IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received August 24, 2020, accepted August 28, 2020, date of publication September 7, 2020, date of current version January 14, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3022057

A Tree-Based Indexing Approach for Diverse Textual Similarity Search

MINGHE YU[1,2], CHENGLIANG CHAI[3], AND GE YU[4], (Senior Member, IEEE)
[1]Software College, Northeastern University, Shenyang 110819, China
[2]Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, Shenzhen 518060, China
[3]Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[4]School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

Corresponding author: Minghe Yu (yuminghe@mail.neu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61902055, in part by the China Postdoctoral Science Foundation under Grant 2019M651134, in part by the Guangdong Province Key Laboratory of Popular High Performance Computers under Grant 2017B030314073, and in part by the Fundamental Research Funds for the Central Universities under Grant N181703006.

ABSTRACT Textual information is ubiquitous in our lives and is becoming an important component of our cognitive society. In the age of big data, we consistently need to traverse substantial amounts of data even to find a little information. To quickly acquire effective information, it is necessary to implement a textual similarity search based on an appropriate index structure to efficiently find results. In this article, we study top-k textual similarity search and develop a tree-based indexing approach that can construct indices to support various similarity functions. Our indexing approach clusters similar records in the same branch offline to improve the performance of online search. Based on the index tree, we present a top-k search algorithm with efficient pruning techniques. The experimental results demonstrate that our algorithm can achieve higher performance and better scalability than the baseline method.

INDEX TERMS Tree-based indexing, top-k similarity search, textual similarity.

I. INTRODUCTION

Textual information exists everywhere in our lives, and it is an important component in constructing a cognitive society. As we live in an era characterized by dramatic, rapid change, we consistently need to traverse a considerable amount of data to find only a few pieces of information. Textual similarity search retrieves a set of input strings and finds all strings similar to the given query. For example, suppose that we want to find publications about "Data Mining in Relational Databases" from Google Scholar. The system not only returns the records containing all five keywords but also retrieves records with some of the keywords in the query. Most of the results do not contain all of the keywords.

There are various applications designed based on textual similarity search. First, it can help us with information retrieval. If we want to find a message in our e-mail inbox, we can use only a few keywords to do so. As we input more keywords, we can narrow the range and locate the correct message. Second, we can use similarity search to integrate and enrich knowledge bases. This is important because a given object might have different names in the different knowledge bases. For example, "Avengers 4" and "Avengers: Endgame" are two objects in different knowledge bases. If we use one object to find similar objects in another knowledge base, after comparing their textual description, it can easily be determined that these two objects represent the same movie. We can integrate them and use both sets of information to enrich the knowledge bases. Third, textual similarity searches can be used for data clustering. We can classify a given dataset into several categories based on textual similarity. For example, when we create a new ontology in a knowledge base, it can be classified into the category containing the ontologies most similar to it.

All of the above applications utilize specific similarity search models. This means that their results are calculated by a specific similarity measure. However, it is not always clear which similarity measure is appropriate. It is necessary to test different measures to identify the best one for the given search task. In addition, some applications present results that

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

demand multiple similarity measures. For example, in genetic engineering, scientists need to compare gene segments and DNA sequences. When sequencing DNA, we need to compare the orders of bases in DNA sequences, which can help us to determine gene mutations. We can also calculate the number of common gene segments between creatures for biological classification. Obviously, these two tasks demand different similarity measures. In some cases, they are subtasks of the same mission (such as COVID-19 analysis). Constructing two models for different subtasks in a single project requires considerable time and space to design search methods and store indices. Therefore, it is necessary to design a unified model to support different similarity measures.

Existing approaches [1]–[4] typically ask users to specify a threshold and return results based on a specific similarity function. However, if users have no idea how similar the records in the dataset are to the query, the given threshold may not facilitate efficient search. Moreover, different similarity functions, such as Jaccard similarity, cosine similarity, Dice similarity and edit distance, have different limitations. For instance, edit distance is a character-based similarity that takes the string as a sorted character set. It can easily find similar strings such as ISBN code. However, when a user uses several keywords when querying a search engine, because the strings contain the same keywords but are sorted in different orders, token-based similarity functions such as Jaccard similarity are more suitable. Therefore, considering these situations, we study top-k textual similarity search in this article, which utilizes a dynamic threshold based on the number of k to find the k most similar records for the given query.

It is worth noting that the number of records has an important effect on search performance. To address this issue, on the one hand, we need to devise an appropriate index structure to store the records that is easy to search. On the other hand, it is necessary to provide efficient pruning techniques to filter records during search. Therefore, we present a tree-based indexing approach that clusters records based on their textual similarity, and similar records are then stored in the same branch. Based on our indexing, we propose a top-k similarity search algorithm to support multiple similarity functions that find the top-k similar results when traversing the tree. With the help of two efficient pruning techniques, our approach is designed to improve search performance.

To summarize, the contribution of our work is as follows:

1) We formalize top-k textual similarity search and devise a tree-based index to support it. The indexing approach clusters similar records in the same branch offline to improve the performance of online search.
2) Based on the tree-based index, we design a textual similarity search method to support multiple similarity functions. To improve search efficiency, we devise an adapted search strategy for our indexing approach. In the search algorithm, we also implement two pruning strategies to tighten the bound of the dynamic threshold to further improve pruning power.
3) We implement our algorithm in several datasets and compare it with the existing algorithm to evaluate its performance. The experimental results show that our algorithm can achieve high performance and good scalability.

The remainder of our paper is organized as follows. We formulate the problem and review existing relevant works in Section II. In Section III, we present the indexing framework. The search algorithm is described in Section IV. The experimental results are shown in Section V. Finally, we conclude our work in Section VI.

II. PRELIMINARIES

In this section, we formalize the definition of top-k textual similarity search and review some relevant existing works.

A. PROBLEM DEFINITION

First, we formally define top-k textual similarity search.

Given a dataset D and a query Q, if we want to find the top-k records from D that are the most textually similar to Q, we need to calculate all similarity values between Q and each record in the dataset. Then, the k records with the highest similarity scores are selected as the results.

Next, we formalize top-k textual similarity search as follows.

Definition 1 (Top-k Textual Similarity Search): Given a dataset D, a query Q, a number of results k and a similarity function, a top-k similarity search attempts to find the k records from D that are the most similar to Q.

TABLE 1. Dataset *D*.

ID	Record	ID	Record
R_1	$ABCD$	R_5	ABD
R_2	ACD	R_6	CDF
R_3	BCG	R_7	BCD
R_4	$BDEF$	R_8	CFG

Example 1: Table 1 is a dataset with 8 records in which each character is a token. Suppose that we use Jaccard similarity to measure similarity. Consider a query "$ACEG$", and we want to find the top 3 results. After calculating the similarity score of the query and each record, we can finally obtain the results r_1, r_2, and r_8.

B. RELATED WORKS

1) TOKEN-BASED SIMILARITY SEARCH

In token-based similarity search, each string is considered a set of tokens. Users find similar results after comparing the common tokens between the token set of query and records in the dataset. The common similarity measures of token-based similarity include Jaccard [5], cosine [6] and Dice [7]. Recently, there have been some studies on token-based string similarity search [4], [8]–[11] Satuluri and Parthasarathy [11] provided a string similarity search method based on locality-sensitive hashing. Li *et al.* [4] developed Flamingo, which utilizes count and heap filters to address this problem and grouped strings by length, and developed different list-merge algorithms for them to improve performance. Specifically, Flamingo contains three algorithms: ScanCount, MergeSkip, and DivideSkip. ScanCount is simply designed based on array and inverted list. MergeSkip improves ScanCount and skips the lists that cannot be results. DivideSkip is the best algorithm in Flamingo. It first splits the token list of strings into two groups. Then, it uses MergeSkip to generate candidates for the short group and checks whether IDs in the candidates appear in the long group to generate results. As there are no additional structures used in DivdeSkip, the space complex is $O(D)$ (where D is the size of the dataset). Zhang *et al.* [8] provided a B^{+}-tree-based approach for top-k similarity search, which reorders tokens in a string by frequency.

2) CHARACTER-BASED SIMILARITY SEARCH

In character-based similarity search, each string is considered a sequence of characters. Users calculate two records' similarity based on the number of their common characters. The most representative character-based similarity functions are edit distance and Hamming distance [12]. In [3], [13]–[19], the researchers studied string similarity search based on edit distance. Specifically, HSTree [20], [21] is designed by partitioning strings into several segments to construct a complete binary tree as an index to organize the data for searching. Lee *et al.* [22] improved HSTree by connecting the inverted lists of the HSTree node and its child nodes to speed the traversal of the index. Wei *et al.* [23] provided a hash-based similarity search approach, which assigns each string a hash lable and identifies dissimilar bit pattens between two hash lables. Deng *et al.* [19] presented a pivotal prefix-based filter algorithm for string similarity search. Zhang *et al.* [24] studied top-k similarity search and proposed a $B^{ed}-$tree, which is a B^{+}-tree-based index to support edit distance by transforming records to integers for pruning. Wang *et al.* [25] studied KNN sequence search. To address this issue, they provided a method, AppGram, that combines a frequency queue and CA algorithm to prune KNN candidates.

Most of the existing works can only perform token-based or character-based similarity search since the indices they construct are based on one kind of similarity function. However, the method we provide in this article can support both types of search. This is because our index tree is constructed by the similarity of strings, not a specific similarity function.

III. TREE-BASED INDEX

To find textual similarity objects from massive datasets, based on a given query q, we need to calculate the similarity score between q and each record in the dataset, which is too inefficient. Thus, we always construct an index structure for the dataset to prune as many unnecessary records as possible. In this article, we construct a tree-based index.

To store each record in a node of the index tree, we calculate the similarity among records. Therefore, records in the same branch are similar. Based on this idea, we formalize the definition of the index tree as follows:

Definition 2 (Index Tree): Given a fanout number f, the dataset is converted into an index tree T based on the similarity among records, where

- Each node stores only one record, and
- Children of a node should share common tokens, and the number of children is less than f.

Specifically, we classify the records based on cluster methods such as k-means and hierarchical theories. When considering a classified record set for constructing a subtree, the record with the highest similarity score with the rest in the cluster is selected to be stored in the root of this subtree, and the others are stored in the branches.

To construct an index tree, suppose that we use k-means theory and that the distance between a node and its child is $1 - Sim$, where Sim is the similarity between two nodes' records. Given a dataset and a fanout f, we first aggregate the dataset into f clusters and select each mean record to build the first-level nodes (the level of the root is 0). The other records are partitioned into each selected node based on clusters. If the number of records classified to a node is less than f, we directly store each of them in the children of this node as leaf nodes. Otherwise, we iteratively repeat the k-means algorithm to select next-level nodes until the number of remaining records is not larger than f. Then, we store them in the leaf nodes and terminate the cluster of this branch. Recursively, we can divide each cluster into smaller clusters and ultimately generate an index tree in which the root of the tree is an empty node and each of the other nodes stores individual records based on textual similarity. That means there is a one-to-one correspondence between a node and a record. For simplicity, in the remainder of the paper, a node is referred to interchangeably with its corresponding record.

In addition to the record, we also store other parameters in a node to speed the search, including the maximum distance $d_{\max}$ between the node and its descendants, the minimum number of tokens in children $n_{\min}$ and the token set ts that contains all the tokens in the node and its descendants. The algorithms utilizing these parameters are described in Selection IV.

As the upper-level nodes in the index tree contain a large number of tokens in their token set, we next discuss how to store the token set in a node. Since the presence of too many tokens in the nodes could increase space complexity, we use hash tables and bloom filters to store these tokens. All the tokens are sorted based on the document frequency df. We store tokens with high df in the hash table and others in the bloom filter. The bloom filter has a probability of false positives, but this will not affect the correctness of our algorithm, and this is discussed in the next section. Utilizing these two storage structures, as the size of hash table S_h and the space of bloom filter S_b are fixed, the size of the token set

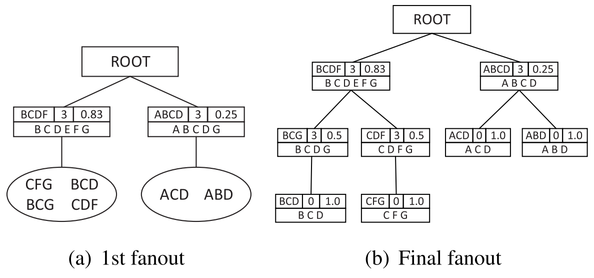


FIGURE 1. **Tree-based index construction.**

is $S_h + S_b$. Suppose that there are N records in the dataset and the average size of records is S_{avg}. Each node corresponds to a distinct record. The space complex of the index tree is $\mathcal{O}((S_{avg} + S_h + S_b) \times N)$.

Example 2: Consider the dataset D in Table 1, where each letter in the records represents a token. Suppose that the fanout f=2, and we use Jaccard similarity to measure the distance. The tree-based index structure can be constructed as shown in Figure 1. We first aggregate the records into two clusters. Utilizing these clusters, we can determine that the first-layer nodes and the other records are stored in their child nodes. In the first step, we classify records into two clusters. "*BCDF*" and "*ABCD*" are the mean records of the two clusters. Therefore, we store them in the first-layer nodes of the index tree. For example, the left child node of ROOT in Figure 1(a) contains the record "*BCDF*". When comparing it with other records in this cluster, its maximum distance is 0.83, and the minimum number of tokens of its following records is 3. This information is shown in the first line of the node. Then, all the tokens contained in these clusters are placed in the second line. Since the structure under this node has not been constructed, we utilize an ellipse containing the remaining records to present the distribution of these records. By repeating this step, we can finally construct the index tree as shown in Figure 1(b).

As shown in the example, the index tree is constructed based on the similarity among records. The greater the detail with which we need to partition the dataset, the more layers are constructed in the tree. This means that the partition of the dataset, which is determined by f, will influence the efficiency of traversing the tree. This is shown in Selection IV.

Updates: For a new record r, we select a branch with the greatest similarity to it by comparing it with the token set of nodes; suppose this branch is rooted by node n. If it is more similar to the token set than node n, we select it as the root, update the parameters on it, and utilize the original record in n for comparison with the children. Otherwise, we compare r with children of node n. We repeat this process until visiting a node that has no more children than the specific fanout. For deletion, if it is not a leaf node, we select the record with the greatest similarity among its descendants to replace the deleted node. For a new vacancy, we repeat this selection until the penultimate level contains specific fanouts.

In this article, all tokens have the same weight when calculating similarity, which means that we compare all tokens in a record with those in another record in determining similarity. Our method can also consider weighted textual similarity for records containing multiple attributes, and the weighted similarity of records is the weighted sum of the similarity scores of all attributes in the records.

IV. TOP-K SIMILARITY SEARCH

In this section, we present a top-k similarity search algorithm utilizing the tree-based index structure provided above. We also design pruning conditions to resolve the challenge of improving the efficiency of traversing the index.

A. SEARCH STRATEGY

After constructing the tree-based index structure, we next need to discuss how to use it to find similarity records.

When traversing the index to search candidate results, we want to visit as few nodes as possible. Therefore, our traversing strategy is designed based on improving the best first traversal (*BFT*) method [26], which always visits the node with the maximum weight from the priority queue.

Given a query Q and a dataset D, first we initialize a threshold τ and priority queue to store the nodes that may have descendants as results. The items in the queue are denoted by $[n, w]$, where n denotes the node in the index tree and its weight is denoted by w. We utilize the similarity between node n and the query to measure the weight. Based on the minimum weight in the priority queue, we use the best first traversal to find the top-k similarity results. As we store records in each internal and leaf node, we can use the original *BFT* method to traverse. For each node n popped from the priority queue, we access this node and compute the similarity between the record on nodes n and Q to determine whether it should be selected into the result set. If it is an answer, we also need to update τ. Then, we calculate the similarity score between its token set and query. If the score is larger than the threshold τ, there may be a result in the descendants of n. Therefore, we push the child of n, which has a distance smaller than $1 - \tau$, into the queue with its weight. Otherwise, we prune the subtree whose root is this child. Obviously, based on the *BFT* method, we can reduce the time for visiting deeper nodes since we always visit the node with the largest weight from the priority queue.

To improve efficiency, we also need some filter techniques to help us further prune unnecessary nodes or branches during traversal, in addition to utilizing the dynamic threshold τ. Thus, we provide some pruning strategies in the next subsection.

B. PRUNING STRATEGIES

The fewer branches we access, the faster we finish traversing the index tree. Thus, it is very important to design efficient pruning techniques to speed traversal. In this subsection, we provide two pruning strategies for our search method.

1) TRIANGLE INEQUALITY

If a node n has a descendant containing a record that may be a result, it should satisfy

$$1 - \tau \geq \mathrm{Dis}(Q, n) + n.d_{\max} \tag{1}$$

where τ is the threshold and $\mathrm{Dis}(*, *)$ denotes the distance between two records based on their similarity.

This inequality is designed based on the trilateral relationship in a triangle where the sum of the distance between the record r on node n and the query. Moreover, the distance between r and records on the descendants of n should be larger than the shortest length $1 - \tau$. As the distance between query Q and node n has already been calculated when node n is pushed into priority queue, to satisfy this triangle inequality, we need to calculate the distance between n and its descendants. The max distance ($d_{\max}$) has already been stored in the node n. Therefore, we only need to estimate whether the inequality has been satisfied.

2) TOKEN BOUND

In addition to utilizing distance among nodes for pruning, we also design a pruning strategy based on token number. The pruning strategy of the token bound utilizes the threshold to deduce the extremum number of common tokens shared between the query and the record that may be a result. Since we want to use it for pruning descendants of a node, we calculate the similarity between the query and the token set of the node to obtain the bound of the token number.

Suppose that we use Jaccard similarity to measure the similarity between two records. For records A and B, their similarity score is

$$J(A, B) = \frac{X}{|A| + |B| - X} \tag{2}$$

where $|A|$ and $|B|$ denote the number of tokens in A and B, respectively. X denotes $|A \bigcap B|$, the number of common tokens shared between A and B.

Therefore, for query Q and threshold τ, if node t is a result, it needs to satisfy

$$\frac{X}{|Q| + |t| - X} \geq \tau \tag{3}$$

Since the token number of the query is confirmed, for the subtree rooted by n, we can obtain the lower bound of token numbers T_{lb} based on $n_{\min}$ to verify whether there may be a result in the descendants of node n, that is,

$$T_{lb} = \left\lfloor \frac{\tau}{1+\tau} \times (|Q| + n.n_{\min}) \right\rfloor + 1 \tag{4}$$

Other textual similarity measures have different bounds of token numbers. We present some token bounds of other similarity measures in Section IV-D.

To utilize this token pruning condition, we need to count the number of common tokens between the query and the token set of a node. For a node n, if this number is not smaller than T_{lb}, we stop counting and treat node n as a candidate to push it into the queue. Otherwise, we prune the subtree rooted by n.

In our index tree, we utilize a bloom filter to store tokens on the nodes. When we verify whether a token is in the bloom filter and find that it is not, this token must not exist in the set. If it is, the token may not be in the set. Therefore, utilizing a bloom filter may not prune all the subtrees to dissatisfy the token bound because of its probability of false positives, and it does not prune the correct results. This means that the bloom filter only impacts the efficiency of pruning but not the correctness.

```
Algorithm 1 Top-k Search ($Q$, $D$)
Input: $D$: Dataset, $Q$: Query
Output: $R$: Results
1  begin
2      Initialize threshold $\tau \leftarrow 0$;
3      Initialize a priority queue $PQ = \varnothing$;
4      Construct an index tree $T$;
5      $PQ \leftarrow [T.root, 0]$
6      while $PQ$ is not empty do
7          $n \leftarrow PQ.dequeue()$;
8          if $PRUNE(Q, n, \tau) = true$ then
9              Prune $n$;
10             Continue;
11         if $R.size() < k$ or $Sim(n.node, Q) > \tau$ then
12             $R.update(n)$;
13             Update $\tau$;
14         if $n$ is not a leaf then
15             for child $c \in n$ do
16                 if $PRUNE(Q, c, \tau) = false$ then
17                     $PQ.enqueue([c, Dis(c, query)])$;
```

```
Function PRUNE($Q, n, \tau$)
Input: $Q$: Query, $n$: Node, $\tau$: Threshold
1  begin
2      if $1 - \tau \leq \mathrm{Dis}(Q, n) + n.d_{max}$ then
3          return false;
4      else
5          Calculate the token bound $TB$;
6          $nts \leftarrow$ the number of common tokens
               between Q and n.$ts$;
7          if $nts \geq TB$ then
8              return true ;
9          else
10             return false;
```

FIGURE 2. Top-k Search Algorithm.

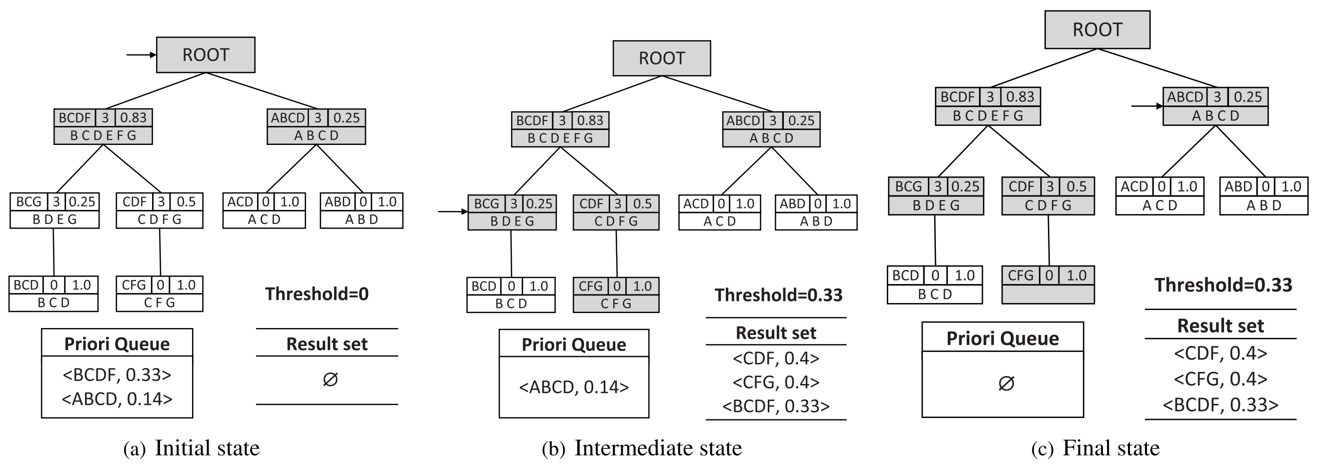


FIGURE 3. An example of search process.

C. TOP-K SEARCH ALGORITHM

When combining the pruning strategies into the *BTF* searching method, our top-k textual similarity search algorithm based on the index tree is shown in Algorithm 1.

We first initialize the threshold and priority queue, then we construct an index tree for the dataset (line 2 - line 4). After constructing the index, we push [*root*, 0] into the priority queue and begin traversal (line 5). For each item popped from the queue, we use triangle inequality and token bound pruning strategies to estimate whether this node and its descendants may contain a result (line 8 – line 10). If not, we prune it. Otherwise, we calculate the similarity between Q and the record in this node to verify whether it could be a result and update τ (line 11 – line 13). As we discuss above, the challenge of improving search efficiency is how to decrease the number of accessed nodes. Therefore, in addition to pruning the item popped from the queue, we also utilize the pruning strategies before we push it. Therefore, we only add the children of an internal node that can survive pruning strategies into the priority queue (line 14 – line 17).

Time Complexity: As our aim is to find the top-k results, we only access at most k leaves with the help of the priority queue. Therefore, the time complexity of our algorithm is based on the number of accessed nodes. In the worst case, each leaf containing a result is in an individual path from the root to itself. Supporting the average length of these paths is L. We need $O(kL)$ searches.

Example 3: Consider the index tree in Figure 1(b). Suppose that the query is "*AGEF*" and we want to find the top 3 similarity records based on Jaccard similarity. The process of traversing the tree for the search query is shown in Figure 3. In these subfigures, the nodes being grey indicates that they have been visited or stored in the priority queue, and the nodes that are pointed represent the node of the current popped item from the queue. For simplicity, we use the record to describe the node in which it is stored. After initialization, we first pop the root. Since it does not contain any record, next we push its children, the nodes on the first level, with their weights into priority queue (Figure 3(a)). Then, we pop the item with node "*BCDF*" to verify whether it is a result and push its children that pass the two pruning conditions into the queue. Iteratively, we can traverse all its descendants. After the third result [*CFG*,0.4] is selected into the result set, we have three results, and the threshold is updated to 0.33. Then, we continue to pop items from the priority queue. That is, [*BCG*,0.17] (Figure 3(b)). After computation, the triangle inequality ($1 - 0.33 \geq 0.83 + 0.25$) is false. Therefore, this node and its descendants are pruned. Then, we continue traversing the tree based on the priority queue and finally pop the last item [*ABCD*, 0.14] from the queue (Figure 3(c)). We find that it also needs to be pruned because there is only one token "*A*" shared by the query's token and this node's token set, which is less than the lower bound lb$=\lfloor 0.33/1.33 * (4+3) \rfloor + 1 = 2$. Therefore, the top 3 similarity records of "*AGEF*" are "*CDF*", "*CFG*", and "*BDEF*".

D. DIFFERENT TEXTUAL SIMILARITY MEASURES

In Subsection IV-B, the textual similarity measure used for our index structure and pruning strategies is Jaccard similarity. Our method can also support other token-based textual similarity functions, including cosine similarity and Dice similarity. Our method can be extended to support a character-based similarity function if it can be transformed into token-based equivalent. To support these similarity measures with our tree-based indexing, we simply replace the similarity measure used for classification, and this will not be given a specific description in each case. Similar to the index structure, the triangle inequality with other similarity functions will also not be described. In this subsection, we propose a token pruning strategy based on several common textual similarity measures.

Given a query Q and a subtree rooted by node t, suppose there is a node n on this subtree and the threshold is τ.

We attempt to determine whether n should be pruned by the token-bound pruning strategy based on the threshold. Therefore, the pruning condition for the token number based on different textual similarity functions is shown as follows:

1) COSINE SIMILARITY

For cosine similarity $COS(r, s) = \frac{|r \cap |s|}{\sqrt{|r \times s|}}$. Suppose that there are X common tokens shared between Q and the string on node t. If there is a result on the subtree rooted by t, we need to satisfy

$$\frac{X}{\sqrt{|Q| \times |t|}} \geq \tau \tag{5}$$

Based on Equation 5, we can generate the minimum token number to make n contain a result on its descendant, that is,

$$\mathrm{T}_{lb} = \left\lfloor \tau \times \sqrt{|Q| \times n.n_{\mathsf{min}}} \right\rfloor + 1 \tag{6}$$

2) DICE SIMILARITY

Similar to Jaccard similarity, given two strings r and s, their Dice similarity score is $DICE(r, s) = \frac{2 \times |r \cap s|}{|r|+|s|}$. We use X to denote the number of common tokens shared by Q and the record on t. Thus, to make node n pass the pruning condition, the similarity score should be larger than τ. Then, we have

$$\frac{2 \times X}{|Q| + |t|} \geq \tau \tag{7}$$

Therefore, the lower bound of the common token number of node n is

$$\mathrm{T}_{lb} = \left\lfloor \frac{\tau}{2} \times (|Q| \times n.n_{\mathsf{min}}) \right\rfloor + 1 \tag{8}$$

3) EDIT DISTANCE

Since the edit distance between two strings is the minimum number of token edit operations (i.e., to calculate the edit distance of strings, we need to consider the maximum number of tokens on which there are no edit operations between two strings. For example, consider two strings " This book is good'' and "This is a good book''; there are 3 common tokens between them, which are "This'', "is'' and "good''. Therefore, the edit distance of two strings s and r is $ED(s, r) = \max(|r| - n_{c_{rs}}, |s| - n_{c_{rs}})$, where $n_{c_{rs}}$ is the maximum number of tokens without an edit operation.

If there is an answer contained in node t under n, it must satisfy

$$\max(|Q| - n_c, |t| - n_c) < \tau \tag{9}$$

where n_c denotes the maximum common token numbers of Q and t.

As we can see, this inequality depends on the size of |Q| and |t|. Thus, to implement this token pruning strategy, we also need to store n_{max}, the token number of the longest string. Based on these two parameters, we can generate the upper bound for pruning, that is,

$$\mathrm{T}_{ub} = \max(M_{min} - \tau, M_{max} - \tau) \tag{10}$$

where M_{min} is the largest token number between Q and n_{min} and M_{max} is that of Q and n_{max}.

V. EXPERIMENT

In this section, we first introduce the datasets and experimental settings. Then, we present the experimental results and analysis of comparative performance.

A. EXPERIMENTAL SETUP

1) DATASETS

To evaluate our algorithm, we utilize three datasets:

1) *DBLP* : This is a real dataset that contains more than one million publications, including the title, authors and provenance. We integrated these three attributes of a publication into one record as all the keywords in a record having the same weight.
2) *IMDB* : This dataset contains records of films and TV series from IMDB. Records in this dataset consist of title, producer, year and category.
3) *PubMed* : Each record in this dataset is a title obtained from a medical publication.

All three datasets have different lengths of records. We could utilize them to evaluate the performance of our method for different kinds of data. The details of these three datasets are shown in Table 2.

TABLE 2. Datasets.

Dataset	*DBLP*	*IMDB*	*PubMed*
Record number	1088728	504881	1000001
Avg num of Token	10.73	5.48	20.17
Avg index size(MB)	73.2	45.0	72.9

2) BASELINE METHOD

To prove the efficiency of our algorithm, we select Flamingo [4] as the comparative experiment. As we discuss in Section II-B, Flamingo can support multiple similarity functions since its index structures are not designed based on specific similarity functions. Therefore, we select DevideSkip, the best Flamingo algorithm, as the baseline method in our experiment.

In the experiments, we employ Flamingo [4] as the baseline method. This is because the motivation of this work is to provide a method for multiple similarity measures. Some recent works can very efficiently support the similarity search problem. However, this work constructs an index or provides a search strategy based on only one particular similarity function. If we want them to support other similarity functions, we need to change the basic framework of these works. To the best of our knowledge, only Flamingo can support multiple similarity measures. Therefore, we only select it for comparison.

3) EVALUATION METRICS

All the algorithms were implemented in Java. All experiments were run on a Linux 10.0.4 machine with an Intel Xeon E5420 2.50 GHz CPU and 15 GB memory.

Due to space constraints, we only demonstrate the experimental results utilizing Jaccard similarity and edit distance.

TABLE 3. Top 10 results of "'Object SQL - A Language for the Design and Implementation of Object Databases. Modern Database Systems1995".

	record	score
1	Design and Implementation of Flora, A Language for Object Algebra.1995	0.471
2	On the design and implementation of a geometric-object-oriented language.2007	0.445
3	The design and implementation of a dataflow language for scriptable debugging.2007	0.445
4	Design and Implementation of a Deductive Query Language for ODMG Compliant Object Databases.EDBT PhD Workshop 2000	0.429
5	Design and Implementation of a Visual Query Language for Large Spatial Databases.IV 2002	0.421
6	he Object-Oriented Design and Implementation of a Relational Database Management System. 1995	0.421
7	The Design and Implementation of a Legal Text Database. DEXA 1994	0.412
8	The Design and Implementation of a Sequence Database System. VLDB 1996	0.412
9	The Design and Implementation of a Domain-Specific Language for Network Performance Testing. 2007	0.4
10	Structured Hypermedia Authoring: A Simple Tool for the Design and Implementation of Structured Hypermedia Databases. CALISCE 1996	0.4

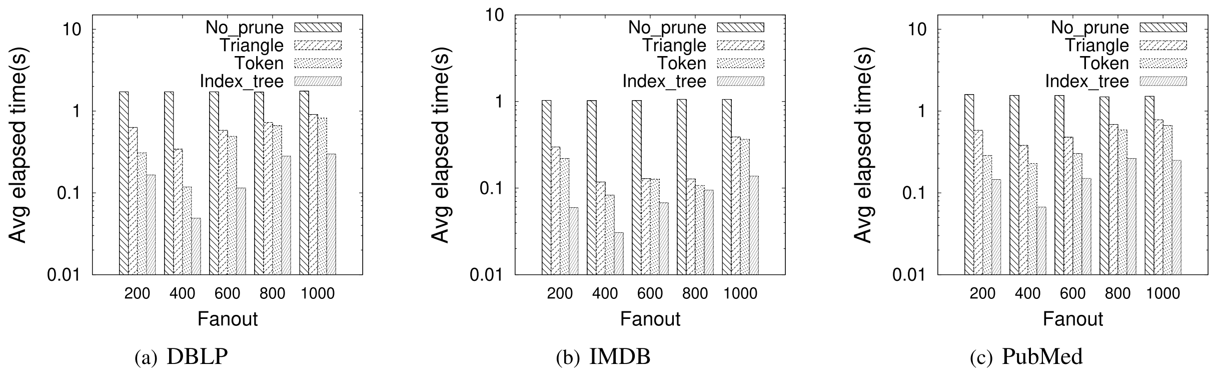


FIGURE 4. Evaluating fanout in tree-based algorithm.

The algorithms based on other similarity functions yield similar results.

B. EVALUATING QUALITY

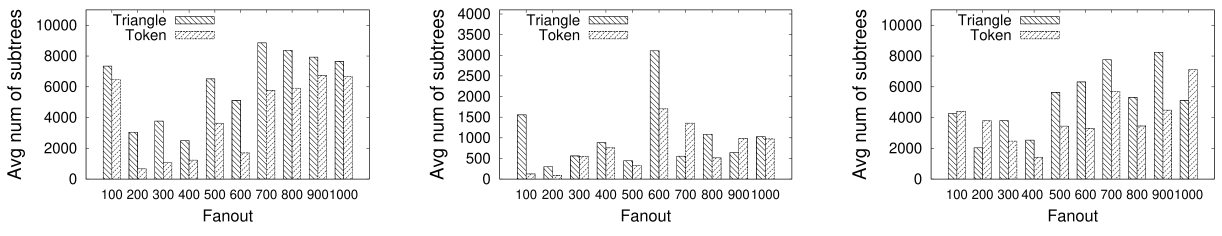
In this subsection, we evaluate the quality of our algorithm. We randomly select a record "Object SQL - A Language for the Design and Implementation of Object Databases. Modern Database Systems 1995" as a query and use our tree-based similarity searching algorithm to find the top 10 results based on Jaccard similarity. The result is shown in Table 3. We also evaluate the quality of Flamingo, and its results are the same as ours. The experimental results show that our method can effectively find similarity results from the database to answer the query.

C. EVALUATING FANOUT

In this subsection, we evaluate the influence of fanout on top-6 similarity searching by varying the number of fanouts in the index tree. We evaluate the influence of fanout on four types of search methods: no pruning strategies, only triangle inequality pruning, only token-bound pruning and using both pruning strategies. The results are shown in Figure 4. As we can see, the method *Index_tree* achieved the highest performance and used both pruning strategies. The algorithms using only one strategy are better than those without pruning strategies. *No_prune* has nearly no effect on the elapsed time because this algorithm only uses a priority queue to verify whether a record would be a result, which makes the method always need to visit leaves for verification. In addition, the change rule of elapsed time was parabolic in each dataset, which means that from the smallest fanout, the time decreased as fanout was added and after the peak was reached, the increase in fanout increased the time. This is because the number of fanouts directly influences the number of levels and leaves in a tree. The greater the number of fanouts is, the fewer layers and the more leaves are generated in the tree. For example, in Figure 4(a), for 200 fanouts, the index tree had 6 levels, and the elapsed times of *No_prune*, *Triangle*, *Token* and *Index_tree* were 1.72 s, 0.63 s, 0.31 s and 0.16 s, respectively. Algorithms with 400 fanouts traverse the tree with 4 levels and take 1.71 s, 0.34 s, 0.11 s, and 0.04 s, respectively. Therefore, to make our algorithm achieve high performance, we need to select an appropriate fanout to balance the layer and number of leaf nodes.

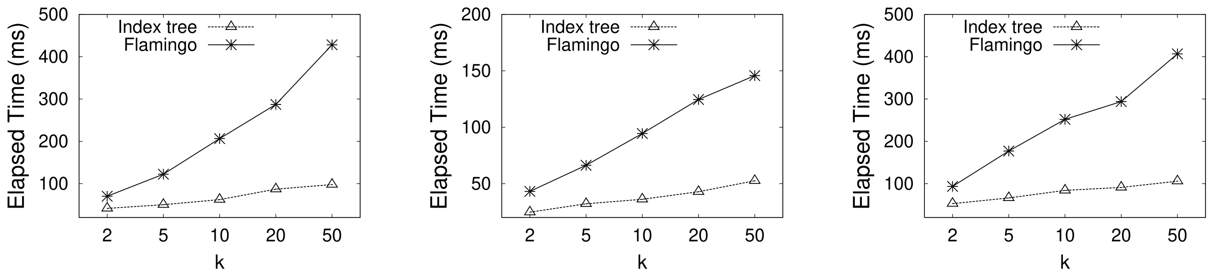
D. EVALUATING PRUNING TECHNIQUES

We evaluate the performance of pruning strategies in this subsection by considering different fanouts. Figure 5 shows the experimental results of the top-5 similarity search. Note that the average number of subtrees pruned by the token pruning strategy was calculated as the subtrees pruned after inequality pruning. Triangle inequality and token pruning clearly achieved efficient performance. This is because while both of



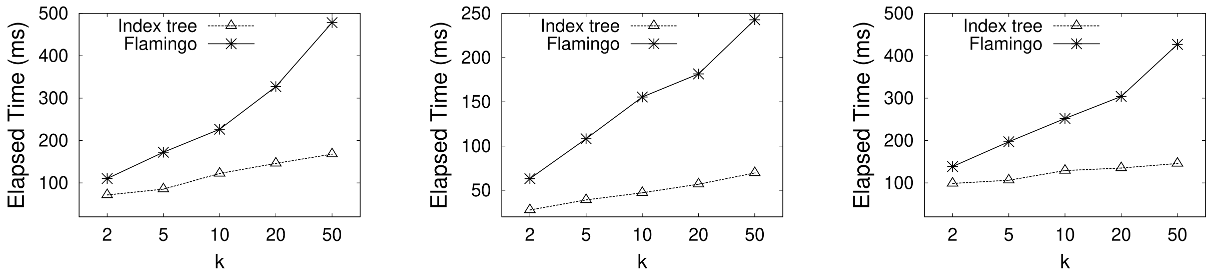
(a) DBLP (b) IMDB (c) PubMed

FIGURE 5. Evaluating pruning conditions in search.



(a) DBLP (b) IMDB (c) PubMed

FIGURE 6. Comparison with the existing approach for Jaccard similarity.



(a) DBLP (b) IMDB (c) PubMed

FIGURE 7. Comparison with the existing approach for edit distance.

them are threshold-based pruning techniques, the inequality utilizes the similarity score while token pruning uses the token set on the node. Moreover, as our index structure is constructed based on clusters with similarity scores, it is not a balance tree. Therefore, the number of pruned subtrees depends on the location of the results.

E. COMPARISON WITH EXISTING WORK

In this subsection, we compare our algorithm with an existing method, Flamingo [4], which also supports multiple textual similarity measures, and we transform it to support the top-k search problem. The time complex of Flamingo is $O(Sl_{avgs} + n_c L)$, where l_{avgs} denotes the average length of the inverted list in the short group, n_c denotes the number of candidates, and S and L denote the number of inverted lists in the short group and long group, respectively.

In the experiments, we evaluated the performance of two algorithms by varying the results number k to support Jaccard similarity and edit distance, which are two representative similarity measures in token-based and character-based similarity measures, respectively. In the evaluations, our algorithm used 400 fanouts, and both pruning techniques were used. The results of comparing our algorithm and Flamingo are shown in Figures 6 and 7.

From these figures, we can see that our algorithm outperformed Flamingo on both similarity measures. This is because although both algorithms utilize the bound of the token number for pruning, our algorithm also uses the triangle inequality as a pruning strategy, and this could not be implemented on Flamingo because its index structure is constructed with an inverted list. The number of candidates generated in Flamingo increases with k since the bound becomes lost. This does not occur in our algorithm, as the power of our pruning techniques increases with k. Therefore, our algorithm is more efficient than Flamingo for top-k searching. For example, in Figure 3(a), it took 122.3 ms to search the top 5 results

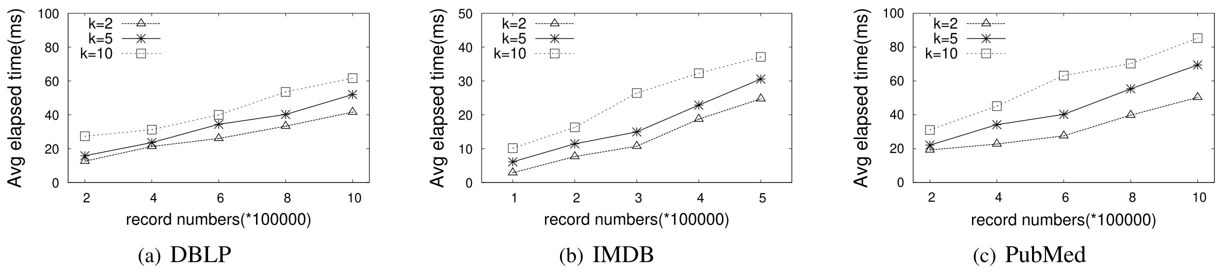


FIGURE 8. Scalability of algorithms.

utilizing Flamingo on DBLP, while our algorithm required only 50.3 ms to finish finding the results.

F. SCALABILITY

We vary the number of records in the dataset to evaluate the scalability of our algorithm. Figure 8 shows the results for different k values. In the experiments, we set fanout=400. The results show that the elapsed time of our algorithm scaled very well for each value of k. The elapsed times grew steadily when increasing the size of the dataset. For example, on PubMed (Figure 8(c)), for k=10, the elapsed time of our tree-based algorithm was 63.2 milliseconds for 600 thousand records. In addition, searching in one million records, it took 85.3 milliseconds. This is because for each number of records, our pruning technique can always effectively prune a large number of unnecessary nodes based on our index structure. With the increase in records, our algorithm can prune more subtrees in the index tree to accelerate search processing.

G. DISCUSSION

In general, we observe that our tree-based similarity search algorithm provides better results than Flamingo in all experiments and all databases. We also obtain the following observations:

1) The index tree we construct for searching can efficiently organize data for searching. The fanout of the index tree influences the power of pruning. The pruning power grows as the fanout increases and falls after a particular number. For the databases we used in the experiments, this number is approximately 400.
2) Our algorithm can address the top-k similarity search problem well on both types of similarity measures. From the experimental results, we can see that the elapsed time increases linearly in the value of k. This is because our pruning strategies are designed based on k. Therefore, the pruning power will grow with k, regardless of what similarity measure is used for searching.

VI. CONCLUSION AND FUTURE WORK

In this article, we studied the top-k textual similarity search problem. We first provide a tree-based indexing approach that constructs indexing based on the textual similarity among records. By utilizing the index tree, we present a search algorithm with two efficient pruning techniques to address the issue. Both techniques can filter the number of nodes to improve pruning. We also introduce how to support multiple similarity measures with our algorithm. The experimental results show that our algorithm is scalable and significantly outperforms the baseline approach. In future research, we will study the top-k similarity join based on the index tree and attempt to present more efficient pruning techniques to improve performance. Moreover, we seek to leverage crowdsourcing techniques [27]–[32] to search entities that are difficult to identify by machine.

REFERENCES

[1] H. Hu, K. Zheng, X. Wang, and A. Zhou, ''GFilter: A general gram filter for string similarity search,'' *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 4, pp. 1005–1018, Apr. 2015.

[2] F. Bi, L. Chang, W. Zhang, and X. Lin, ''Efficient string similarity search: A cross pivotal based approach,'' in *Proc. DASFAA*, Hanoi, Vietnam, 2015, pp. 545–564.

[3] S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, ''Good parts first—A new algorithm for approximate search in Lexica and string databases,'' 2013, *arXiv:1301.0722*. [Online]. Available: https://arxiv.org/abs/1301.0722

[4] C. Li, J. Lu, and Y. Lu, ''Efficient merging and filtering algorithms for approximate string searches,'' in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 257–266.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik, ''A primitive operator for similarity joins in data cleaning,'' in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Apr. 2006, p. 5.

[6] A. Behm, S. Ji, C. Li, and J. Lu, ''Space-constrained gram-based indexing for efficient approximate string search,'' in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 604–615.

[7] R. J. Bayardo, Y. Ma, and R. Srikant, ''Scaling up all pairs similarity search,'' in *Proc. 16th Int. Conf. World Wide Web WWW*, 2007, pp. 131–140.

[8] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan, ''An efficient framework for exact set similarity search using tree structure indexes,'' in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 759–770.

[9] J. Kim and H. Lee, ''Efficient exact similarity searches using multiple token orderings,'' in *Proc. IEEE 28th Int. Conf. Data Eng.*, Apr. 2012, pp. 822–833.

[10] G. Li, J. Feng, and C. Li, ''Supporting search-as-You-type using SQL in databases,'' *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 2, pp. 461–475, Feb. 2013.

[11] V. Satuluri and S. Parthasarathy, ''Bayesian locality sensitive hashing for fast similarity search,'' *Proc. VLDB Endowment*, vol. 5, no. 5, pp. 430–441, Jan. 2012.

[12] X. Zhang, J. Qin, W. Wang, Y. Sun, and J. Lu, ''HmSearch: An efficient Hamming distance query processing algorithm,'' in *Proc. 25th Int. Conf. Sci. Stat. Database Manage. SSDBM*, 2013, p. 19.

[13] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: An adaptive framework for similarity join and search," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2012, pp. 85–96.
[14] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Simlarity search for Web services," in *Proc. VLDB*, Toronto, ON, Canada, 2004, pp. 372–383.
[15] M. Hadjieleftheriou, N. Koudas, and D. Srivastava, "Incremental maintenance of length normalized indexes for approximate string matching," in *Proc. 35th SIGMOD Int. Conf. Manage. Data SIGMOD*, 2009, pp. 429–440.
[16] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast indexes and algorithms for set similarity selection queries," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 267–276.
[17] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *Proc. VLDB*, Vienna, Austria: University of Vienna, 2007, pp. 303–314.
[18] M. Kim, K. Whang, J. L. Lee, and M. Lee, "n-Gram/2L: A space and time efficient two-level n-gram inverted index structure," in *Proc. VLDB*, Trondheim, Norway, 2005, pp. 325–336.
[19] D. Deng, G. Li, and J. Feng, "A pivotal prefix based filtering algorithm for string similarity search," in *Proc. ACM SIGMOD Int. Conf. Manage. Data SIGMOD*, 2014, pp. 673–684.
[20] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng, "Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 519–530.
[21] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, and J. Feng, "A unified framework for string similarity search with edit-distance constraint," *VLDB J.*, vol. 26, no. 2, pp. 249–274, Apr. 2017.
[22] T. Lee, T.-S. Chung, and J. Kim, "Optimized signature selection for efficient string similarity search," *IEEE Access*, vol. 8, pp. 98193–98204, 2020.
[23] H. Wei, J. X. Yu, and C. Lu, "String similarity search: A hash-based approach," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 1, pp. 170–184, Jan. 2018.
[24] 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 *Proc. Int. Conf. Manage. Data SIGMOD*, 2010, pp. 915–926.
[25] X. Wang, X. Ding, A. K. H. Tung, and Z. Zhang, "Efficient and effective KNN sequence search with approximate n-grams," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 1–12, Sep. 2013.
[26] R. Zhong, G. Li, K.-L. Tan, and L. Zhou, "G-tree: An efficient index for KNN search on road networks," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage. CIKM*, 2013, pp. 39–48.
[27] C. Chai, J. Fan, G. Li, J. Wang, and Y. Zheng, "Crowdsourcing database systems: Overview and challenges," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Apr. 2019, pp. 2052–2055.
[28] G. Li, C. Chai, J. Fan, X. Weng, J. Li, Y. Zheng, Y. Li, X. Yu, X. Zhang, and H. Yuan, "CDB: A crowd-powered database system," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 1926–1929, Aug. 2018.
[29] C. Chai, G. Li, J. Li, D. Deng, and J. Feng, "A partial-order-based framework for cost-effective crowdsourced entity resolution," *VLDB J.*, vol. 27, no. 6, pp. 745–770, Dec. 2018.
[30] C. Chai, G. Li, J. Li, D. Deng, and J. Feng, "Cost-effective crowdsourced entity resolution: A partial-order approach," in *Proc. Int. Conf. Manage. Data SIGMOD*, 2016, pp. 969–984.
[31] G. Li, H. Yuan, C. Chai, J. Fan, X. Weng, J. Li, Y. Zheng, Y. Li, X. Yu, and X. Zhang, "CDB: Optimizing queries with crowd-based selections and joins," in *Proc. ACM Int. Conf. Manage. Data SIGMOD*, 2017, pp. 1463–1478.
[32] C. Chai, J. Fan, and G. Li, "Incentive-based entity collection using crowdsourcing," in *Proc. IEEE 34th Int. Conf. Data Eng. (ICDE)*, Apr. 2018, pp. 341–352.

MINGHE YU was born in Shenyang, Liaoning, China, in 1989. She received the B.S. degree in computer science and technology from Northeastern University, Shenyang, in 2012, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2018.

Since 2018, she has been a Lecturer with the Software College, Northeastern University. Her research interests include big data, information retrieval, and data mining.

CHENGLIANG CHAI was born in Harbin, Heilongjiang, China, in 1992. He received the bachelor's degree in computer science and technology from the Harbin Institute of Technology, Harbin, in 2015. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Tsinghua University, Beijing, China.

His research interests include crowdsourcing, data management, and data mining.

GE YU (Senior Member, IEEE) was born in Dalian, Liaoning, China, in 1962. He received the B.E. and M.E. degrees in computer science from Northeastern University, Shenyang, China, in 1982 and 1986, respectively, and the Ph.D. degree in computer science from Kyushu University, Fukuoka, Japan, in 1996.

He has been a Professor with Northeastern University, since 1996. His research interests include distributed data databases, distributed and parallel computing, and blockchain techniques.

Dr. Yu is a Fellow of CCF and a member of ACM. He served as an Associate Editor for the IEEE Transactions on Knowledge and Data Engineering, the *Chinese Journal of Computers*, the *Journal of Software*, and the *Journal of Computer Research and Development*.