that only the inverted lists of $n$-grams belonging to string $q$ would be examined. In this case (i.e., when $T \leq 0$) this technique resorts to a linear scan. However, for KNN queries, this approach will miss some of the $K$ nearest neighbors if they have no $n$-grams in common with the query. That's why Flamingo [6], makes the assumption that all $K$ nearest neighbors share at least one $n$-gram with the query. Finally, this technique also incurs relatively high space overhead in comparison with the size of the original dataset, due to the decomposition of strings into overlapping $n$-grams. Li et al. [4] propose the use of variable length $n$-grams in order to reduce the size of inverted lists. Kim et al. [21] and Behm et al. [9] propose various algorithms for compressing inverted lists without affecting query performance. Recently, [7], [8] propose the prefix-based approach which is optimized for a specific threshold. In [9], [10], they focus on designing disk-based indexes to support string similarity search by extending $n$-gram based inverted indexes. Finally, and most importantly, $n$-gram based inverted indexes, incur one random I/O per candidate verification needed.

The second approach builds tries to answer selection queries based on edit distance. Chaudhuri and Kaushik [12] and Ji et al. [11], [22] present specialized algorithms on tries that preserve the state of previous computations on a particular query string, in order to speed up the computation of edit distance on subsequent extensions of the query string (by appending characters to the end of the string). Based on similar ideas, Deng et al. [23] extend the trie structure to support KNN queries. However, these trie-based approaches are main memory only.

The third approach uses B$^+$-trees to index strings for answering range and KNN queries. Zhang et al. [13] propose to organize the string collection using a $B^+$-tree index based on three unique global orders. Then, they present specialized algorithms that use the $B^+$-tree to answer range and KNN queries efficiently. This approach suffers when the indexed strings are short.

There also exist some related work about how to partition the dataset [24], [25], [26]. Jin et al. [24], [25] cluster the dataset using the $k$-medoids algorithm. Each medoid is selected as a reference string and strings in the dataset are assigned to the referencee string with the closest distances. In [26], they propose a method called Maximum Variance to select the reference strings. Specifically, they iteratively select the reference strings, and at each iteration, a string that makes the maximum variance for partial strings in the dataset is selected as the reference string. Although these approaches are proposed based on certain criteria, all of them lack an explicit definition what partitioning strategy can optimize the query performance. To address this problem, we argue that the optimal partitioning strategy is not to minimize the distance of strings from the respective reference strings, but to directly minimize the expected number of strings that need to be verified per query. Indeed, due to the different partitioning strategy, in [26], the distance between each string in the dataset and every reference string is required to maintain in main memory. Such maintenance incurs prohibitive storage cost and update cost when integrating this method into existing DBMSs. Take a dataset with 1M strings for example. Suppose 1% of strings are taken as the reference strings (in their experiments, the percentage is set to at least 2.5%). As we can see, the storage cost to maintain these distances is 4 * $10^6$ * $10^4$B = 40GB. Although they argue that the storage cost can be reduced by maintaining the distances from each string to a portion of reference strings, the storage cost can still be expensive.

| Symbol | Definition |
|---|---|
| $U$ | a universal string domain |
| $S$ | a string collection, $S \subset U$ |
| $s_i$ | the $i^{\text{th}}$ string in $S$ |
| $q$ | a query string |
| $ed(q, s)$ | the edit distance between $q$ and $s \in S$ |
| $\theta$ | the threshold for range search |
| $K$ | the threshold for KNN search |
| $N$ | the number of partitions |
| $P_i$ | a partition, where $P_i \subseteq S, 1 \leq i \leq N$ |
| $o_i$ | the reference string corresponding to $P_i$ |
| $P_i.l$ | minimum edit distance between $o_i$ and all $s \in P_i$ |
| $P_i.u$ | maximum edit distance between $o_i$ and all $s \in P_i$ |
| $P_i[j]$ | $\{s \mid s \in P_i, ed(s, o_i) = j\}$ |

TABLE 1
Symbols and their definitions.

## 3 Problem Definition

Let alphabet $\Sigma$ be a finite nonempty set of symbols. For string $s$, we use $|s|$ to denote the length of $s$, and $s[i]$ $(1 \leq i \leq |s|)$ to denote the $i^{\text{th}}$ symbol of $s$, where $s[i] \in \Sigma$. An edit operation on $s$ is either one of the following: **(1) Insertion:** insert a symbol $x \in \Sigma$ into $s$ to form a new string $\bar{s} = s[1] \dots s[i-1]xs[i] \dots s[|s|]$; **(2) Deletion:** delete a symbol $s[i]$ from $s$ to form a new string $\bar{s} = s[1] \dots s[i-1]s[i+1] \dots s[|s|]$; **(3) Substitution:** substitute a symbol $s[i]$ of $s$ with $x \in \Sigma$ to form a new string $\bar{s} = s[1] \dots s[i-1]xs[i+1] \dots s[|s|]$. The notation used throughout the paper is summarized in Table 1. Now we can define edit distance:

**Definition 1. (Edit Distance)** *Given two strings $s$ and $\bar{s}$, the edit distance between $s$ and $\bar{s}$, denoted as $ed(s, \bar{s})$, is defined*

*as the minimum number of edit operations that are required to transform $s$ to $\bar{s}$.*

String similarity queries based on edit distance are defined as follows:

**Definition 2. (Range Search)** *Given a query string $q$ and a collection of strings $S$, a range query returns a set of strings $\bar{S} \subseteq S$, such that, $\forall \bar{s} \in \bar{S}$, $ed(\bar{s}, q) \leq \theta$, while $\forall s \in S - \bar{S}$, $ed(s, q) > \theta$, where $\theta$ is a user specified threshold.*

Range search is an important query type for string similarity search when the length of strings in the dataset is uniformly short (e.g., employee names and company names in relational databases). In general, when the length of the strings in the dataset is short, user defined thresholds also tend to be small (large edit distance thresholds on short strings result in too many meaningless answers). However, when the average length of strings in the dataset is relatively large and non uniform (e.g., a database of protein sequences), using a small edit distance threshold may not always produce any results. As such, KNN search in this case is a more meaningful query:

**Definition 3. (KNN Search)** *Given a query string $q$ and a collection of strings $S$, a KNN query returns a set of strings $\bar{S} \subseteq S$, such that $|\bar{S}| = K$, and $\forall \bar{s} \in \bar{S}$, $\forall s \in S - \bar{S}$, $ed(\bar{s}, q) \leq ed(s, q)$, where $K$ is a user specified constant.*

The edit distance between $s$ and $\bar{s}$ can be computed using dynamic programming and involves the use of an $(|s| + 1) * (|\bar{s}| + 1)$ matrix. The value of the element on row $i$ $(0 \leq i \leq |s|)$, column $j$ $(0 \leq j \leq |\bar{s}|)$ of the matrix, denoted as $d[i][j]$, is:

$$d[i][j] = \begin{cases} i, & \text{if } j = 0; \\ j, & \text{if } i = 0; \\ d[i-1][j-1], & \text{if } s[i-1] = \bar{s}[j-1]; \\ 1 + min\{d[i-1][j-1], d[i-1][j], d[i][j-1]\}, & \text{else.} \end{cases}$$

Then, $d[|s|][|\bar{s}|]$ is the actual edit distance between $s$ and $\bar{s}$. The time complexity of this dynamic programming approach is $O(|s| * |\bar{s}|)$ (the fastest known algorithm has complexity $O(|s| * \max\{1, |\bar{s}|/log|s|\})$ when $|s| \geq |\bar{s}|$ [27]). For our purposes we need to verify whether the edit distance between two strings is not greater than a user specified threshold $\theta$. The verification algorithm provided in [13] has the time complexity $O(\max(|s|, |\bar{s}|)\theta)$.

## 4 Partitioning Based Solution

Our main idea is to split the string collection into partitions, where each partition is characterized by a representative *reference* string and each string in the collection is assigned to one and only one partition based on a certain criterion. Based on an important fact that edit distance is a metric (Theorem 9.4, Chapter 9 in [28]), we elaborate on how to apply the triangle inequality to prune strings with edit distance to the query that is greater than the given query threshold. We then show how to map strings in partitions to a one dimensional space and index these strings using a B$^+$-tree based on the iDistance approach. Finally, we develop two approaches to answer range and KNN queries, respectively, by traversing the B$^+$-tree.

| String | Reference String | Edit Distance |
|---|---|---|
| Robert Marcus | Robert Mercas | 2 |
| Robert Morris | Robert Mercas | 3 |
| Robert Berks | Robert Mercas | 3 |
| Robert Fergus | Robert Mercas | 3 |
| Robert Lewis | Robert Mercas | 4 |

TABLE 2
Edit distance between strings and the reference string.

### 4.1 Preliminaries

Let $O$ be a set of reference strings and $|O| = N$. Given a string collection $S$, $\forall s \in S$, $s$ is assigned to one and only one reference string $o \in O$ according to a certain criterion which will be addressed in the next section. Hence, we can split $S$ into $N$ partitions, i.e., $S = \cup_{i=1}^{N}\{P_i\}$ and $P_i \cap P_j = \emptyset$. By default, let $o_i$ be the reference string of partition $P_i$ and $P_i.l$, $P_i.u$ be the minimum and maximum edit distance, respectively, between all $s \in P_i$ and $o_i$. Formally, $P_i.l = \min\{ed(s, o_i) | s \in P_i\}$, $P_i.u = \max\{ed(s, o_i) | s \in P_i\}$. For simplicity, let partition $P_i$ be denoted with $P_i(o, l, u)$, where $l = P_i.l$ and $u = P_i.u$. Note that the edit distance between any two strings must be an integer. Hence, given an integer $j$, we let $P_i[j] = \{s | s \in P_i, ed(s, o_i) = j\}$.

**Example 1.** *Consider a set of strings shown in the first column of Table 2 as partition $P$. Suppose that string "Robert Mercas" is selected as the reference string $o$. The edit distance between each string in $P$ and $o$ is shown in the third column. $P$ can be represented as $P$("Robert Mercas", 2, 4). $P[2]$ = {"Robert Marcus"}, $P[3]$ = {"Robert Morris", "Robert Berks", "Robert Fergus"}, and $P[4]$ = {"Robert Lewis"}.* □

### 4.2 Verification of Partitions

When a user issues a range query, the relationship between any partition and the query string will belong to one of the following types:

**Definition 4. (Prunable Partition)** *Given a range search with threshold $\theta$ and query string $q$, a partition $P$ is a prunable partition if and only if one of the following conditions holds: (1) $ed(o, q) - P.u > \theta$; or (2) $P.l - ed(o, q) > \theta$.*

Figure 1 shows an example of prunable partition. From the figure, we can see that there is no overlap between the search range and the edit distance range of the partition.

**Pruning Rule 1.** *If partition $P$ is a prunable partition, then $\forall s \in P$, $ed(s, q) > \theta$. Thus, strings that lie in $P$ do not have to be verified and can be pruned directly.*

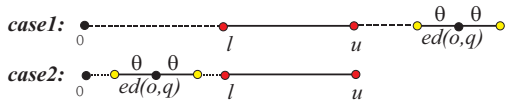



Fig. 1. Prunable Partition

*Proof:* Suppose $ed(o,q) - P.u > \theta$. $\forall s \in P$, since $ed(o,s) \leq P.u$, according to the triangle inequality, $ed(s,q) \geq ed(o,q) - ed(o,s) \geq ed(o,q) - P.u > \theta$. We omit the proof of case 2 which is analogous to that of case 1. □

**Definition 5. (Candidate Partition)** *Given a range search with threshold $\theta$ and query string $q$, partition $P$ is a candidate partition if and only if one of the following conditions holds: (1) $P.l \leq ed(o,q) \leq P.u$; or (2) $P.l - \theta \leq ed(o,q) \leq P.l$; or (3) $P.u \leq ed(o,q) \leq P.u + \theta$;*

Figure 2 shows an example of candidate partition. From the figure we can see that there is overlap between the search range and the edit distance range of the partition for each case.

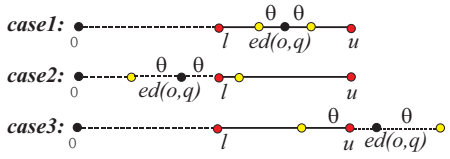



Fig. 2. Candidate Partition

**Pruning Rule 2.** *Suppose $P$ is a candidate partition. Then, $\forall s \in P$, $s$ needs to be verified if and only if*

$$lb \leq ed(o,s) \leq ub \tag{1}$$

*where $lb = \max\{ed(o,q) - \theta, P.l\}, ub = \min\{ed(o,q) + \theta, P.u\}$. We refer to range $[lb, ub]$ as* **candidate region**.

*Proof:* $\forall s \in S$, suppose that $|o,s| < lb$ or $|o,s| > ub$. For the first case, we can derive $|o,s| < ed(o,q) - \theta$ since $|o,s| \geq P.l$. Hence, $\theta < ed(o,q) - |o,s| \leq |q,s|$; For the second case, we can derive $|o,s| > ed(o,q) + \theta$ since $|o,s| \leq P.u$. Thus, $\theta < |o,s| - ed(o,q)| \leq |s,q|$. □

**Definition 6. (Selectable Partition)** *Given a range search with threshold $\theta$ and query string $q$, a partition $P$ is a selectable partition if and only if the following condition holds: $ed(o,q) + P.u \leq \theta$.*

Figure 3 shows an example of a selectable partition. Generally, selectable partitions occur when $\theta$ is relatively large and $P.u$ is relatively small.

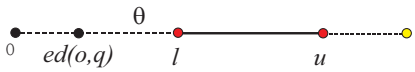



Fig. 3. Selectable Partition

**Pruning Rule 3.** *Suppose that partition $P$ is a selectable partition. Then, $\forall s \in P$, $ed(s,q) \leq \theta$, and $s$ is reported as a result. We refer to range $[P.l, P.u]$ as* **selectable region**.

*Proof:* $\forall s \in P$, based on the triangle inequality, $ed(q,s) \leq ed(o,s) + ed(o,q)$. Since $ed(o,q) + P.u \leq \theta$ and $ed(o,s) \leq P.u$, we derive $ed(q,s) \leq P.u + ed(o,q) \leq \theta$. □

To answer both range queries and KNN queries, we sequentially process partitions. In general, based on the above three pruning rules, for each partition $P$, if $P$ is either a prunable partition or a selectable partition, no further verification[1] of edit distances on the strings contained in the partition is required. If $P$ is a candidate partition, then only strings that lie in the candidate region need to be verified. Note that in order to identify the relationship between each partition and the query string, we need to actually *compute* (rather than *verify*) the edit distance between the query and all reference strings (hence, the number of reference strings selected affects the cost of the algorithms).

### 4.3 Index Construction

We now show how to employ the iDistance method [14], [15] to index the partitions using a single B$^+$-tree. As already discussed, strings in each partition $P_i$ can be ordered in a one-dimensional space by using their distances from a reference string $o_i$. In order to index all strings of the collection into a single B$^+$-tree, we must guarantee that no strings from the other partitions are accessed when we try to traverse strings of a specific partition. In other words, we must guarantee that strings of the same partition are clustered within continuous physical pages. Hence, an alternative way is that we map the edit distance of each string to a new value based on its reference string $o_i$ using the following formula:

$$\overline{ed}(s,o_i) = i \times c + ed(s,o_i) \tag{2}$$

where $c$ can be set to 1 plus the maximum length of all strings in the collection (note that the edit distance between any two strings in the collection cannot exceed the maximum length). This mapping limits the range of mapped distances for strings in partition $P_i$ to $[i \times c, (i+1) \times c)$, and consequently, there is no overlap between any two partitions as each partition is mapped to a separate range. Figure 4 shows an example of indexing strings in two adjacent partitions $P_i$ and $P_{i+1}$ using a B$^+$-tree. As the range of edit distances between strings in $P_i$ and $o_i$ is $[P_i.l, P_i.u]$, all strings will be indexed using the range of values from $i \times c + P_i.l$ to $i \times c + P_i.u$ in the leaf nodes of the B$^+$-tree. Similarly for $P_{i+1}$. Hence, given a partition $P_i$, $\forall s \in P_i$ we can construct a tuple $\langle \overline{ed}(s,o_i), \tau(s) \rangle$, where $\overline{ed}(s,o_i)$ is the key, and $\tau(s)$ is the physical address of $s$ in the data file, and insert the tuple into the $B^+$-tree. Hence, we can easily build a clustered $B^+$-tree as the index.

**Remark.** To insert a string $s$, we first identify the reference string to which $s$ is assigned. For ease of explanation, $o$ is denoted as the reference string. Then, we compute $\overline{ed}(s,o)$ according to Equation 2, construct a

1. We will explain the reason why it is unnecessary to do the verification even for answering KNN queries in the next section.

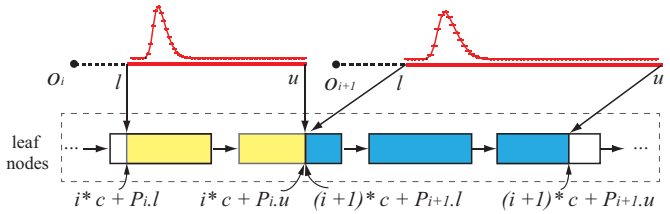



Fig. 4. Keys of Strings from Two Adjacent Partitions in the B$^+$-tree.

**Algorithm 1:** `RangeSearch` $(\mathbf{P}, btree, q, \theta)$

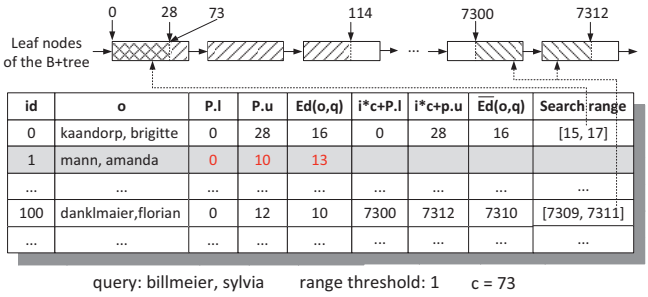
```
1  SR ← ∅; // a list maintains search range for partitions
2  foreach P_i ∈ P do
3      if P_i is a selectable partition then
4          SR.add(⟨[i*c + P_i.l, i*c + P_i.u], true⟩);
5      else if P_i is a candidate partition then
6          SR.add(⟨[i*c + lb, i*c + ub], false⟩);
7  S̄ ← ∅;        scan.keys ← SR;
8  while btGetNextIndexTuple(btree, &scan) do
9      s ← getString(scan.τs);
10     if scan.selectable then
11         S̄ ← S̄ ∪ {s};
12     else if VerifyED(s, q, θ) ≠ −1 then
13         S̄ ← S̄ ∪ {s};
14 return S̄;
```

tuple $\langle \overline{ed}(s, o_i), \tau(s) \rangle$, and insert the tuple into the B$^+$-tree. Similarly, to delete a string $s$, we also need to identify the reference string $o$ to which $s$ is assigned, compute $\overline{ed}(s, o)$, find all keys that are equal to $\overline{ed}(s, o_i)$ in the B$^+$-tree, and remove all strings that are equal to $s$ by fetching them from the disk. An update operation can typically be transformed into a delete operation and an insert operation, and hence we omit the details.

### 4.4 Range and KNN Search Algorithms

To determine the relationship between $q$ and each partition, we need to actually compute the edit distance between $q$ and each reference string. Hence, to efficiently answer both range and KNN queries, the number of reference strings cannot be too large. In this way, we maintain the information of each partition $P_i$, including the reference string $P_i.o$, minimum and maximum edit distance $P_i.l$ and $P_i.u$, in main memory.

Algorithm 1 shows how to answer range queries. First, we collect the search ranges of candidate regions and selectable regions by sequentially checking partitions based on the *Pruning Rule 2 and 3*, and maintain the search ranges using a list named $SR$ (line 1–6). Subsequently, we verify strings with their keys in these search ranges by traversing the $B^+-$tree once. To facilitate identification of these strings, we maintain a data structure $scan$ with search keys equal to $SR$, and employ function *btGetNextIndexTuple* to identify strings with their keys in search ranges progressively according to the ascending order of the keys (line 7–13). For each qualified string $s$, if its key lies in the selectable region, we add it to $\bar{S}$ directly (line 10–11); otherwise, we verify $s$ using Algorithm **??** and add $s$ to $\bar{S}$ if $ed(q, s) \leq \theta$ (line 12–13). Note that in our case it is able to avoid the random I/O problem because we maintain the clustered $B^+-$tree.




| id | o | P.l | P.u | Ed(o,q) | i*c+P.l | i*c+p.u | $\overline{Ed}$(o,q) | Search range |
|---|---|---|---|---|---|---|---|---|
| 0 | kaandorp, brigitte | 0 | 28 | 16 | 0 | 28 | 16 | [15, 17] |
| 1 | mann, amanda | 0 | 10 | 13 | | | | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 100 | danklmaier,florian | 0 | 12 | 10 | 7300 | 7312 | 7310 | [7309, 7311] |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

query: billmeier, sylvia  range threshold: 1  c = 73

Fig. 5. A Running Example of Executing a Range Search.

**Example 2. (Range Search)** *The table in Figure 5 shows the statistics of partitions for the actor name dataset taken from the IMDB database, where $c = 73$. Each partition is associated with an id, a reference string $o$, $P.l$, and $P.u$. Given a query "billmeier, sylvia" with $\theta = 1$, we first identify the search range for each partition (if any) shown in the last column and combine them together. We then verify candidates by traversing the $B^+-$tree once. By invoking function btGetNextIndexTuple, we locate key 15, the minimum search key of $P_0$, in the first leaf node of the $B^+$-tree. After verifying all qualified strings in $P_0$, btGetNextIndexTuple will switch to next leaf node that contains keys in the search ranges. In our example, the second leaf node is skipped as it does not contain any qualified strings.*

The KNN algorithm uses a top-down strategy. We transform KNN search into a sequence of range searches. Specifically, at the $i^{\text{th}}$ $(i \geq 1)$ iteration, we issue a range query with threshold $\theta = i - 1$, and refine the KNN candidates of $q$ by strings that have been probed. The iteration goes on until the maximum edit distance of strings in the KNN candidates is less than or equal to $\theta$.

Algorithm 2 describes the details to answer KNN search. Initially, we set search range $\theta$ to 0, current candidates $\bar{S}$ to $\emptyset$, and maximum edit distance $maxED$ between strings in $\bar{S}$ and $q$ to $\infty$ (line 1). We then perform a sequence of range searches (lines 2–22) until $maxED \leq \theta$ (line 17,21). At the $i^{\text{th}}$ iteration, to avoid redundant verification of strings probed in the previous iterations, we set search range to $[i*c+\theta, i*c+\theta]$ for a selectable partition and $[i*c+lb, i*c+lb]$, $[i*c+ub, i*c+ub]$ for a candidate partition, respectively (line 3–10). For a string $s$ in a selectable partition, on one hand, $ed(s, q) \leq \theta$ based on Pruning Rule 3; on the other hand, $ed(s, q) > \theta - 1$ since it was not discovered in the previous iterations. Therefore, we guarantee $ed(s, q) = \theta$ (line 14–15); for a string $s$ in a candidate partition, we verify it using $maxED$ as the threshold, and update $\bar{S}$ if necessary

**Algorithm 2:** KNNSearch($\mathbf{P}, btree, q, K$)

```
1  θ ← 0;   S̄ ← ∅;   maxED ← ∞;
2  while true do
3      SR ← ∅; // a list maintains search range for partitions
4      foreach P_i ∈ P do
5          if P_i is a selectable partition then
6              SR.add(⟨[i*c+θ, i*c+θ], true⟩);
7          else if P_i is a candidate partition then
8              SR.add(⟨[i*c+lb, i*c+lb], false⟩);
9              if lb ≠ ub then
10                 SR.add(⟨[i*c+ub, i*c+ub], false⟩);
11     scan.keys ← SR;
12     while btGetNextIndexTuple(btree, &scan) do
13         s ← getString(scan.τs);
14         if scan.selectable then
15             S̄ ← S̄ ∪ {⟨s, θ⟩};
16             update maxED by S̄ and K;
17             if maxED ≤ θ then return S̄;
18         else if VerifyED(s, q, maxED) ≠ −1 then
19             S̄ ← S̄ ∪ {⟨s, ed(s, q)⟩};
20             update maxED by S̄ and K;
21             if maxED ≤ θ then return S̄;
22     θ ← θ + 1;
```

| | | | | θ = 0 | θ = 1 | θ = 2 |
|---|---|---|---|---|---|---|
| id | o | Ed(o,q) | $\overline{Ed}$(o,q) | Search range | Search range | Search range |
| 0 | kaandorp, brigitte | 16 | 16 | [16,16] | [15,15],[17,17] | [14,14],[18,18] |
| 1 | mann, amanda | 13 | 86 | | | |
| ... | ... | ... | ... | ... | ... | ... |
| 100 | danklmaier,florian | 10 | 7310 | [7310,7310] | [7309,9309],[7311,9311] | [7308,9308],[7312,9312] |
| ... | ... | ... | ... | ... | ... | ... |

query: billmeier, sylvia K=16 c = 73

Fig. 6. A Running Example of Executing a KNN Search.

(line 18–19). When $\bar{S}$ is updated, we then refine $maxED$ accordingly and return $\bar{S}$ if $maxED \leq \theta$.

**Example 3.** *Consider a KNN query in the same setting as Example 2. We first issue a range search with $\theta = 0$, and the search range for each partition is described in the $5^{th}$ column of Figure 6. Note that $P_1$ is a prunable partition and no verification of strings in $P_1$ is required. Suppose that $maxED > 0$ after performing the first range search. We then issue another range search with $\theta = 1$, and the search range is described in the $6^{th}$ column. Note that search ranges in the first iteration are excluded. We iteratively conduct range queries until $K$ strings with the maximum distances to $q$ equal to or less than the current $\theta$.*

### 4.5 Cost Analysis

The computation cost of both Algorithm 1 (*RangeSearch*) and Algorithm 2 (*KNNSearch*) consists of three parts:(1) $T_c$, the time to compute the edit distance between the query and all reference strings; (2) $T_v$, the time to verify the edit distance between $q$ and all candidate strings; (3) the time to load the index nodes (if necessary), and strings lying in the selected partitions and candidate regions of candidate partitions from the disk. Let $\Lambda$ be the time to fetch a page from the disk. The time complexity of *RangeSearch* is:

$$T_{RS} = T_c + T_v + (I_{RS} + D_{RS}) \times \Lambda \quad (3)$$

where $I_{RS}$ and $D_{RS}$ are the number of index nodes and data nodes that are loaded to main memory, respectively during the range search. The time complexity of *KNNSearch* is:

$$T_{KS} = T_c + T_v + (I_{KNN} + D_{KNN}) \times \Lambda \quad (4)$$

where $I_{KNN}$ and $D_{KNN}$ are the number of index nodes and data nodes that are loaded to main memory, respectively during the KNN search. Note that for range search or KNN search at each iteration, a data node is at most loaded to main memory once due to the consistency of our storage strategy and access mechanism which are previously described.

## 5 Distribution-Aware Partitioning

Regarding the query cost presented in the previous section, $T_c$ remains constant once the reference strings are selected, and $T_v$ relies on strings lying in candidate region of each partition $P_i$. For the I/O cost, besides requesting and loading candidate strings (in the candidate regions) to main memory, it also consists of loading strings in the selectable partitions. According to the definition of *selectable partition*, to qualify $P_i$ as a selectable partition, we need to guarantee that $ed(o, q) + P_i.u \leq \theta$. However, in practice, due to the high dimensionality of the string domain [13], given a query $q$ and a threshold $\theta$, the edit distance from $q$ to $o$ is often larger compared with the threshold $\theta$ (small $\theta$ is required in order to achieve meaningful results). In this way, the number of strings lying in the selectable partitions should be rather small. Hence, our optimization objective to reduce the query cost is to minimize the number of strings in the candidate regions.

Recall that our approach splits the string collection into partitions and assigns each string in the collection to a proper partition. To achieve the best query performance by minimizing the number of strings in the candidate regions, we need to consider the following two concerns carefully:

- choosing appropriate reference strings
- assigning each string to a reference string optimally.

In this section, we first elaborate on how to assign each string to a reference string such that the number of strings in the candidate regions is minimized, and then describe how to choose proper reference strings.

### 5.1 Partition Assignment

Suppose we have obtained a set of partitions and their reference strings. *Which partition should a new string be*

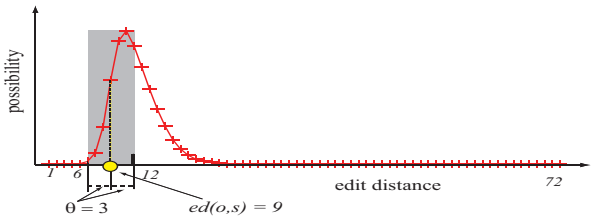



Fig. 7. A Running Example of Computing Relative Verification Likelihood

*assigned to such that its likelihood of avoiding verification is maximized?* To answer this question, traditional clustering methods involve assigning the new string to the partition with the minimum edit distance or maximum edit distance (hierarchical clustering methods). For example, in iDistance, the whole data space is divided into Voronoi cells, i.e., each point is assigned to the reference point with the minimum distance. In this way, ranges formed by the data partitions that intersect with the query range are examined. This partitioning strategy works well in the low-dimensional space since many of the partitions do not overlap with the query range can be pruned. However, regarding our application scenario, due to the high dimensionality of the string domain [13], most of the partitions overlap with the query range. Therefore, the pruning power is degraded significantly. To achieve the optimal query performance, instead, we aim to minimize the expected number of strings that need to be verified per query.

Given a reference string $o$, let $f_o(q,i)$ be the probability density function (PDF) over a universal string domain $U$ that describes the probability that $ed(o,q) = i, \forall q \in U$. Formally,

$$f_o(q,i) = \frac{|\{q|\forall q \in U, ed(o,q) = i\}|}{|U|} \tag{5}$$

Deriving the actual PDF of each reference string $o$ for a universal string domain $U$ is impossible. Hence, we approximate the PDF by using $S$ or a sample of $S$ instead of $U$. In practice, we can also approximate $f_o(q,i)$ after collecting a proper number of queries and using these queries as $U$.

**Lemma 1. (Relative Verification Likelihood)** *Given a string $s$ that belongs to partition $P$, when we issue a range query with threshold $\theta$, the likelihood that $s$ needs to be verified with respect to reference string $o$ is:*

$$\rho_o(s) = \sum_{L \le i \le U} f_o(q,i) \tag{6}$$

*where* $L$=max$\{P.l, |ed(o,s)-\theta|\}$, $U$=min$\{P.u, |ed(o,s)|+\theta\}$.

**Example 4. (Relative Verification Likelihood)** *Figure 7 shows the PDF, $f_o(q,i)$, for a particular reference string $o$, using actor names taken from the IMDB database. Suppose that the edit distance between $o$ and some string $s$ is 9. If a user issues a range query with query string $q$ and $\theta = 3$, then $s$ does not have to be verified when $ed(o,q) > 12$ or $ed(o,q) < 6$. Thus, the likelihood that $s$ needs to be verified is equal to $\sum_{6 \le i \le 12} f_o(q,i)$ (the gray region in the figure).*

Equation 6 expresses the likelihood that $s$ will have to be verified with respect to reference string $o$. Therefore, given a set of strings $S$, the *expected number* of strings in $S$ that need to be verified with respect to reference string $o$ for a range query is:

$$E_o(S) = \sum_{s \in S} \rho_o(s). \tag{7}$$

In order to decrease the verification likelihood for $S$, we can leverage a set of reference strings $O$ and assign strings optimally among multiple reference strings. Assuming that we are given a set of reference strings, the partition assignment can be accomplished as follows.

**Definition 7. (Verification Likelihood)** *Given a set of reference strings $O$, the verification likelihood of string $s$ is defined as:*

$$\psi_O(s) = \min_{o \in O}\{\rho_o(s)\}. \tag{8}$$

**Lemma 2. (Partition Assignment)** *Given a set of reference strings $O$, the expected number of strings from $S$ that need to be verified is:*

$$E_O(S) = \sum_{s \in S} \psi_O(s). \tag{9}$$

*The objective function of a partition assignment is to minimize $E_O(S)$.*

The distribution based partitioning method greedily assigns each string to the reference string that yields the minimum relative verification likelihood. In this way, once the reference strings are selected, the expected number of strings from $S$ that need to be verified is minimized. Given that the query threshold $\theta$ is unknown in advance, in order to perform distribution based partitioning we need to use a pre-defined threshold $\alpha$ for computing the verification likelihood for each reference string. A natural question is how to select a good value for $\alpha$. From our experiments we observe that the expected number of strings from $S$ that need to be verified varies slightly with different $\alpha$ for a given string distribution, which makes our approach overall insensitive to this parameter. In order to approximately achieve the best performance, one could set $\alpha$ to the maximum frequently used $\theta$ across all queries.

### 5.2 Selection of the Reference Strings

Given a collection of strings $S$, our objective is to select $N$ reference strings $O$ such that $E_O(S)$ can be minimized.

**Lemma 3.** *The problem of selecting a set of reference strings $O$ such that $E_O(S)$ is minimized is NP-hard.*

*Proof sketch:* Our problem is essentially the $K$-Means problem with a different optimization objective. Hence, we can reduce the $K$-means problem, which is NP-hard, to our problem. Besides, it is also NP-hard to find *Median String* and *$N$-Median Strings* as well [16].

**Algorithm 3:** pivotSel($S, N$)

```
1 S ← select a sample from S; //  use S instead of S
2 T ← select a sample from S; //  extract O from T
3 foreach s ∈ T do
4     compute f_s(q, i) over S;
5 O ← ∅;
6 for i = 1 to N do
7     select o ∈ T such that E_{O∪{o}}(S) is minimized;
8     O ← O ∪ {o};
9 return O;
```

- **Median String Problem:** Given a set of strings $S$, find a string $o \in U$ such that $\sum_{s \in S} ed(o, s)$ is minimized;
- **N-median Strings Problem:** Given a set of strings $S$, find a set of strings $\bar{S} \subset U$ such that $|\bar{S}| = N$ and $\sum_{s \in S} \min\{ed(s, \bar{s}) | \bar{s} \in \bar{S}\}$ is minimized.

□

To address our problem practically, we assume $O \subset S$ *and set* $N = |O|$ *to be a tunable, user defined parameter*. Based on the above assumption, we propose a heuristic approach to extract the reference strings greedily. Algorithm 3 gives the pseudo-code of our proposed approach. To enable fast reference string selection, we select two samples $\mathcal{S}$ and $\mathcal{T}$ from the string collection $S$ (line 1–2). We extract $O$ from $\mathcal{T}$ and $\forall s \in \mathcal{T}$, we compute $f_s(q, i)$ based on objects in $\mathcal{S}$ according to Equation 5 (line 3–4). We then iteratively extract reference strings in a greedy manner (line 6–8) and finally return $O$ as the reference strings.

The time complexity of Algorithm 3 consists of two parts. The first part has complexity $O(\Sigma_{s \in \mathcal{T}} \Sigma_{s' \in \mathcal{S}} |s| \times |s'|)$ to compute the PDF for each $s$. The second part has complexity $O(N \times |\mathcal{T}| \times |\mathcal{S}|)$ to obtain $O$. To select a reference string $o$, we need to identify the minimum verification likelihood for each $s' \in \mathcal{S}$ by checking each $s \in \mathcal{T}$ that has not been selected. As we can use an array to maintain the current minimum verification likelihood for each $s'$, the time complexity of selecting a reference string $o$ is $O(|\mathcal{T}| \times |\mathcal{S}|)$.

**Remark.** Although we consider the number $N$ of reference strings as a tuning parameter, the best $N$ depends on the application domains, including the average length $AL$, the deviation $DL$ of the lengths for the strings in the dataset. According to our experiments, setting $N = 2000$ and $N = 500$ can achieve approximately best query performance for the dataset with $AL \approx 15, DL \approx 4$ and $AL \approx 67, DL \approx 25$, respectively. Generally, for a dataset with long strings, $N$ should be set to a relatively small value since computing the edit distances between queries and reference strings are costly while it is effective to apply other filtering techniques, such as length filtering, character count filtering to do the verification. Rather, for a dataset with short strings, $N$ could be set to a relatively large value. Since strings are short, the pruning power of applying length filtering and character count filtering techniques degrades. By introducing a larger number of reference strings, the candidate set size can be reduced. Theoretically, the optimal $N$ is able to achieve when by enlarging $N$, the saved verification cost (due to the decrease of the candidates) is less than the increase computation cost (due to the increase of the reference strings).

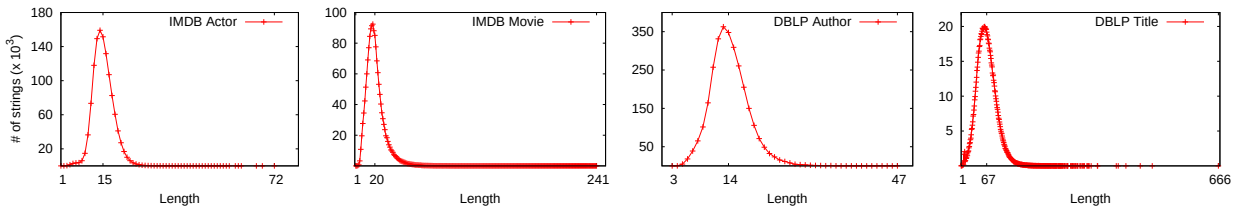
## 6 Experimental Evaluation

We evaluate state-of-the-art approaches as well as our approach in the experiments. For reference, we also compare the proposed approach with other in-memory approaches and the results can be found in the appendix.

- **EXH** is an exhaustive approach that sequentially verifies strings in the dataset using the verification Algorithm in [13]. For the KNN search, it takes the first $K$ strings as the candidate set and then refines the candidate set by verifying the remaining strings.
- **Flamingo** is one of the latest $n$-gram based inverted index approaches. Note that Flamingo is continuously updated by assimilating other $n$-gram based techniques, and we use its latest version. As discussed in Section 2, it may lead to *false negatives* for KNN search since in their method, they assume that any result must share at least one $n$-gram in common with the query.
- $B^{\mathbf{ed}}$**-tree**[13] organizes strings in the dataset as a $B^+$-tree according to a certain order. For each experiment, we enumerate all the orders and present the result based on the order that achieves the best performance.
- **PBI** is our proposed **P**artitioning **B**ased **I**ndexing approach.

Except EXH, either the source code or the binary code of the other approaches is generously provided by the authors. All approaches are implemented in C++ and compiled using GCC 4.1.2 with "-O3" flag. By default, the size of both index and data pages is set to 8KB, and the buffer size is set to 16MB.

We use four publicly available real datasets from [13] and show the statistics of the datasets in Table 3. To better illustrate the datasets, we also show the string length distribution of each dataset in Figure 8. The query workload for each dataset is also provided by the authors in [13]. String length distribution of the query workload is shown in Table 4. As we can see, the average length and variance of the query workload basically follow the same distributions as those shown in Table 3. All experiments are conducted on a PC with Intel X3430 2.4GHz processor, 8GB of memory, and CentOS 5.5 operating system. We evaluate the performance of all approaches in terms of index construction time, index size, average query response time, average candidate set size and average number of I/Os.




(a) IMDB Actor (b) IMDB Movie (c) DBLP Author (d) DBLP Title

Fig. 8. String Length Distribution

| Dataset | Cardinality | Size (M) | Min. | Max. | Avg. | Var. |
|---|---|---|---|---|---|---|
| Actor | 1213391 | 19 | 1 | 72 | 15 | 3 |
| Movie | 1568885 | 41 | 1 | 240 | 19 | 9 |
| Author | 2948929 | 67 | 3 | 47 | 14 | 4 |
| Title | 1158648 | 84 | 1 | 666 | 67 | 25 |

TABLE 3
Statistics of the datasets

| Dataset | # of Queries | Min. | Max. | Avg. | Var. |
|---|---|---|---|---|---|
| Actor | 100 | 3 | 26 | 15 | 4 |
| Movie | 100 | 5 | 59 | 20 | 10 |
| Author | 100 | 6 | 21 | 14 | 3 |
| Title | 100 | 1 | 156 | 65 | 27 |

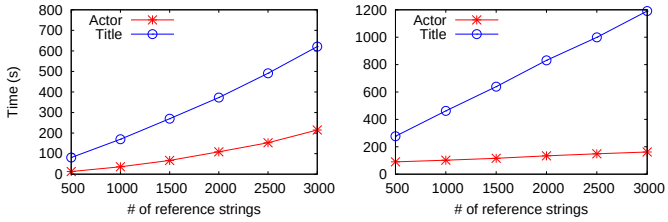
TABLE 4
String Length Distribution of the Query Workload

### 6.1 Partitioning Based Indexing

We first evaluate the performance of index construction for PBI and then analyze the parameters that potentially affect the performance of PBI. By default, we evaluate the performance over Actor and Title datasets, and set $N = 2000$, $\theta = 4$.

#### 6.1.1 Index Construction

The number of reference strings $N$ varies from 500 to 3000. To properly leverage the quality and the efficiency of extracting reference strings, $|\mathcal{S}|$ and $|\mathcal{T}|$ used in Algorithm 3 are set to 20,000 and $4\times N$, respectively.




(a) Selection of Reference Strings (b) Partition Assignment

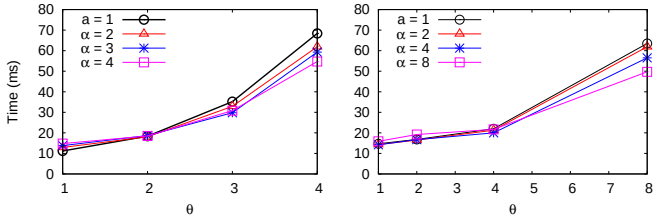
Fig. 9. Index Construction for PBI

As discussed, the index construction of PBI consists of three stages: (1) selection of reference strings, (2) partition assignment, and (3) $B^+$-tree construction. Recall that in stage (1), we first compute the PDF for each string in $\mathcal{T}$ over $\mathcal{S}$, and then sequentially select the reference strings from $\mathcal{T}$. Besides, in stage (2), each string in dataset $S$ is assigned to the reference string with the minimum verification likelihood. Since building the index is an offline process, to speedup the construction, we parallelize stage (1) and (2) so that each process unit takes similar number of strings in $\mathcal{T}$ for stage (1) and $S$ for stage (2). Specifically, we issue 40 threads to perform stage (1). Regarding stage (2), we can process the partition assignment for each string individually. Hence, we employ MapReduce [29], which is a widely accepted framework for distributed computing over shared-nothing clusters, to do the partition assignment. The configuration of the cluster can be found in [30]. We use 40 computing nodes, and each node is configured to run three reduce tasks. Figure 9(a) plots the selection cost. We can observe that the cost increases first linearly and then tends to be quadratic when $N$ varies. Regarding the composition of the selection cost, the PDF computation increases linearly while selecting the reference strings increases quadratically due to the time complexity $O(N\times|\mathcal{T}|\times|\mathcal{S}|)$. When $N$ is small, the PDF computation dominates the overall cost while selecting the reference strings takes a growing proportion when $N$ increases. Besides, as the average length of strings in Title dataset is larger than that in Actor dataset, the selection cost over Title dataset is more expensive than that over Actor dataset. Figure 9(b) plots the partition assignment cost. As expected, the cost increases linearly when $N$ varies, and the slope relies on the length of the strings in the dataset. The time it takes to construct the $B^+$-trees for Actor and Title datasets is 1.14s and 1.13s respectively, which only depends on the cardinality of the dataset and remains constant when $N$ varies.

#### 6.1.2 Effect of Parameters

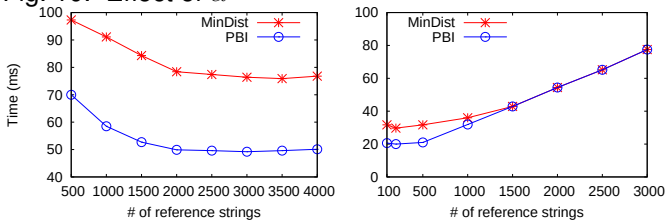
We first investigate the query performance as a function of parameter $\alpha$. Figure 10 plots the results. We observe that in each case the query performance is the best when $\alpha \simeq \theta$, and varies slightly using different $\alpha$, which makes our approach overall insensitive to this parameter. Nevertheless, when the query threshold $\theta$ varies, the average query response time increases more and more rapidly and the time gap of using different $\alpha$ becomes larger. In general, to approximately achieve the best performance across different query thresholds $\theta$, we can select the maximum frequently used $\theta$ as $\alpha$. Hence, in the remaining experiments, we use $\alpha = 4$ for Actor, Movie, Author datasets, and $\alpha = 8$ for Title dataset.

We then study the effect of varying $N$. To demonstrate the effectiveness of our proposed partitioning strategy, we also test an alternative partitioning strategy in which




(a) Actor (b) Title

Fig. 10. Effect of $\alpha$



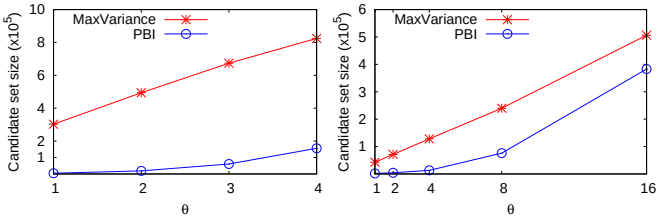

(a) Actor (b) Title

Fig. 11. Effect of $N$

each string is assigned to the reference string with the minimum edit distance. We refer to this approach as *MinDist*. In MinDist, the reference strings are extracted randomly. Figure 11 plots the results. For Actor dataset, increasing the number of partitions results in a decreasing cost of answering queries. Nevertheless, there is an obvious trend of diminishing returns. This is due to the fact that the cost of computing the edit distance between the query and the reference strings becomes comparable to the cost of verifying candidates, and outweighs the cost savings from pruning smaller and smaller partitions (recall that computing is much costlier than verifying). This is also confirmed in Figure 11(b). For Title dataset, when we increase the number of partitions, the query performance degrades. Recall that strings in Title dataset have much larger average length compared with that of the remaining three datasets, and computing the edit distance between the query and the references dominates the overall cost. We can also observe that PBI can achieve at lease 30% gain compared with MinDist when the edit distance computation cost is not the dominant factor.

We also evaluate the effect of using different strategies to select reference strings by comparing [26] (referred as *MaxVariance*) with PBI. Note that we also assign each object to a single reference string in MaxVariance. Figure 12 plots the results. We observe that the number of candidates in MaxVariance is 7–70 times, 1.3–25 times larger than that in PBI over Actor and Title datasets, respectively. The main reason is that in MaxVariance, although for each reference string $o$, strings close or far way to $o$ are assigned to $o$, there still exist many strings for which we cannot find effective reference strings to be assigned due to the high dimensionality property of the string domain. For those strings, the pruning power degrades significantly. While in PBI, we assign each string to the reference string that leads to a minimum verification cost, that's why PBI performs better than MaxVariance.

In the rest, we set $N = 2000$ for Actor, Movie, and




(a) Actor (b) Title

Fig. 12. Effect of Partitioning Strategies

| Dataset | $B^{ed}$-Tree | Flamingo | PBI |
|---|---|---|---|
| Actor | 36 | 3.87 | 243 |
| Movie | 46 | 6.41 | 320 |
| Author | 84 | 9.19 | 356 |
| Title | 17 | 11.42 | 1204 |

TABLE 5
Index Construction Time (s)

Author datasets, and $N = 500$ for Title dataset.

### 6.2 Comparison With Alternatives

We then evaluate the performance of PBI in comparison with existing approaches in term of index construction time, index size, range queries and KNN queries.

#### 6.2.1 *Index construction time*

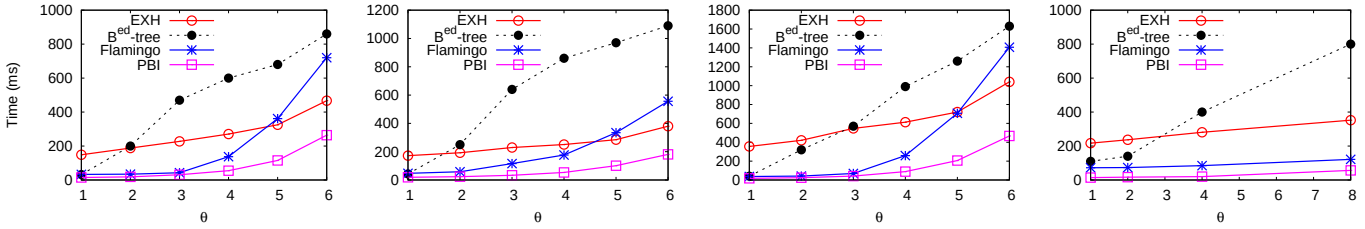
Table 5 summarizes the index construction time. We show the construction time of PBI under the setting which is described in the previous section. We can see that Flamingo takes the minimal construction time, followed by $B^{\text{ed}}$-tree and PBI. Although PBI takes the maximal construction time, we argue that building the index is an off-line process, and it can be easily parallelized. The construction time can be reduced by adding more computing resources.

#### 6.2.2 *Index size*

Table 6 summarizes the index size. We can observe that Flamingo have the largest index sizes (about 6 times of the data size). On the other hand, PBI almost has the smallest index size, followed by $B^{ed}$-tree. Note that the index size of PBI only relies on the cardinality of the dataset, not the length of the strings, while that of the other approaches rely on both.

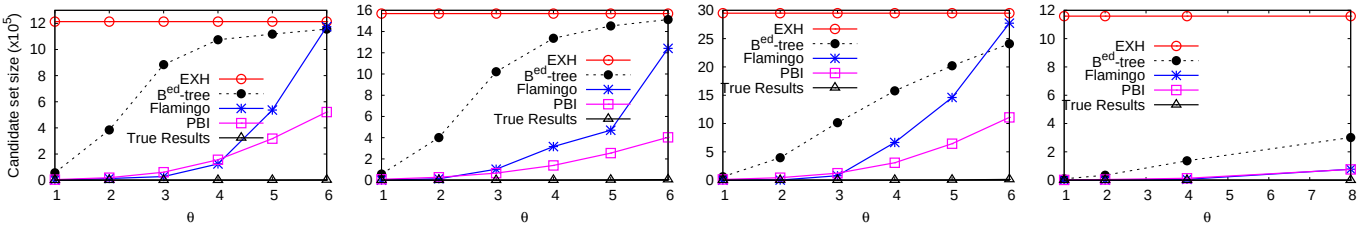
#### 6.2.3 *Range Queries*

Figure 13 plots the average response time as a function of query threshold $\theta$. The results show that PBI outperforms the other approaches on all datasets regardless of $\theta$. When $\theta$ is small (1 and 2), both Flamingo and $B^{\text{ed}}$-tree perform better than EXH. However, the performance of $B^{\text{ed}}$-tree degrades rather dramatically as $\theta$ increases. The performance of Flamingo over the first three datasets degrades smoothly when $\theta$ varies from 1 to 3, but dramatically when $\theta$ varies from 4 to 6. It even performs worse than EXH when $\theta$ varies from 5 to 6. To identify why, we show the average candidate set size and



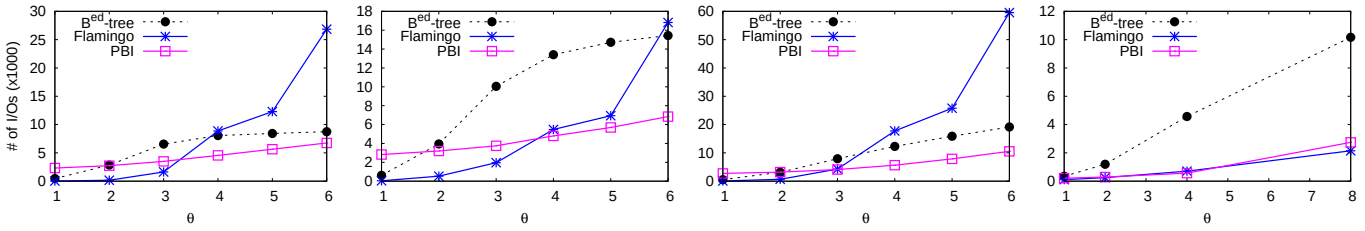

(a) IMDB Actor (b) IMDB Movie (c) DBLP Author (d) DBLP Title

Fig. 13. Average Response Time for Range Queries



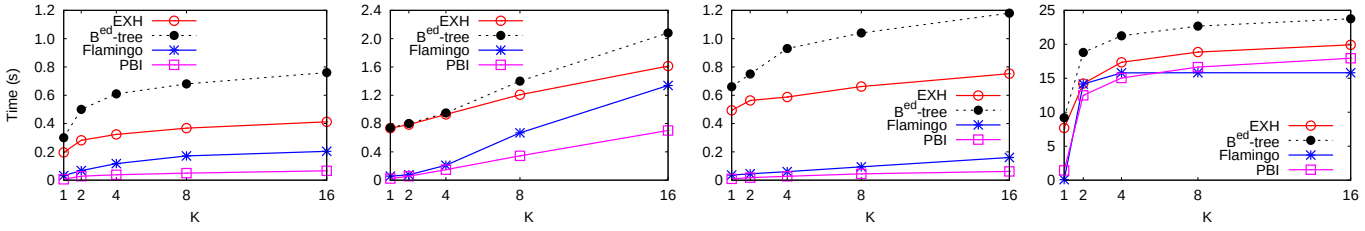

(a) IMDB Actor (b) IMDB Movie (c) DBLP Author (d) DBLP Title

Fig. 14. Average Candidate Set Size for Range Queries




(a) IMDB Actor (b) IMDB Movie (c) DBLP Author (d) DBLP Title

Fig. 15. Average Number of I/Os for Range Queries




(a) IMDB Actor (b) IMDB Movie (c) DBLP Author (d) DBLP Title

Fig. 16. Average Response Time for KNN Queries

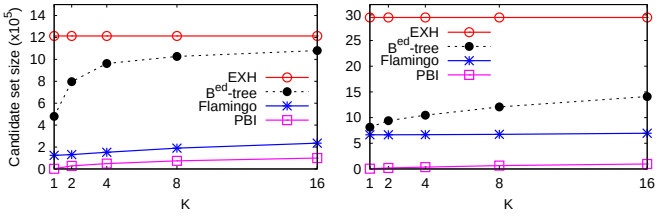
| Dataset | Data Size | $B^{ed}$-Tree | Flamingo | PBI |
|---|---|---|---|---|
| Actor | 19 | 36 | 116 | 46 |
| Movie | 31 | 63 | 206 | 57 |
| Author | 43 | 93 | 286 | 98 |
| Title | 75 | 155 | 397 | 41 |

TABLE 6
Index size (MB)

average number of I/Os for each experiment in Figure 14 and Figure 15, respectively. We can observe that, basically, the maximum number of strings that need to be verified is EXH, followed by $B^{ed}$-tree. The reason, as already argued in [13], is that the effectiveness of string orders are negatively affected when the average length of the strings is short. When $\theta$ is small, the number of strings that need to be verified is rather similar using Flamingo and PBI. However, when $\theta$ varies from 4 to 6, the number of verified strings using Flamingo increases significantly. For Flamingo, on one hand, a large $\theta$ leads to a large number of strings that need to be verified; on the other hand, this approach results in false negatives for many queries when $\theta$ is large, and a sequential scan is required in this case. Due to random accesses to the data file, the I/O cost degrades rather dramatically for Flamingo when the number of candidates is large (over the first three datasets). The I/O cost for $B^{ed}$-tree increases smoothly for the datasets with short strings while degrades rather dramatically for the datasets with long strings.

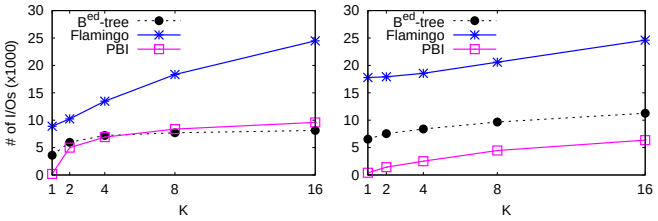
### 6.2.4 KNN Queries

For reference, Table 7 shows the average maximum edit distances of the $K$ nearest neighbors for all queries in each experiment. Figure 16 plots the KNN query




(a) IMDB Actor (b) DBLP Author

Fig. 17. Average Candidate Set Size for KNN Queries




(a) IMDB Actor (b) DBLP Author

Fig. 18. Average Number of I/Os for KNN Queries

performance as a function of $K$. From the figure, we can observe that PBI performs the best among all the approaches over the first three datasets. In particular, when $K = 16$, PBI performs 2-3 times faster than Flamingo, 2-12 times faster than EXH, and 3-19 times faster than $B^{ed}$-tree. Although Flamingo performs better than PBI when $K \geq 8$ over Title dataset, as we can see in Table 8, Flamingo starts to omit partial results. An interesting observation is that the performance of all techniques over Title dataset seems to deteriorate significantly (more than 10 seconds) when $K \geq 2$. This is because for Title dataset, the average length and variance of the strings are large. In this case, issuing a range query with a small threshold often returns no results. Thus, in order to identify $K$ results, a large threshold is required, which results in many iterations and increased cost (see Table 7). We also show the average candidate set size and I/O cost in Figure 17 and Figure 18, respectively. Due to space limitations, we do not show the results for Movie and Title datasets. In general, EXH has the largest candidate set size, followed by $B^{\text{ed}}$-tree, Flamingo, PBI. This finding is not surprising since PBI can answer KNN queries progressively and any string in the candidate set is probed at most once. Notice that in Flamingo, a string might be traversed across multiple iterations. The I/O cost for PBI and $B^{\text{ed}}$-tree increases rather smoothly for all the datasets. Due to random accesses to the data file, the I/O cost of Flamingo deteriorate significantly.

### 6.2.5 Scalability

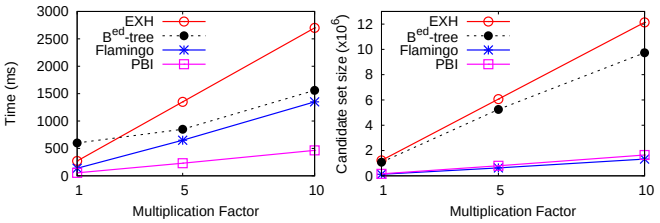
We now investigate the scalability of four approaches in term of answering range queries where $\theta = 4$. We enlarge the Actor dataset by 5 times and 10 times as follows: (1) we first extract distinct first names and distinct last names. Specifically, there are 147,775 different first names and 368,338 last names; (2) we then randomly select first name and last name to combine an actor name; (3) we generate other 4 * 1213391 and 9 * 1213391 strings, respectively. From Figure 19, we can see that both running time and the candidate set size of all the four approaches increases linearly when we enlarge the data size. Due to the same reasons discussed above, PBI performs the best, followed by Flamingo, B$^{ed}$-tree, and EXH While PBI and Flamingo generate the minimum number of candidates, followed by B$^{ed}$-tree, and EXH.

| Dataset | K=1 | K=2 | K=4 | K=8 | K=16 |
|---|---|---|---|---|---|
| Actor | $\theta = 0$ | $\theta = 2.5$ | $\theta = 3.5$ | $\theta = 4.0$ | $\theta = 4.3$ |
| Movie | $\theta = 0$ | $\theta = 0.25$ | $\theta = 1.4$ | $\theta = 4.8$ | $\theta = 8.1$ |
| Author | $\theta = 0$ | $\theta = 0.4$ | $\theta = 0.8$ | $\theta = 1.5$ | $\theta = 2.1$ |
| Title | $\theta = 0$ | $\theta = 27$ | $\theta = 31$ | $\theta = 32$ | $\theta = 34$ |

TABLE 7
Average Edit Distances of the $K^{\text{th}}$ NN.

| Dataset | $K$=1 | $K$=2 | $K$=4 | $K$=8 | $K$=16 |
|---|---|---|---|---|---|
| Actor | 0 | 0 | 0 | 0.01 | 0.22 |
| Movie | 0 | 0 | 0 | 0.04 | 2.5 |
| Author | 0 | 0 | 0 | 0.03 | 0.19 |
| Title | 0 | 0 | 0 | 0.24 | 5.47 |

TABLE 8
Average Number of Lost Results in Flamingo




(a) Time (b) Candidate Set Size

Fig. 19. Scalability

### 6.2.6 Summary of New Findings

- There does not exist a single approach that can outperform the others for all settings, while in most cases, PBI performs the best.
- Flamingo works efficiently when $\theta$ is small. However, its performance degrades significantly and is even worse than that of EXH when $\theta$ is large. Besides, it may lead to false positives for answering KNN queries.
- $B^{ed}$-tree degenerates to partially sequential scans of the tree when the average length of strings in the dataset is short.
- No approach works well for KNN queries on datasets with very long strings. In fact, we argue that in this case using other similarity measures, like Jaccard or cosince, might be able to extract more meaningful results (see the average edit distance of the 2$^{\text{th}}$ NN over Title dataset is 27 while the average length of strings is 67).

## 7 Conclusion

In this paper, we propose partitioning based approaches that employ the $B^+$-trees to answer both range and KNN

queries. We demonstrate that optimal partitioning of the dataset is an NP-hard problem. Hence, we propose a heuristic approach for extracting the reference strings greedily and design an optimal partition assignment scheme to minimize the candidate set size. Through extensive experiments over four real datasets, in comparison with state-of-the-art techniques, we demonstrate that our proposed approach provides superior performance for most cases in term of index size, query response time, average candidate set size, and I/O cost.

## ACKNOWLEDGMENTS

The work in this paper was in part supported by the Singapore Ministry of Education Grant No. R-252-000-454-112. Xiaoyong Du was partially supported by NSF China Grant 61170010.

## REFERENCES

[1] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava, "Benchmarking declarative approximate selection predicates," in *SIGMOD*, 2007, pp. 353–364.
[2] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik, "Example-driven design of efficient record matching queries," in *VLDB*, 2007, pp. 327–338.
[3] 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*, 2001, pp. 491–500.
[4] C. Li, B. Wang, and X. Yang, "Vgram: Improving performance of approximate queries on string collections using variable-length grams," in *VLDB*, 2007, pp. 303–314.
[5] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.
[6] R. Vernicaand and C. Li, "Efficient top-k algorithms for fuzzy search in string collections," in *KEYS '09: Proceedings of the First International Workshop on Keyword Search on Structured Data*, 2009, pp. 9–14.
[7] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin, "Efficient exact edit similarity query processing with the asymmetric signature scheme," in *SIGMOD Conference*, 2011, pp. 1033–1044.
[8] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *SIGMOD Conference*, 2012, pp. 85–96.
[9] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in *ICDE*, 2009, pp. 604–615.
[10] A. Behm, C. Li, and M. J. Carey, "Answering approximate string queries on large data sets using external memory," in *ICDE*, 2011, pp. 888–899.
[11] S. Ji, G. Li, Li, and J. Feng, "Efficient interactive fuzzy keyword search," in *WWW*, 2009, pp. 371–380.
[12] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," in *SIGMOD*, 2009, pp. 707–718.
[13] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava, "Bed-tree: an all-purpose index structure for string similarity search based on edit distance," in *SIGMOD*, 2010, pp. 915–926.
[14] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish, "Indexing the distance: An efficient method to knn processing," in *VLDB*, 2001, pp. 421–430.
[15] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "idistance: An adaptive $b^+$-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.
[16] C. de la Higuera and F. Casacuberta, "Topology of strings: median string is np-complete," *Theor. Comput. Sci.*, vol. 230, no. 1-2, pp. 39–48, 2000.
[17] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006, p. 5.
[18] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 918–929.
[19] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition-based method for similarity joins," *PVLDB*, vol. 5, no. 3, pp. 253–264, 2011.
[20] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *SIGMOD*, 2004, pp. 743–754.
[21] M.-S. Kim, K. young Whang, J.-G. Lee, and M. jae Lee, "n-gram/2l: A space and time efficient two-level n-gram inverted index structure," in *VLDB*, 2005, pp. 325–336.
[22] G. Li, S. Ji, C. Li, and J. Feng, "Efficient fuzzy full-text type-ahead search," *VLDB J.*, vol. 20, no. 4, pp. 617–640, 2011.
[23] D. Deng, G. Li, J. Feng, and W.-S. Li, "Top-k string similarity search with edit-distance constraints," in *ICDE*, 2013, pp. 925–936.
[24] L. Jin, C. Li, and S. Mehrotra, "Efficient record linkage in large data sets," in *DASFAA*, 2003, pp. 137–146.
[25] L. Jin, C. Li, and R. Vernica, "Sepia: estimating selectivities of approximate string predicates in large databases," *VLDB J.*, vol. 17, no. 5, pp. 1213–1229, 2008.
[26] J. Venkateswaran, D. Lachwani, T. Kahveci, and C. M. Jermaine, "Reference-based indexing of sequence databases," in *VLDB*, 2006, pp. 906–917.
[27] W. J. Masek and M. Paterson, "A faster algorithm computing string edit distances," *J. Comput. Syst. Sci.*, vol. 20, no. 1, pp. 18–31, 1980.
[28] M. S. Waterman, *Introduction to computational biology - maps, sequences, and genomes: interdisciplinary statistics*. CRC Press, 1995.
[29] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[30] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient processing of k nearest neighbor joins using mapreduce," *PVLDB*, vol. 5, no. 10, pp. 1016–1027, 2012.

**Wei Lu** is a research fellow at NUS, Singapore. He received his Ph.D degree in computer science from Renmin University of China in 2011. His research interest includes query processing in the context of spatiotemporal, cloud database systems and applications.

**Xiaoyong Du** is a professor at Renmin University of China. He received his Ph.D. degree from Nagoya Institute of Technology in 1997. His research focuses on intelligent information retrieval, high performance database and unstructured data management.

**Marios Hadjieleftheriou** received the electrical and computer engineering diploma in 1998 from the National Technical University of Athens, Greece, and the PhD degree in computer science from the University of California, Riverside, in 2004. He has worked as a research associate at Boston University. Currently, he is working for AT&T Labs Research. His research interests include in the areas of databases and data management in general.

**Beng Chin Ooi** eng Chin Ooieng Chin OoiBis a distinguished Professor of Computer Science at the National University of Singapore (NUS). He obtained his BSc (1st Class Honors) and PhD from Monash University, Australia, in 1985 and 1989 respectively. Beng Chin's research interests include database system architectures, performance issues, indexing techniques and query processing, in the context of multimedia, spatio-temporal, distributed, parallel, P2P, and Cloud database systems and applications. Beng Chin is the recipient of ACM SIGMOD 2009 Contributions award, a co-winner of the 2011 Singapore President's Science Award, the recipient of 2012 IEEE Computer Society Kanai award and 2013 NUS Outstanding Researcher Award. He is a fellow of the ACM and IEEE.