A Transformation-based Framework for KNN Set Similarity Search

Yong Zhang *Member, IEEE*, Jiacheng Wu, Jin Wang, Chunxiao Xing *Member, IEEE*

Abstract—Set similarity search is a fundamental operation in a variety of applications. While many previous studies focus on threshold based set similarity search and join, few efforts have been paid for KNN set similarity search. In this paper, we propose a transformation based framework to solve the problem of KNN set similarity search, which given a collection of set records and a query set, returns k results with the largest similarity to the query. We devise an effective transformation mechanism to transform sets with various lengths to fixed length vectors which can map similar sets closer to each other. Then we index such vectors with a tiny tree structure. Next we propose efficient search algorithms and pruning strategies to perform exact KNN set similarity search. We also design an estimation technique by leveraging the data distribution to support approximate KNN search, which can speed up the search while retaining high recall. Experimental results on real world datasets show that our framework significantly outperforms state-of-the-art methods in both memory and disk based settings.

Index Terms—Similarity Search, KNN, Jaccard, Indexing

1 INTRODUCTION

Set similarity search is a fundamental operation in a variety of applications, such as data cleaning [7], data integration [20], web search [3], near duplicate detection [26] and bioinformatics etc. There is a long stream of research on the problem of set similarity search. Given a collection of set records, a query and a similarity function, the algorithm will return all the set records that are similarity with the query. There are many metrics to measure the similarity between two sets, such as OVERLAP, JACCARD, COSINE and DICE. In this paper we use the widely applied JACCARD to quantify the similarity between two sets, but our proposed techniques can be easily extended to other set-based similarity functions. Previous approaches require users to specify a threshold of similarity. However, in many scenarios it is rather difficult to specify such a threshold. For example, if a user types in the keywords "New York, restaurant, steak" in a search engine, he or she may intend to find a restaurant that serves steak. Usually, the users will pay more attention for the results which rank in the front, say the top five ones. In this case, if we use threshold-based search instead of KNN similarity search, it is difficult to find the results that are more attractive for users.

In this paper, we study the problem of KNN set similarity search, which given a collection of set records, a query and a number k, returns the top-k results with the largest JACCARD similarity to the query. We will use "KNN search" for short in the paper without ambiguity. As is known to all, the problem of similarity search is Orthogonal Vectors Problem hard to solve [2]. So it is necessary to devise efficient algorithms to improve the performance for practical instances. There are already some existing approaches for threshold based set similarity search and join [3], [7], [14], [24], [26], one straight forward solution is to extend them to support KNN search as following. This can be done by initializing the similarity threshold as 1 and decreasing it by a fixed *step* (say 0.05) every time. For each threshold, we apply existing threshold-based approaches to obtain the similar records. This step is repeated until we obtain k results. However, this simple strategy is rather expensive as we need to execute multiple search operations during enumeration. Besides, as there is infinite number of thresholds, it is difficult to select a proper value of step. A large step will result in more than k results, which include many dissimilar records; while a small step will lead to more search operations and thus bring heavy overhead. There are also some previous studies on the KNN similarity search with edit distance constraint [8], [22], [23], [27] on string data. They adopt filter-and-verify frameworks and propose effective filtering techniques to avoid redundant computing on dissimilar records. As verifying the edit distance between two strings requires $\mathcal{O}(n^2)$ time, they devise complex filters to reduce the number of verifications. However, as the verification time for set similarity metrics is just $\mathcal{O}(n)$, it is not proper to adopt such techniques for edit distance to support our problem due to their heavy filter cost. Similar phenomenon has also been observed in a previous study: in the experimental study of exact set similarity join [16], it reports that the main costs are spent on the filtering phase, while the verifications can be done efficiently. Xiao et al. [25] studied the problem of top-k set similarity join. It is not efficient to extend it to support our problem because its optimizations are made based on the problem setting that the index is constructed in an online manner. For our KNN search problem, we need to build the index ahead of time before the search begins. Based on above discussions, we find it is not efficient to directly extend the previous approaches for threshold-based similarity search and join to support KNN similarity search. Zhang et al. [30] proposed a tree-based framework to support both threshold and KNN set similarity search. It constructs index by mapping set records into numerical values. In this process, there

- *Y. Zhang, J. Wu and C. Xing are with RIIT, TNList, Institute of Internet Industry, Dept. of Computer Science and Technology, Tsinghua University, Beijing, China. Email: {zhangyong05,xingcx}@tsinghua.edu.cn, wu-jc18@mails.tsinghua.edu.cn;*
- *J. Wang is with Computer Science Department, University of California, Los Angeles. Email: jinwang@cs.ucla.edu*

will be a great loss of useful information and lead to poor filter power.

To address above issues, we propose a transformation based framework to efficiently support KNN set similarity search. Motivated by the work of word embedding [17] in the area of natural language processing, we transform all set records with variant lengths to representative vectors with fixed length. By carefully devising the transformation, we can guarantee that the representative vectors of similar records will be close to each other. We first provide a metric to evaluate the quality of transformations. As achieving optimal transformation is NP-Hard, we devise a greedy algorithm to generate high quality transformation with low processing time. Next we use a R-Tree to index all the representative vectors. Due to the properties of R-Tree, our work can efficiently support both memory and disk based settings. Then we propose an efficient KNN search algorithm by leveraging the property of R-Tree to prune dissimilar records in batch. We further propose a dual-transformation based algorithm to capture more information from the original set records so as to achieve better pruning power.

Moreover, as in many situations it is not required to return the exact KNN results, we also propose an approximate KNN search algorithm which is much faster than the exact algorithm and with high recall at the same time. To reach this goal, we devise an iterative estimator to model the data distribution. We evaluate our proposed methods using four widely used datasets, on both memory and disk based settings. Experimental results show that our framework significantly outperforms state-of-the-art methods.

To sum up, the contribution of this paper is as following:

- We propose a transformation based framework to support the problem of KNN set similarity search for practical instances. We devise an effective greedy algorithm to transform set records to fixed length vectors and index them with a R-Tree structure.
- We propose an efficient KNN search algorithm by leveraging the properties of R-Tree. We further devise a dual-representation strategy to enhance the filtering power.
- We also design an approximate KNN search algorithm by leveraging the statistical information to model data distribution. Then we build an iterative estimator to improve the search performance.
- We conduct an extensive set of experiments on several real world datasets. Experimental results show that our framework outperforms state-of-the-art methods on both memory and disk based settings.

The rest of the paper is organized as following. We discuss related work in Section 2. We formalize the problem definition in Section 3. We introduce the transformation mechanism and indexing techniques in Section 4. We propose the exact KNN search algorithm and pruning strategies in Section 5. We introduce the iterative estimation techniques and approximate KNN algorithm in Section 6. We provide experimental results in Section 7. Finally the conclusion is made in Section 8.

2 Related Work

2.1 Set Similarity Queries

Set similarity queries have attracted significant attentions from the database community. Many previous studies adopted the filter-and-verification framework for the problem of set similarity join. A comprehensive experimental survey is made in [16]. Chaudhuri et al. [7] proposed prefix filter to prune dissimilar records without common prefix, which is followed by a series of related works. Bayardo et al. [3] improved prefix filter by adopting a proper global order of all tokens. Xiao et al. [26] devised the positional filter to further reduce false positive matchings. Vernica et al. [20] devised a parallel algorithm for set similarity join based on the idea of prefix filter. Deng et al. [9] proposed a partition based framework to improve filter power. Wang et al. [24] utilized the relations between tokens and achieved state-of-the-art performance in exact set similarity join.

There are also some all-purpose frameworks that can support multiple operations as well as set similarity metrics. Li et al. [14] proposed Flamingo, an all-purposed framework for similarity search and join under various similarity metrics. Behm et al. [5] extended it to disk-based settings. Zhang et al. [30] proposed a tree based index structure focusing on exact set similarity search problems.

2.2 KNN Similarity Search

KNN similarity search is an important operation which is widely used in different areas, such as road network [32], graph data [13] and probabilistic database [18]. To the best of our knowledge, no prior study focused on specifically improving the performance of KNN set similarity search except the all purpose frameworks [14] and [30]. Xiao et al. [25] studied the problem of top-k set similarity join, which specified the threshold ahead of time and constructed index in an on-line step. This is different from the KNN similarity search problem, which calls for off-line index construction and the ability to support any threshold.

There are several previous studies about string KNN similarity search with edit distance constraint. Yang et al. [27] utilized signatures with varied length to make a trade-off between filter cost and filter power. Deng et al. [8] devised a trie-based framework to compute the edit distance in batch. Wang et al. [23] designed a novel signature named approximate gram to enhance filter power. Wang et al. [22] proposed a hierarchical index to address both threshold and top-k similarity search. Zhang et al. [31] proposed B^{ed}-Tree, an all purpose index structure for string similarity search based on edit distance. Due to the high cost of verifying edit distance, these methods focused on improving the filter power. However, as the cost of verifying set based similarity metrics is much lower, adopting such filter techniques can lead to heavy filter cost which can even counteract the benefit brought by them.

2.3 Locality Sensitive Hash

Locality Sensitive Hash(LSH) is an effective technique for similarity search in high dimensional spaces [11]. The basic idea is to find a family of hash functions with which two objects with high similarity are very likely to be assigned the same hash signature. MinHash [6] is an approximate technique for JACCARD similarity. Zhai et al. [29] focused on approximate set similarity join for lower thresholds. Sun et al. [19] addressed the problem of c-approximate nearest neighbor similarity search, which is different from our problem. Gao et al. [10] devised a learning based method to improve the effectiveness of LSH. LSH based techniques are orthogonal to our work and can be seamlessly integrated into our proposed framework.

3 Problem Definition

In this paper, we use JACCARD as the metric to evaluate the similarity between two set records. Given two records X and Y, the JACCARD similarity between them is defined as

$\text{JACCARD}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$, where $|X|$ is the size of record X. The range of $\text{JACCARD}(X,Y)$ is $[0,1]$. Here in this work, we assume all the set records are multi-sets, which means that duplicate elements in each set are allowed. Next we give the formal problem definition in Definition 1.

Definition 1 (KNN Set Similarity Search). Given a collection of set records $\mathcal{U}$ and a query Q, the KNN Set Similarity Search returns a subset $\mathcal{R} \subseteq \mathcal{U}$ such that $|\mathcal{R}| = k$ and for $\forall X \in \mathcal{R}$ and $Y \in \mathcal{U} - \mathcal{R}$, we have $\text{JACCARD}(X,Q) \geq \text{JACCARD}(Y,Q)$.

Example 1. Table 1 shows a collection of records. Suppose query $Q = \{x_1, x_3, x_5, x_8, x_{10}, x_{12}, x_{14}, x_{16}, x_{18}, x_{20}\}$, and $k = 2$. The top-2 results are $\{X_5, X_6\}$ because the JACCARD similarity between Q and the two records are 0.750 and 0.692, respectively. And the JACCARD similarity for other records are no larger than 0.643.

TABLE 1
A Sample Dataset of Set Records

ID	Record
X_1	$\{x_1, x_2, x_3, x_5, x_6, x_7, x_9, x_{10}, x_{11}, x_{18}\}$
X_2	$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{12}, x_{13}, x_{14}, x_{19}\}$
X_3	$\{x_1, x_2, x_4, x_5, x_6, x_7, x_8, x_{10}, x_{11}, x_{13}, x_{16}, x_{17}\}$
X_4	$\{x_1, x_3, x_4, x_7, x_8, x_9, x_{11}, x_{13}, x_{14}, x_{17}, x_{20}\}$
X_5	$\{x_1, x_3, x_5, x_8, x_{10}, x_{12}, x_{14}, x_{15}, x_{18}, x_{19}, x_{20}\}$
X_6	$\{x_2, x_3, x_5, x_8, x_9, x_{10}, x_{12}, x_{14}, x_{15}, x_{16}, x_{18}, x_{20}\}$
X_7	$\{x_2, x_4, x_7, x_{10}, x_{11}, x_{13}, x_{14}, x_{16}, x_{17}, x_{19}, x_{20}\}$
X_8	$\{x_4, x_5, x_6, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{14}, x_{19}, x_{20}\}$

4 TRANSFORMATION FRAMEWORK

In this section, we propose a transformation based framework to support KNN set similarity search. We first transform all set records into representative vectors with fixed length which can capture their key characteristics related to set similarity. Then we can deduce an upper bound of set similarity between two records by leveraging the distance between them after transformation. As the length of representative vectors is much smaller than that of original set records, calculating such distance is a rather light-weighted operation. We first introduce the transformation based framework in Section 4.1. We then prove that finding the optimal transformation is NP-Hard and propose an efficient greedy algorithm to generate the transformation in Section 4.2. Finally we introduce how to organize the records into existing R-Tree index in Section 4.3.

4.1 Motivation of Transformation

Existing approaches employed the filter-and-verify framework for set similarity search and join. They generated signatures from original records and organized them into inverted lists. As such signatures can be used to deduce a bound of similarity, they make use of these signatures to filter out dissimilar records. However, scanning the inverted lists can be an expensive operation since there are many redundant information in the inverted lists. For a record with l tokens, it will appear in l inverted lists. And for a given query Q, the filter cost will be dominated by the average length of all records in the collection, which will result in poor scalability. As set similarity metrics are relatively light weighted, the filter cost could even counteract the benefits of filtering out dissimilar records.

To address this problem, we propose a transformation based framework that eliminates redundancy in the index structure. Moreover, its performance is independent from the length of records. The basic idea is that for each record $X \in \mathcal{U}$, we transform it into a representative vector $\omega[X]$ with fixed length m. We guarantee such a transformation can reflect necessary information regarding the similarity. Then we can deduce a bound of set similarity from the distance between representative vectors.

The next problem becomes how to propose an effective transformation. Previous approaches use one token as the unit for filtering and build inverted list for each token. Then there will be $|\Sigma|$ inverted lists, where Σ is the global dictionary of all tokens in the collection. The basic idea is to regard each token as a signature for deciding the similarity. Then a straight forward way is to represent each record as a $|\Sigma|$-dimension vector. However, obviously this is not a good choice due to the large value of $|\Sigma|$. To reduce the size, we can divide all tokens into m groups. And we use $\omega_i[X]$ to denote the i^{th} dimension of record X. And the cardinality of $\omega_i[X]$ is correspondingly the value of the i^{th} dimension of the representative vector.

Formally, we group all $|\Sigma|$ tokens into m groups, $\mathcal{G} = \{G_1, G_2, \cdots, G_m\}$. For a record X we have $\omega_i[X] = \sum_{t \in G_i} \mathbf{1}\{t \in X\}$, which is the number of tokens in X that belong to group i. Then we can define the transformation distance between two records by looking at the representative vectors. We claim that it serves as an upper bound of the JACCARD similarity as is shown in Lemma 1.

Definition 2 (Transformation Distance). Given two set records X, Y and a specified transformation ω, we define the transformation distance $\mathsf{TransDist}(\omega, X, Y)$ between them w.r.t representative vectors, which is:

$$\mathsf{TransDist}(\omega, X, Y) = 1 - \left(\frac{|X| + |Y|}{\sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y])} - 1\right)^{-1} \tag{1}$$

Lemma 1. Given two set records X, Y and a transformation ω, the JACCARD similarity between those two records is no greater than the transformation distance $\mathsf{TransDist}(\omega, X, Y)$.

Proof. See Appendix A. □

4.2 Greedy Group Mechanism

Next we will discuss how to generate an effective transformation ω, i.e., how to transform a set record into an m-dimensional vector. It is obvious that different divisions of groups will lead to different tightness of the bound provided by transformation distance. As is shown in Lemma 1, given two sets X and Y, the smaller value $\sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y])$ is, the closer the upper bound is to the real value of JACCARD similarity. Following this route, to minimize the error of estimation, for all the records $X \in \mathcal{U}$, an optimal transformation mechanism should minimize the following objective:

$$\sum_{\substack{\langle X,Y \rangle \in \mathcal{U}^2 \\ X \neq Y}} \sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y]) \tag{2}$$

Unfortunately, we can show that maximizing the value of Equation 2 is NP-Hard in Theorem 1.

Theorem 1. Finding an optimal transformation mechanism is NP-Hard.

Proof. See Appendix B. □

In order to find an effective transformation, we assign tokens into different groups by considering the *frequency* of each token,

Algorithm 1: Greedy Grouping Mechanism ($\mathcal{U}$, m)

Input: $\mathcal{U}$: The collection of set records, m: The number of groups
Output: $\mathcal{G}$: The groups of tokens

1 **begin**
2 Traverse $\mathcal{U}$, get the global dictionary of tokens Σ sorted by token frequency;
3 Initialize the total frequency of each group as 0;
4 **for** *each token* $t \in \Sigma$ **do**
5 Assign t to group G_{min} with minimum total frequency f_{min};
6 Update G_{min} and f_{min};
7 return $\mathcal{G}$;
8 **end**

which is the total number of its appearance in all records in the dataset. It is obvious that tokens with similar frequency should not be put into the same group. The reason is that for two records X and Y, such tokens will be treated as same ones and the value of $\min(\omega_i[X], \omega_i[Y])$ will not increase. As the sum of all token frequencies is constant, we should make the total frequency of tokens in each group nearly the same to make a larger value of Equation 2.

Based on above observation, we then propose a greedy group mechanism considering the total token frequency f_i of each group G_i. The detailed process is shown in Algorithm 1. We first traverse all the records in $\mathcal{U}$ and obtain the global dictionary of tokens; then we sort all tokens in the descent order of token frequency (line 2). The total frequency of each group is initialized as 0 (line 3). Next for each token in the global dictionary, we assign it to the group with minimum total frequency (line 5). If there is a tie, we will assign it to the group with smaller subscription. After assigning a token, we will update the total frequency of the assigned group and the current group with minimum frequency (line 6). Finally we return all the groups.

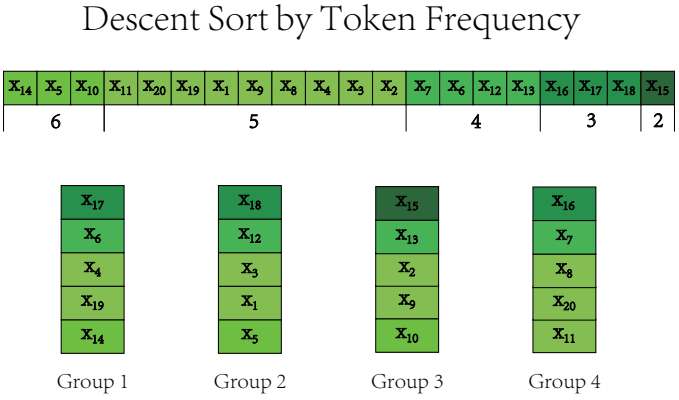
Complexity Next we analyze the complexity of Algorithm 1. We first need to traverse the set record with average length $\bar{l}$ in $\mathcal{U}$, sort Σ, and then traverse each token in Σ. During traversing each token, we need to find the group with the minimum total frequency which costs $\log(m)$ using priority queue. Thus the total time complexity is $\mathcal{O}(|\mathcal{U}| \cdot \bar{l} + |\Sigma| \cdot (\log|\Sigma| + \log m))$.

Example 2. We show the example of Greedy Grouping Mechanism on the data collection in Table 1 with $m = 4$. We first get a global dictionary of tokens and sort all tokens by frequency as is shown on top of Figure 1. The way to group all tokens is according to Algorithm 1. The final result of grouping are shown at the bottom of Figure 1.

Next for a given record X_2, we map its tokens into 4 groups according to above results: tokens x_4, x_6, x_{14}, x_{19} are mapped to group 1; tokens x_1, x_3, x_5, x_{12} are mapped to group 2; tokens x_2, x_9, x_{10} are mapped to group 3; no tokens are in group 4. Then we can get the representative vector of X_2 as $\{4, 4, 3, 0\}$. Actually, the representative vectors of records from Table 1 are shown in Table 2.

4.3 Index Construction

With above techniques, each set record is represented with an m-dimension vector. Then we index them with an R-Tree index. For this application scenario, we do not to worry about the influence of curse-of-dimensionality. The reason is that for our problem,



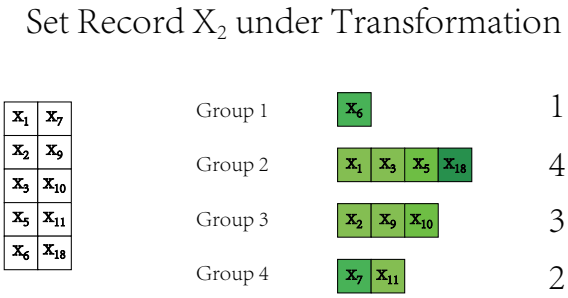


Fig. 1. Greedy Group Mechanism

TABLE 2
Representative Vectors of Set Records

ID	Vector
$\omega[X_1]$	$\{1, 4, 3, 2\}$
$\omega[X_2]$	$\{4, 4, 3, 0\}$
$\omega[X_3]$	$\{4, 2, 3, 4\}$
$\omega[X_4]$	$\{3, 2, 2, 4\}$
$\omega[X_5]$	$\{2, 5, 2, 2\}$
$\omega[X_6]$	$\{1, 4, 4, 3\}$
$\omega[X_7]$	$\{4, 0, 3, 4\}$
$\omega[X_8]$	$\{4, 2, 2, 3\}$

as we treat all the set records as bags of tokens, even if every record is modeled as an d-dimensional data where d is the size of global dictionary of tokens, we only need to look at the non-zero dimensions when calculating Jaccard similarity. Therefore, the data sparsity problem which occurs in many complex data objects (e.g. sequence, image, video) mentioned in the previous paper, will not seriously harm the performance of Jaccard based similarity search.

Next we will first have a brief introduction about the properties of the R-Tree index. The R-Tree is a height balanced index structure for multi-dimensional data [12]. Each node in R-Tree is corresponding to a Minimum Bounding Rectangle (MBR) which denotes the area covered by it. There are two kinds of nodes: an *internal node* consists of entries pointing to child nodes in the next level of tree; while a *leaf node* consists of data entries.

Here we define the size of a node as that of a disk page. Then we can obtain the *fan out* of the index, which is the number of entries that fit in a node. The value of fan out can be set between the maximum and some value that is no larger than the ceiling of maximum divided by 2. The actual fan out of the root is at least 2. Therefore, our framework can naturally support both in-memory and disk settings. And the R-Tree also supports insert and update operations. Similar to B-Tree, such operations eliminate nodes with underflow and reinsert their entries, and also split

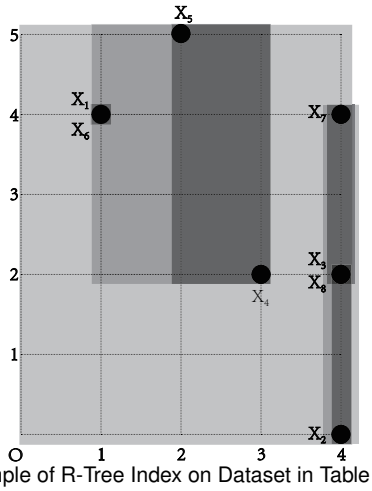


Fig. 2. An Example of R-Tree Index on Dataset in Table 1

overflow nodes. Figure 2 shows an example of R-tree that indexes the vectors in Table 2.

The typical query supported by the R-tree is range query: given a hyper-rectangle, it retrieves all data entries that overlap with the region of this query. But it can also efficiently support KNN search. We will talk about it later in Section 5.

Finally we introduce the process of constructing the index structure. We first generate the global token dictionary and divide them into m groups with Algorithm 1. We then transform each record in the dataset using the generated groups by computing $\omega_i[X]$ with the tokens of X in group i. Next we adopt state-of-the-art method [12] to index all the m-dimensional vectors into the R-Tree structure.

5 Exact KNN Algorithm

In this section, we introduce the exact KNN search algorithm based on the index structure. We first propose a KNN algorithm which can prune dissimilar records in batch by leveraging the property of R-Tree index in Section 5.1. We then further improve the filter power by extending the current transformation in Section 5.2 and devise an optimized search algorithm in Section 5.3.

5.1 KNN Search Algorithm with Upper Bounding

The basic idea of performing KNN search on a collection of set records is as following: we maintain a priority queue $\mathcal{R}$ to keep the current k promising results. Let $\mathsf{UB}^{\mathcal{R}}$ denote the largest JACCARD distance between the records in $\mathcal{R}$ to the query Q. Obviously $\mathsf{UB}^{\mathcal{R}}$ is an upper bound of the JACCARD distance for KNN results to the query. In other words, we can prune an object if its JACCARD *distance* to the query is no smaller than $\mathsf{UB}^{\mathcal{R}}$. Here for $\mathcal{R}$ we maintain $\mathsf{UB}^{\mathcal{R}}$ that is an *upper bound of Jaccard Distance* for KNN results to the query. When searching on the R-Tree index, we use the Transformation Distance to filter out dissimilar records; when performing verification, we use the real JACCARD distance. Every time when JACCARD distance is updated, we will update $\mathsf{UB}^{\mathcal{R}}$ at the same time. With the help of index structure, we can accelerate above process by avoiding a large portion of dissimilar records. Given the query Q, we first transform it into an m-dimension vector $\omega[Q]$. Next we traverse the R-Tree index in a top down manner and find all leaf nodes that might contain candidates with the help of $\mathsf{UB}^{\mathcal{R}}$. We then verify all the records in such nodes to update $\mathcal{R}$ in a similar way.

The next problem becomes how to efficiently locate the leaf nodes containing KNN results. There are two key points towards this goal. Firstly, we should prune dissimilar records in batch by taking advantage of R-Tree index. Secondly, we should avoid visiting dissimilar leaf nodes which will involve many unnecessary verifications.

We have an important observation on the nodes in an R-Tree index regarding the transformation distance. Given a representative vector $\omega[Q]$ and a node N in the R-Tree, we can deduce a minimum transformation distance between $\omega[Q]$ and all records in the subtree rooted by N. This can be realized using the properties of MBR of node N, which covers all the records in the subtree. Then if the minimum transformation distance between $\omega[Q]$ and the MBR of N is larger than $\mathsf{UB}^{\mathcal{R}}$, we can prune the records in the subtree rooted by N in batch. Here we denote the MBR of N as $\mathcal{B}_N = \prod_{j=1}^{|\omega[Q]|}[\mathcal{B}_j^{\perp}, \mathcal{B}_j^{\top}]$, where $\mathcal{B}_j^{\perp}$ and $\mathcal{B}_j^{\top}$ are the maximum and minimum value of the j^{th} dimension, respectively. We formally define the query-node minimum transformation distance in Definition 3.

Definition 3 (Query-Node Minimum Transformation Distance). Given a record Q and a node N, the minimum transformation distance between $\omega[Q]$ and N, denoted as $\mathsf{MinDist}(\omega, Q, N)$, is the distance between the vector and the nearest plane of hyper rectangle of $\mathcal{B}_N$.

$$\mathsf{MinDist}(\omega, Q, N) = 1 - \left(\frac{n(\omega, Q, N)}{d(\omega, Q, N)} - 1\right)^{-1} \quad (3)$$

where

$$n(\omega, Q, N) = \sum_{i=1}^{m} \begin{cases} \omega_i[Q] + \mathcal{B}_i^{\perp} & \omega_i[Q] < \mathcal{B}_i^{\perp} \\ \omega_i[Q] + \omega_i[Q] & \mathcal{B}_i^{\perp} \le \omega_i[Q] < \mathcal{B}_i^{\top} \\ \omega_i[Q] + \mathcal{B}_i^{\top} & \mathcal{B}_i^{\top} \le \omega_i[Q] \end{cases} \quad (4)$$

and

$$d(\omega, Q, N) = \sum_{i=1}^{m} \begin{cases} \omega_i[Q] & \omega_i[Q] < \mathcal{B}_i^{\perp} \\ \omega_i[Q] & \mathcal{B}_i^{\perp} \le \omega_i[Q] < \mathcal{B}_i^{\top} \\ \mathcal{B}_i^{\top} & \mathcal{B}_i^{\top} \le \omega_i[Q] \end{cases} \quad (5)$$

Next we can deduce the lower bound of JACCARD distance, with the help of query-node minimum transformation distance.

Lemma 2. Given a record Q and a node N, $\mathsf{MinDist}(\omega, Q, N)$ is the lower bound of JACCARD distance between Q and any record $X \in N$.

Proof. See Appendix C. □

Example 3. Given a query Q, and the transformation ω defined in the previous section, we have the representative vector $\omega[Q] = \{1, 5, 1, 3\}$. Besides, the MBR of node R_3 in Figure 3 is $[4, 4] \times [0, 4] \times [2, 3] \times [0, 4]$. Therefore, we can compute $n(\omega, Q, N) = 1 + 4 + 5 + 4 + 1 + 2 + 3 + 3 = 23$, $d(\omega, Q, N) = 1 + 4 + 1 + 3 = 9$. Therefore, we will get $\mathsf{MinDist}(\omega, Q, N) = 1 - (23/9 - 1)^{-1} = 0.357$, which is the lower bound of JACCARD distance.

Algorithm 2 shows the process of exact KNN algorithm. First we initialize the result set $\mathcal{R}$ (line 2) and use a queue $\mathcal{Q}$ to buffer the intermediate results (line 3). Then we perform a breadth first search on the R-Tree starting from the root node. Each time we pick the node N on the front of $\mathcal{Q}$ which has the minimum value of $\mathsf{MinDist}(\omega, Q, N)$ with Equation 3. If $\mathsf{MinDist}(\omega, Q, N)$ is larger than $\mathsf{UB}^{\mathcal{R}}$, we can terminate the search algorithm (line 7). Otherwise, if N is a leaf node, we will perform verification on all the records of N and update $\mathcal{R}$, $\mathsf{UB}^{\mathcal{R}}$ accordingly; if N is a non-leaf node, we will iteratively check N's children. For each children N_c of N, if $\mathsf{MinDist}(\omega, Q, N_c)$ is no more than $\mathsf{UB}^{\mathcal{R}}$, we

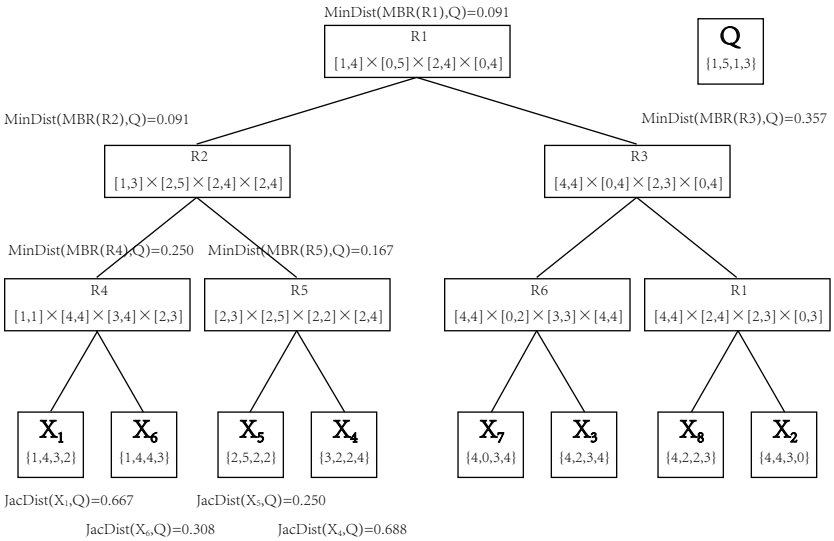


Fig. 3. Illustration of Minimum Transformation Distance between Query and Nodes

will add it into $\mathcal{Q}$ (line 16). Otherwise, we can prune the subtree rooted by it in batch. Finally we return the set $\mathcal{R}$ as KNN results.

Algorithm 2: Exact KNN Algorithm($\mathcal{T}$, Q, ω, k)

Input: $\mathcal{T}$: The R-Tree index, Q: The given query
ω: The transformation,k: The number of results
Output: $\mathcal{R}$: The KNN results

```
1  begin
2      Initialize R, UB^R;
3      Insert the root node of T into Q;
4      while Q is not empty do
5          Dequeue the node N with minimum
           MinDist(ω, Q, N) from Q;
6          if MinDist(ω, Q, N) ≥ UB^R then
7          Break;
8          if N is leaf node then
9          for each record X ∈ N do
10             if JAC(X, Q) ≥ 1 − UB^R then
11             Add X into R;
12             Update UB^R;
13         else
14             for each child N_c of N do
15                 if MinDist(ω, Q, N_c) ≤ UB^R then
16                 Add N_c into Q;
17     return R;
18 end
```

Example 4. The R-tree index for the data collection in Table 2 is shown in Figure 3. First suppose $k = 2$. For the given query Q, its representative vector $\omega[Q] = \{1, 5, 1, 3\}$. Then we start from root node R_1. We calculate the MinDist between $\omega[Q]$ and the MBR for its each children node: 0.091 for R_2 and 0.357 for R_3. Since $\mathsf{MinDist}(\omega, R_2, Q)$ is smaller, we first iterate on this sub-tree. So we need to add R_4 and R_5 into the priority queue, which stores the R-tree nodes to be processed next and calculate the MinDist for them which are 0.250 and 0.167, respectively. Then we reach the leaf node R_5 and calculate the JacDist for records X_5 and X_4. Meanwhile, we need to update the $\mathsf{UB}^{\mathcal{R}} = 0.688$. Next the algorithm will visit R_4, since we find $\mathsf{JacDist}(Q, X_6)$ is lower than $\mathsf{UB}^{\mathcal{R}}$ and therefore, we update $\mathsf{UB}^{\mathcal{R}} = 0.308$. Also, we need to remove the X_4 from $\mathcal{R}$. Then the queue of candidate node only has R_3 left. However, $\mathsf{MinDist}(\omega, Q, R_3) = 0.357$ which is greater than 0.308. Therefore, our KNN search stops and returns the current result. In this process, we prune the right subtree of root node which contains four records.

Finally we show the correctness of above KNN algorithm in Theorem 2.

Theorem 2. The results returned by Algorithm 2 involve no false negative.

Proof. See Appendix D. □

5.2 Multiple Transformations Framework

With a single transformation, we can only reveal a facet of the features in a set record. Therefore, the pruning power could be weaken due to loss of information. To address this problem, we discuss the way of utilizing multiple independent transformations to excavate more information of data distribution.

To reach this goal, the first problem is how to construct index and perform filtering with the help of multiple transformations. The basic idea is that given a set of transformations Ω, for each record $X \in \mathcal{U}$ under transformation $\omega \in \Omega$, we could generate different representative vector $\omega[X]$ individually. We give an order to Ω and assign each transformation in Ω with a number, thus we use ω_i to represent the i^{th} transformation. Then we define the joint representative vector under Ω by concatenating vectors generated by different transformations $\omega_i[X]$ into one vector:

$$\biguplus_{\Omega}[X] = \bigoplus_{i=1}^{|\Omega|} \omega_i[X] \tag{6}$$

where $\bigoplus$ is the operation of concatenation. And we call $\biguplus_{\Omega}$ the multiple transformation operation.

Therefore, with the help of joint representative vector, we could apply multiple transformations pruning techniques.

In the phase of index construction, we first create the transformation set Ω discussed above, and then map each record X into multiple representative vector $\biguplus_{\Omega}[X]$ and index them with an R-tree index.

After constructing the index, we could accelerate the search progress in the same way just as what is mentioned before by utilizing Algorithm 2. The only difference is that we need to

replace the transformation distance and query-node minimum transformation distance with the new distance as is shown in Definition 4.

Definition 4 (Multiple-Transformation Distance). Give two set records $X$and Y, and the multiple transformations $\biguplus_\Omega$, the definition of Multiple-Transformation Distance is:

$$\mathsf{TransDist}(\biguplus_\Omega, X, Y) = \max_{1 \le i \le |\Omega|} \mathsf{TransDist}(\omega_i, X, Y) \quad (7)$$

Based on Lemma 1, we could deduce that the Multiple-Transformation Distance is also an lower bound of JACCARD distance as is demonstrated in Lemma 3.

Lemma 3. Multiple-Transformation Distance serves as the lower bound of the JACCARD distance.

Proof. See Appendix E. □

Example 5. Suppose we have two different pairs:
$Y_1 = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ and
$Y_2 = \{x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}\}$;
$Z_1 = \{x_1, x_3, x_5, x_7, x_9, x_{11}, x_{13}, x_{15}\}$ and
$Z_2 = \{x_2, x_4, x_6, x_8, x_{10}, x_{12}, x_{14}, x_{16}\}$.
Meanwhile, given two different transformation: ω_1 maps x_1, x_2, x_3, x_4 to group 1, x_5, x_6, x_7, x_8 to group 2, $x_9, x_{10}, x_{11}, x_{12}$ to group 3 and $x_{13}, x_{14}, x_{15}, x_{16}$ to group 4; and ω_2 maps x_1, x_3, x_5, x_7 to group 1,$x_9, x_{11}, x_{13}, x_{15}$ to group 2,x_2, x_4, x_6, x_8 to group 3, and $x_{10}, x_{12}, x_{14}, x_{16}$ to group 4. Correspondingly we could get their representative vector under ω_1 and ω_2 individually.
$\omega_1[Y_1] = \{4, 4, 0, 0\}, \omega_1[Y_2] = \{0, 0, 4, 4\}$,
$\omega_1[Z_1] = \{2, 2, 2, 2\}, \omega_1[Z_2] = \{2, 2, 2, 2\}$;
$\omega_2[Y_1] = \{2, 2, 2, 2\}, \omega_2[Y_2] = \{2, 2, 2, 2\}$,
$\omega_2[Z_1] = \{4, 4, 0, 0\}, \omega_2[Z_2] = \{0, 0, 4, 4\}$.
Therefore, we have $\mathsf{TransDist}(\biguplus_\Omega, Y_1, Y_2) = 0$, $\mathsf{TransDist}(\biguplus_\Omega, Z_1, Z_2) = 0$ for both given pairs compared with $\mathsf{TransDist}(\omega_1, Y_1, Y_2) = 0$, $\mathsf{TransDist}(\omega_1, Z_1, Z_2) = 1$ and $\mathsf{TransDist}(\omega_2, Y_1, Y_2) = 1$, $\mathsf{TransDist}(\omega_2, Z_1, Z_2) = 0$, which shows that multiple-transformation distance is closer to JACCARD distance than each single transformation distance.

Similarly, the Multiple-Transformation Distance can be used in the R-Tree index to prune dissimilar records in batch. On the basis of Query-Node Minimum Transformation Distance, we define a similar mechanism under the multiple transformations in Definition 5.

Definition 5 (Query-Node Minimum Multiple-Transformation Distance). Given a multiple representative vector $\biguplus_\Omega[Q]$ and a node N, the Query-Node Minimum Multiple-Transformation Distance between $\biguplus_\Omega[Q]$ and N, denoted as $\mathsf{MinDist}(\biguplus_\Omega, Q, N)$, is the distance between the multiple vectors to the nearest plane of the hyper rectangle of MBR.

We could calculate $\mathsf{MinDist}(\biguplus_\Omega, Q, N)$ as the following ways:

$$\mathsf{MinDist}(\bigoplus_\Omega, Q, N) = \max_{1 \le i \le |\Omega|} (\mathsf{MinDist}(\omega_i, Q, N)) \quad (8)$$

Example 6. Given the representative vector of a query $Q = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$ and the MBR of node N $\prod_{i=1}^{8}[1, 2]$ under multiple-transformation which combines ω_1 and ω_2 mentioned in Example 5, we get $\omega_1[Q] = \{4, 4, 0, 0\}$ and $\omega_2[Q] = \{2, 2, 2, 2\}$. Based on Example 3, we get $\mathsf{MinDist}(\omega_1, Q, N) = 0.6, \mathsf{MinDist}(\omega_2, Q, N) = 0.0$. Therefore, we could get $\mathsf{MinDist}(\biguplus_\Omega, Q, N) = \max\{0.6, 0.0\} = 0.6$.

Since we get the maximum individual transformation distance among all transformations to calculate the Query-Node Minimum Multiple-transformation Distance, it is obvious that this distance has almost the same property but no less than the individual Query-Node Minimum transformation distance. It is easy to see that the Query-Node Minimum Multiple-Transformation Distance can be a tighter bound than that in Definition 3. We prove it in Lemma 4.

Lemma 4. Given a record Q and a node N, $\mathsf{MinDist}(\biguplus_\Omega, Q, N)$ is the tighter lower bound for JACCARD distance between Q and any record $X \in N$ than $\mathsf{MinDist}(\omega_i, Q, N), \forall \omega_i \in \Omega$, individually.

Proof. See Appendix F. □

5.3 Multiple Transformation based KNN Search

Although Multiple-transformation Distance can improve filter power, the overall performance could deteriorate with a large number of transformations. The overhead comes from several aspects: Firstly, we need to store joint representative vector into the index. A larger number of transformations will result in extra space overhead. Secondly, as specified in Equation 8, we need to calculate the Query-Node Minimum Transformation Distance multiple times. Thirdly, it is difficult to construct multiple transformations with high quality. If two transformations are very similar, the lower bound deduced from them will also be very close. In this case, there will not be improvement of filter power.

Based on these considerations, we propose a dual-transformation framework which utilizes only two transformations. Then the problem becomes how to construct them. Regarding the third concern above, it is better to construct two *dissimilar* transformation. Here the definition of "similar" is: for two similar transformations, records mapped into the same group under one transformation are in a great possibility to be mapped into the same group under another. Then we proposed a metric to measure the similarity between two transformations.

Definition 6 (Transformation Similarity). Give two transformations ω_1 and ω_2 which groups all tokens in Σ into two individual groups $\mathcal{G}^1 = \{\mathcal{G}_1^1, \mathcal{G}_2^1 ... \mathcal{G}_m^1\}$ and $\mathcal{G}^2 = \{\mathcal{G}_1^2, \mathcal{G}_2^2, ... \mathcal{G}_m^2\}$, the transformation similarity between ω_1 and ω_2 can be defined as:

$$Q(\omega_1, \omega_2) = \max_{i \le m} \max_{j \le m} \sum_{t \in \mathcal{G}_i^1 \cap \mathcal{G}_j^2} f(t) \quad (9)$$

where $f(t)$ represents the frequency of token t.

The transformation similarity measures how similar two transformations are. The goal is trying to minimize the similarity. However, it is very expensive to get a minimum value of this objective function. The reason is that we need to iterate over all possible different groupings, whose search space is exponential with the number of tokens. One way to solve this problem is to start from the original transformation in Section 4. Following the similarity function in Definition 6, we construct a transformation that is dissimilar with it. In this way, we can obtain a pair of dissimilar transformations efficiently.

Algorithm 3 shows the process of generating dual-transformation. In order to use the original transformation proposed by Algorithm 1, we first utilize the Greedy Grouping

Algorithm 3: Greedy Multiple Grouping Mechanism ($\mathcal{U}, m$)

Input: $\mathcal{U}$: The collection of set records, m: The number of groups
Output: $\mathcal{G}, \mathcal{H}$: Two different output groups of tokens

```
1  begin
2      G = Greedy Grouping Mechanism(U, m);
3      Initialize ℑ as ∅;
4      for each group G_i in G do
5          K ← Greedy Grouping Mechanism(G_i, m);
6          Append K to ℑ;
7      Initialize the total frequency of all groups in H as 0;
8      for each transformation K ∈ ℑ do
9          for each K_i ∈ K do
10             Find group H_min with minimum total
               frequency f_min with index h;
11             while ∃k, K_k ⊆ H_min ∧ K_k ∈ K do
12                 Find group H_min with the next minimum
                   total frequency;
13             Assign all tokens in K_i to H_min;
14             Update H_min and f_min;
15     return G, H;
16 end
```

Mechanism to get our first transformation $\mathcal{G}$ (line 2). Then we want to create another transformation dissimilar with $\mathcal{G}$ from it by traversing all groups in $\mathcal{G}$ to construct groups in the new transformation. For each group $G_i \in \mathcal{G}$ which also could be seen as token set, we create transformation for it with the Greedy Grouping Mechanism and collect all transformations into $\Im$(line 6). The total frequency of each group in the new transformation $\mathcal{H}$ is initialized as 0 (line 7). Next we traverse each transformation $\mathcal{K}$ in $\Im$, and each group K_i in the current transformation $\mathcal{K}$, we assign the tokens in K_i to the group H_{min} with minimum total frequency (line 13) only if the output group H_{min} has no other tokens containing in group K_k from same transformations $\mathcal{K}$ with K_i. This setting guarantees that the output group should not contain different parts of tokens from the corresponding group in $\mathcal{G}$. Otherwise, we need to select another group with minimum total frequency except for H_{min} until the requirement is satisfied. After assigning a token, we will update the total frequency of the assigned group and the current group with minimum frequency (line 14). Finally we return the output groups $\mathcal{G}, \mathcal{H}$.

Actually, we would get two dissimilar transformations from Algorithm 3 (A running example is shown in Appendix H). Moreover, based on Lemma 4, the filtering power of our dual-transformation is stronger than that of single transformation.

Complexity The overall running time of Algorithm 3 consists three parts:

- Create the first transformation $\mathcal{G}$: the total time is $\mathcal{O}(|\mathcal{U}| \cdot \bar{l} + |\Sigma| \cdot (\log|\Sigma| + \log m))$.
- Split each group with around $\frac{|\Sigma|}{m}$ tokens in $\mathcal{G}$: it needs to call the function Greedy Grouping Mechanism m times. The total time is $\mathcal{O}(|\Sigma| + |\Sigma| \cdot \log m)$.
- Traverse $\mathcal{K}$ and assign tokens with minimum total frequency to $\mathcal{H}_{min}$. The total time is $O(m^2 \log m + |\Sigma|)$.

6 Approximate Search Algorithm

In this section, we propose an approximate KNN algorithm which does not return the exact KNN results but runs much faster. We first introduce the idea of distribution aware approximation of KNN results in Section 6.1. We then talk about how to make the partition by leveraging the our R-Tree index in Section 6.2.

6.1 Distribution-aware Approximation

The general idea of approximate algorithm is to estimate the KNN results by considering the distribution of data. Then we can find the KNN results according to the "density" of data without traversing the index. That is when performing KNN search, we fit the given query into area which is the closest to it and find the approximately k nearest neighbors based on the distribution of data. To reach this goal, we partition the space composed by all representative vectors into a collection of p buckets $B = \{b_1, b_2, \cdots, b_p\}$. Here we use U to denote the set of all representative vectors for records in $\mathcal{U}$ and abuse the notation of b_i to represent all the records in the bucket b_i. We utilize the MinSkew [1] principle, which tries to uniformly distribute records across all buckets. Here a bucket is defined as the MBR that encloses all vectors belonging to it. Therefore, buckets may overlap with each other, but one record only belongs to one bucket.

Given a query Q, we can generate the range of Q denoted as R_Q^r which is a circle with Q as the center and r as radius. Then under the assumption of uniform distribution, for each bucket $b_i \in B$ which overlaps with the range of Q, the total number of records in the overlap area $b_i \cap R_Q^r$ can be estimated as proportional to the total number of records in b_i, i.e., $n_i \frac{|b_i \cap R_Q^r|}{|b_i|}$, where n_i is the number of vectors in b_i.

Then the way of approximately getting KNN results is as following: we increase the value of r from 0 incrementally and collect the records within r distance to Q until we have $\epsilon \cdot k$. Here ϵ is a tunable parameter to denote the portion of candidates to collect. We will show its influence and settings later in the experiments. For each value of r, we estimate the total number of records in $\cup b_i \cap R_Q^r$ for all buckets s.t. $b_i \cap R_Q^r \neq \emptyset$ and regard them as candidates. If there are already $\epsilon \cdot k$ records in the candidate set, we will stop here and verify their JACCARD similarity. Finally we will return the top-k records with the largest JACCARD similarity among those collected results.

The next problem becomes how to partition all datasets into p buckets. According to the principle of MinSkew, we should try to minimize the difference between buckets. To this end, we use the perimeter, i.e., total length of planes to evaluate the quality of bucket b_i as is shown in Equation 10:

$$\Upsilon(b_i) = n_i \sum_{i=1}^{m} L_i \tag{10}$$

where a bucket can be represented by its length of planes, i.e., $\langle L_1, L_2, \cdots, L_m \rangle$ along each dimension. Here the meaning of $\Upsilon(b_i)$ is the "uniformity" of all records in b_i. The larger value $\Upsilon(b_i)$ is, the heavier skewness bucket b_i has, and correspondingly the larger error of estimation will be. As we use the intersection of b_i and R_Q^r to estimate the KNN results, it is easy to see that buckets with larger perimeter will lead to more errors in estimation, which is similar to the case of node splitting in R^*-Tree [4]. Thus the problem becomes how to build p buckets for all records U and minimize the total uniformity of all buckets. Under the assumption of uniform distribution [1], n_i can be regarded as a constant. Then we need to partition all records into p buckets so that the total perimeter of all buckets is minimized.

To sum up, the goal of bucket construction is to minimize the value of $\sum \Upsilon(b_i)$. However, we find that for $m > 1$ minimizing such value is NP-Hard:

Theorem 3. For a collection of vectors $U \in R^m$, $m > 1$, $p > 1$. The problem of dividing U into p buckets $B = \{b_1, b_2, \cdots, b_p\}$ with minimum value of $\sum_{i=1}^{p} \Upsilon(b_i)$, s.t. $\forall i, j \in [1, p], i \neq j$, $b_i \cap b_j = \emptyset$, $\mathcal{B}_{b_i} \cap \mathcal{B}_{b_j}$ could be non-empty, is NP-Hard.

Proof. See Appendix G. □

Thus we need to make some heuristic approaches to find such buckets. We will introduce details in the next subsection.

6.2 Iterative Bucket Construction

Next we talk about how to generate the buckets. As shown in Theorem 3, finding the optimal partition of buckets is NP-Hard. One way to make an approximation is to adopt the idea similar to clustering algorithms such as hierarchical clustering. The basic idea is to regard each record as a bucket, i.e., starting with n buckets, and in each step we merge two buckets into one until we obtain p buckets. Here the objective function is: $\sum_{i=1}^{p} \Upsilon(b_i) = \sum_{i=1}^{p} n_i \sum_{j=1}^{m} L_j$.

In each step, we can select the buckets to merge based on the idea of gradient descent: we try to merge each bucket with its neighbors and adopt the selection which can minimize above objective function. This method will run $n - p$ steps. For each step, there will be $\mathcal{O}(n^2)$ trials of merge operation. Thus the time complexity of this strategy is $\mathcal{O}(n^3)$.

We can see that it can be very expensive to apply above method on the data collection since the value of n can be very large. The reason is that we need to construct the buckets from scratch without any prior knowledge. Recall that we have already built the R-Tree index which tries to minimize the overlap between the nodes. Therefore, the MBRs of R-Tree can be a good starting point for constructing the buckets since we need to minimize the total perimeter of the buckets. Based on this observation, we then propose an iterative approach to construct the buckets by leveraging the existing R-Tree index.

Given the R-Tree index $\mathcal{T}$, we first traverse from the root and locate at the first level with M ($M > p$) nodes. Then the problem becomes constructing p buckets from the M nodes instead of n records, where $M << n$. We adopt a similar merging based strategy: the algorithm runs p steps and in each step we construct a bucket. Here let $\mathcal{P}(\cdot)$ denote the half perimeter of a given set of MBRs. Let U' denoted the set of unassigned records; let n' denote the number of records in U'. Then at the step i, the objective function we need to minimize can be calculated as the sum of skewness of bucket b_i and that of remaining records:

$$\Upsilon(b_i) = n_i \cdot \mathcal{P}(b_i) + n' \cdot \left(\frac{\mathcal{A}(U')}{p - i}\right)^{\frac{1}{m}} \cdot m \qquad (11)$$

where $\mathcal{A}(U')$ is the area for MBR of the remaining records covered by nodes which do not belong to any bucket so far.

The process of iterative bucket construction is shown in Algorithm 4 (A running example is shown in Appendix H). We first locate at the first level of $\mathcal{T}$ with more than p nodes (line 2). In step i, we initialize the bucket b_i with the node N_l having left-most MBR (line 4). Next, we try to add remaining nodes one by one from left to right into b_i and look at the value of Equation 11. That is, we calculate the values of $\Upsilon(b_i)$ by adding each remaining node to b_i. For each time, we add the node that leads to the smallest $\Upsilon(b_i)$ value to b_i, and repeat. If no node addition reduces the value of $\Upsilon(b_i)$, the step i finishes and the current bucket becomes b_i in the results (line 8). When there are p–1 buckets constructed, we group all remaining nodes into the last bucket and stop the algorithm (line 9).

Algorithm 4: Iterative Bucket Construction($\mathcal{T}$, p)

Input: $\mathcal{T}$: The R-Tree Index; p: The number of buckets
Output: B: The constructed buckets

```
1  begin
2      Find the first level in T with more than p nodes;
3      for i = 1 to p − 1 do
4          Initilize bucket b_i with node N_l;
5          for node N ∈ U' do
6              Find the bucket b_i leading to smallest Υ(b_i);
7              if The value of Υ(b_i) is not reduced then
8                  Remove N and stop here for b_i;
9      Add the remaining nodes into bucket b_p;
10     return B;
11 end
```

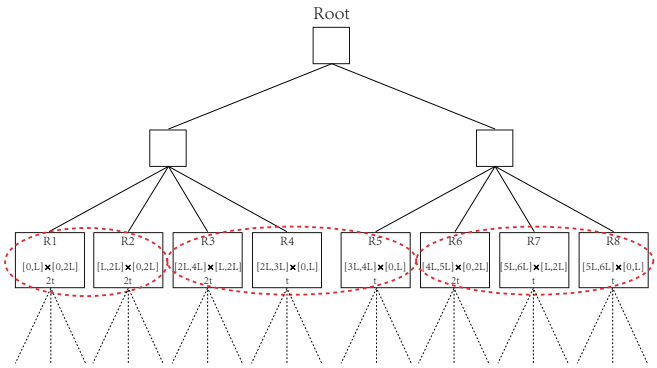


Fig. 4. Iterative Method for Buckets Construction

TABLE 3
Statistics of Datasets

Dataset	Cardinality	Max Len	Min Len	Ave Len
KOSARAK	990,002	2498	1	8.1
LIVEJOURNAL	3,201,203	300	9	35.1
DBLP	4,039,510	245	1	7.1
PUBMED	20,916,083	3383	28	110.2

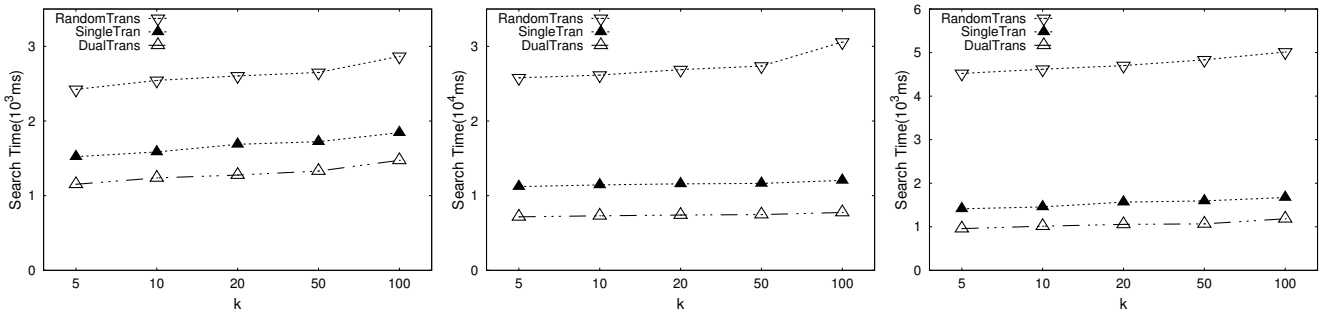
7 EVALUATION

7.1 Experiment Setup

We use four real world datasets to evaluate our proposed techniques:

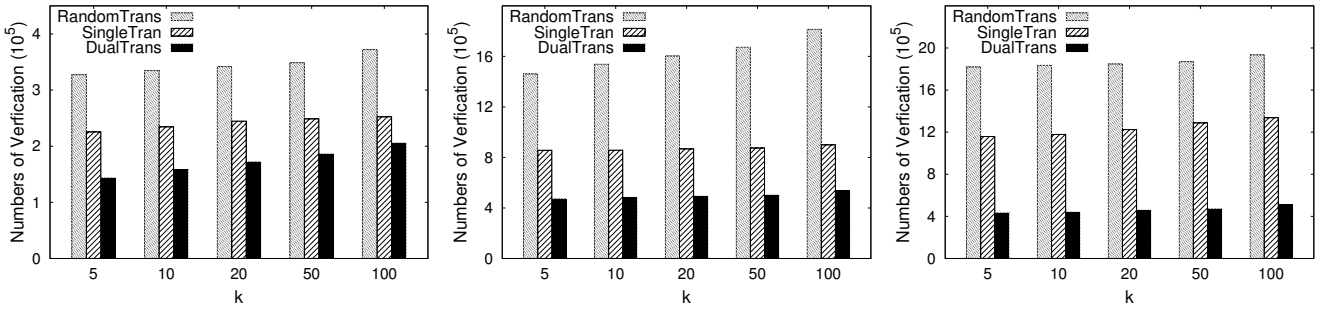
1) DBLP [1]: a collection of titles and authors from dblp computer science bibliography. The goal of this collection is to provide real bibliography that is based on real scenarios. It could be used for query reformulation or other types of search research. We tokenized the records in this dataset following previous study, that is we split all records based on non-alphanumeric characters.
2) PUBMED [2]: a dataset of basic information of biomedical literature from PubMed in XML format. We select the abstract part of the datasets and divide the abstracts into tokens based on spaces.
3) LIVEJOURNAL [3]: the dataset contains a list of user group memberships. Each line contains a user identifier followed by a group identifier (separated by a tab), implying that the user is a member of the group.

1. http://dblp.uni-trier.de/
2. https://www.ncbi.nlm.nih.gov/pubmed/
3. http://socialnetworks.mpi-sws.org/data-imc2007.html



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 5. Effect of Proposed Techniques: Query Time



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 6. Effect of Proposed Techniques: Number of Candidates

4) KOSARAK [4]: the dataset is provided by Ferenc Bodon and contains anonymous click-stream data of a Hungarian on-line news portal.

The detailed statistical information is shown in Table 3. We evaluate our disk-based algorithms on the PUBMED dataset, which is much larger than the other datasets. And the other three datasets are used to evaluate the in-memory algorithms. We implemented our methods with C++ and compiled using GCC 4.9.4 with -O3 flag. We obtain the source codes of all baseline methods from the authors which are also implemented with C++. All the experiments were run on a Ubuntu server machine with 2.40GHz Intel(R) Xeon E52653 CPU with 16 cores and 32GB memory.

We compare our method with state-of-the-art methods on both in-memory and disk-based settings. To the best of our knowledge, among all existing studies, only MultiTree [30] and the top-k search algorithm proposed in Flamingo [21](ver. 4.1) support KNN set similarity search. We also extend the top-k set join algorithm proposed in [25] based on their original implementation to support KNN set similarity join and proposed PP-Topk algorithm in the following way: we group the records by length and build the index for the maximum length of prefix. Then we start from Jaccard similarity 1.0 and incrementally decrease the value of similarity according to the techniques proposed in [25] until we collect k results. For disk-based settings, we only compare with disk-based algorithm of Flamingo [5] as MultiTree only works for in-memory settings. We will first show the results of in-memory settings (Section 7.2-7.5) and then show it disk-based settings (Section 7.6). To initialize the R-Tree, our implementation performs bulk loading to construct the R-Tree from a dataset in one time and then perform the queries.

For approximate KNN set similarity search, we extend min-hash algorithm [6] to do approximate KNN set search, denoted as MinHash as the baseline method of our Approx algorithm. In MinHash, we first utilize min-hash algorithm to transform the original set records into hashed vectors. Then we utilize the similar approach proposed in Approx to incrementally search for the approximate KNN result.

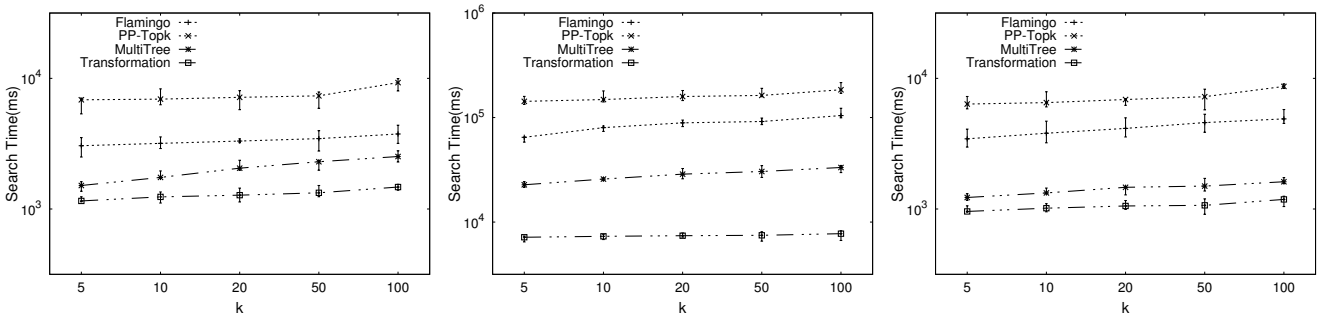
4. http://fimi.ua.ac.be/data/

7.2 Effect of Proposed Techniques

We first evaluate the effectiveness of our transformation techniques. To this end, we propose three methods: RandomTrans is the method that randomly groups the tokens into m groups to generate the transformation. SingleTran is the method that uses one transformation generated by the greedy grouping mechanism. DualTrans is the dual-transformation based method.

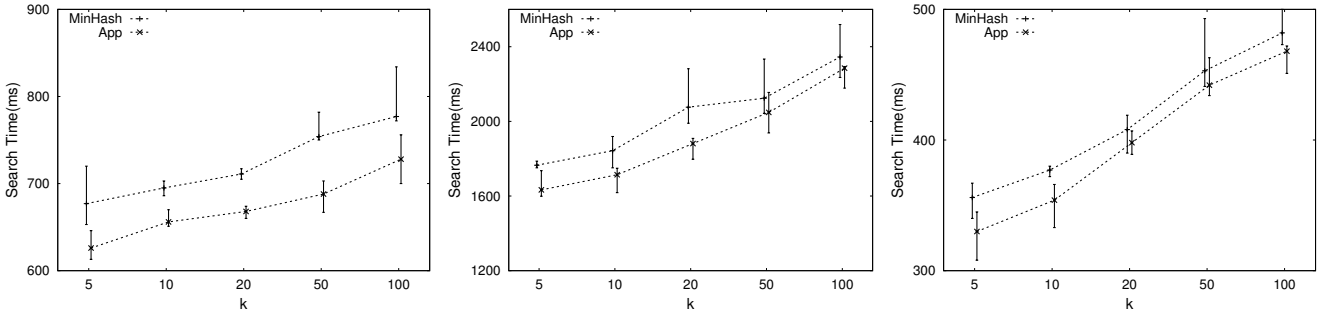
The results of average query time are shown in Figure 5. We can see that among all methods, DualTrans has the best performance. The reason is that by utilizing two transformations, it can capture more characteristics regarding set similarity. Thus it can provide a tighter bound of JACCARD similarity and prune more dissimilar records. Compared with RandomTrans, SingleTran has much better performance. This is because random generated transformation always leads to uneven distribution among different groups, which results in looser bound of JACCARD similarity compared to SingleTran. It demonstrates the importance to generate a proper transformation in order to achieve good performance. For example, on dataset LIVEJOURNAL when $k = 5$, the query time of RandomTrans is 25784 ms; while the query time of SingleTran and DualTrans are 11218 ms and 7161 ms, respectively.

Actually, from the aspects of performance, DualTrans outperforms SingleTran obviously. This is because for DualTrans, we construct two orthogonal transformations to calculate upper bound individually whose quantity relation differs with specific nodes or records to enlarge pruning power. The only extra cost for DualTrans, compared to SingleTran, is that the number of comparisons in each region of R-tree is doubled than that in SingleTran. This would slightly increase the filter cost but will



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 7. Compare with State-of-the-art Methods: Exact Algorithms



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 8. Compare with State-of-the-art Methods: Approximation Algorithms

definitely result in greater filter power, i.e., much smaller number of verifications. From Figure 6, we also find that the pruning power of DualTrans exceeds that of SingleTran because of our ingenious design for multiple transformation.

In order to further demonstrate the filter power of proposed methods, we also report the number of candidates for each method as shown in Figure 6. DualTrans has the least number of candidates. While RandomTrans has the largest number of candidates under all settings. This is corresponding with the results shown in Figure 5. For instance, on the LIVEJOURNAL for $k = 5$, RandomTrans involves 1,460,904 candidates, while SingleTran reduces the number to about 856,875. And DualTrans involves only 474,204 candidates. From above results, we can conclude that our proposed transformation techniques can obviously improve the filter power as the number of candidates is reduced.

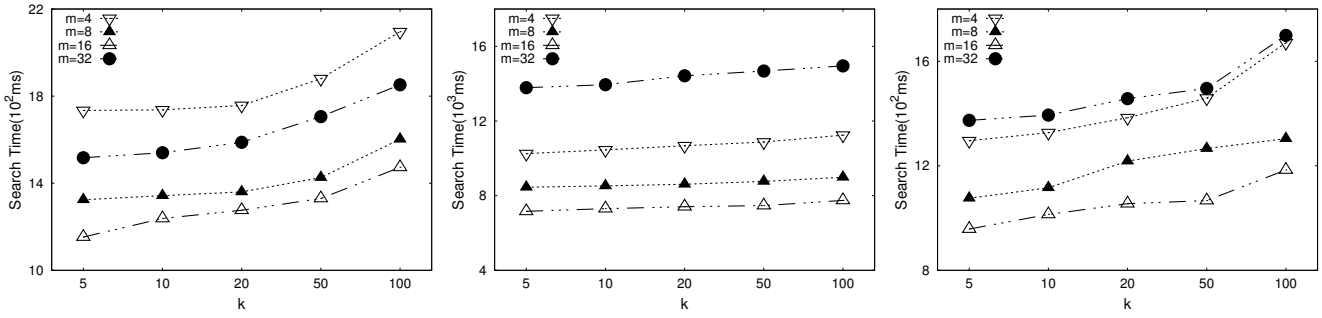
Next we study the effect of some parameters. First we look at the dimension m of representative vectors . Figure 9 shows the results when value of m varies. We can see that when $m = 16$, our method has the best results. This is consistent with our implementation strategy. Considering the memory alignment, we'd better to let m be power of 2. Specifically, we assign 2 bytes for each dimension and store all dimensions of representative vector contiguously in memory layout. Actually, 2 bytes (0-65535) are large enough for the number of tokens in one dimension of a record. Besides, for DualTrans, each transformation in DualTrans have $m/2$ dimensions. So if m becomes smaller, each transformation will only have 4 dimensions, which is insufficient to distinguish different records. Moreover, when $m = 16$, it requires 16 bytes to store the representative vectors. As the size of cache line is 64B in modern computer architecture, each cache line could hold 2 representative vectors. In addition, we need 2 vectors to represent an MBR of one node in R-tree. Therefore, when we perform searching in R-tree, we can acquire the MBR of a node by visiting one cache line so as to improve the overall performance.

Also, we study the effect of ϵ in Approx since it plays an important role on search time and recall rate. During the experiments, the results on all three datasets show the similar trends. Due to the space limitation, here we only report the result on LIVEJOURNAL. Figure 10 shows the results when ϵ varies on LIVEJOURNAL. It is obvious that the recall rate will be higher while Approx costs more search time with larger ϵ. However, we can see that when ϵ is larger than 1000, the recall rate is growing slowly and stays at a relatively high value ($\geq 75\%$) while the search time still increases rapidly. Therefore, we choose $\epsilon = 1000$ for our experiments. While Approx has reasonably high recall rate, it requires much less search time than the exact algorithm. For example, when $k = 50$ Approx only needs 2048ms, while DualTrans requires 7468ms.

7.3 Compared with State-of-the-art Methods

Next we compare our DualTrans with state-of-the-art methods. For each dataset, we randomly select 10,000 records from it as query and report the average query time. We also also show some min-max error bars to assess the stability of results. For all baseline methods, we try our best to tune the parameters according to the descriptions in previous studies and report their best results. Here Transformation is the DualTrans method in Figure 5, which has the best performance.

The results of comparing Transformation with Flamingo, PP-Topk and MultiTree are shown in Figure 7. We can see that our methods has 1.27 to 23.81 (on average 4.51) times performance gain than state-of-the-art methods. For example, on the dataset LIVEJOURNAL while $k = 20$, Flamingo took 89,024 ms, while Transformation only took 7,408 ms. The reason is that Flamingo spends too much time on scanning inverted lists. And for records with larger average length, this filtering process becomes more expensive. As the complexity of calculating JACCARD similarity is only $\mathcal{O}(n)$, its not proper for Flamingo to spend too much time on improving filter power. For MultiTree, as



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 9. Effect of Representation Vector Dimensions

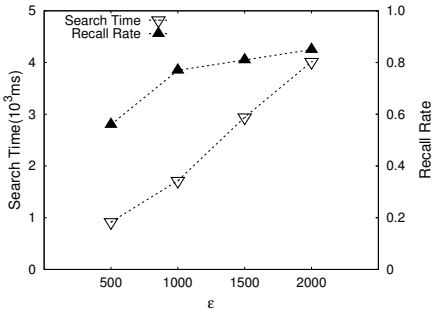


Fig. 10. Effect of ϵ in Approx

they map records into one-dimensional numerical value, they will lose significant portion of information and thus lead to poor filter power. Also, though many optimization approaches are proposed to speedup the join performance by [25], such techniques are more suitable for join rather than search. Therefore, we can see from PP-Topk that directly extending from existing study will lead to suboptimal performance.

We also show the result of the approximate KNN algorithm Approx in Figure 8. The performance of Approx significantly outperforms MinHash. The reason is that Approx only needs to visit the bucket and can avoid traversing the index. Besides, the search space is also much smaller as we only need to perform incremental search on the buckets. The reason is that the hash index of MinHash distributes messier than that of Approx, and MinHash needs to access more nodes than Approx to get the same candidate records. At the same time, we report the recall of Approx (App) and MinHash (MH) for each value of k in Table 4. It ic computed by dividing the number of correct top-k records by k. As the top-k results are not unique, we will treat any result with correct similarity as valid. We can see that except for the efficiency of Approx, the overall recall rate is also better. For example, on dataset LIVEJOURNAL the average recall rate of Approx is 0.742 while that of MinHash is 0.464. The main reason is that MinHash specifies the order of the dimensions. However, this order could cause some false negative. For instance, if the min-hash results of record are same as that of query records but different from the first index, though they tend to be similar in high possibility, the record will be accessed later than other records with the same first index which might be dissimilar.

7.4 Indexing Techniques

Next we report the results about indexing. Here we focus on two issues: index size and index construction time. The index sizes of different methods are shown in Table 5. The index size of our

TABLE 4
The Recall Rate of Approx

k	KOSARAK		LIVEJOURNAL		DBLP		PUBMED
	MH	App	MH	App	MH	App	App
5	0.42	0.65	0.48	0.82	0.33	0.64	0.75
10	0.40	0.71	0.44	0.77	0.37	0.69	0.73
20	0.38	0.68	0.39	0.68	0.42	0.71	0.67
50	0.41	0.64	0.51	0.69	0.38	0.63	0.74
100	0.37	0.66	0.50	0.75	0.36	0.62	0.77

TABLE 5
Index Size

size(MB)	Flamingo	PP-Topk	MultiTree	DualTrans
KOSARAK	193.4	217.5	68.3	118.7
LIVEJOURNAL	753.3	782.1	461.5	625.2
DBLP	525.7	544.8	453.4	606.3
PUBMED	6075.1	N/A	N/A	3906.4

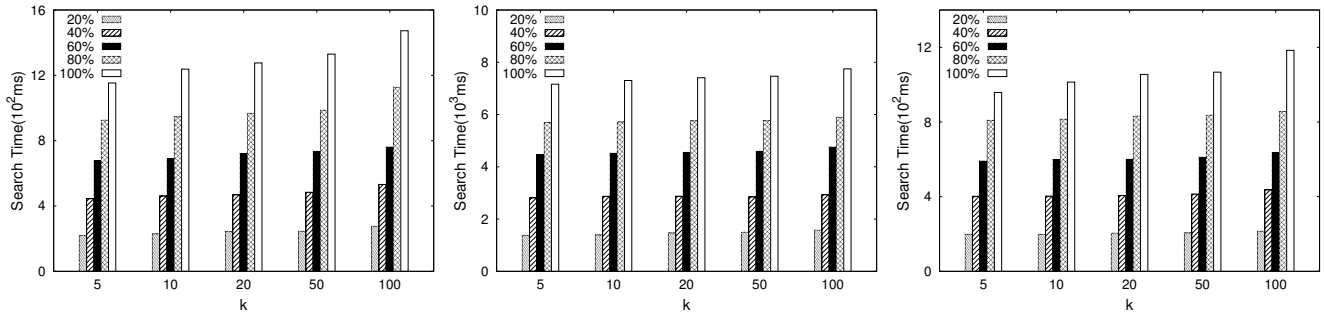
method is significantly smaller than that of Flamingo. This is because we just store the data in leaf nodes of R-Tree and do not build inverted lists. In this way, the index size only relies on the cardinality of dataset rather than the length of records. Among all methods, MultiTree has the smallest index size. The reason is that MultiTree maps each record into a numerical value and constructs a B^+-Tree like index on them. However, there might be significant loss of useful information in this process. Compared with MultiTree, the index size of our method is only slightly larger but our method achieves much better search performance.

The index construction time is shown in Figure 13. Actually the index construction time of both Transformation and MultiTree is significantly less than Flamingo. This is because these two methods do not need to build inverted lists. Meanwhile, Transformation has comparable index construction time with MultiTree. For our method, the main cost of this process is to map the original records into representative vectors and generate dual transformation.

7.5 Scalability

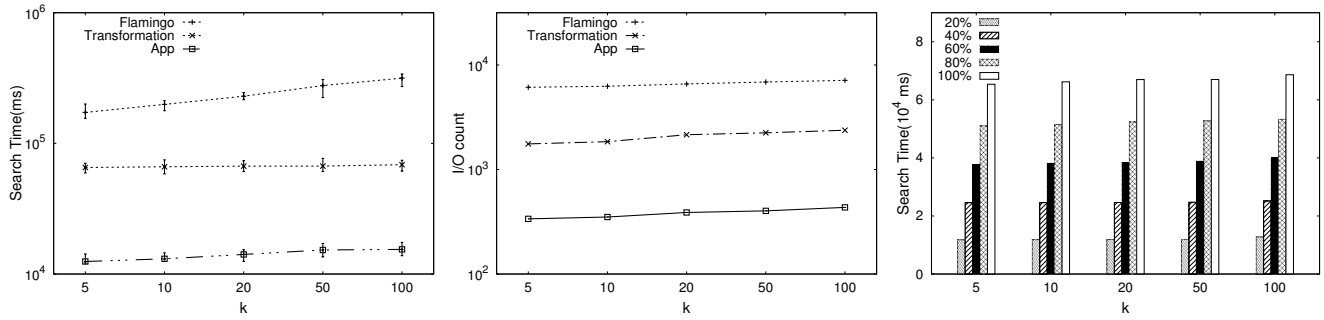
Then we evaluate the scalability of our method. Here we report the results of Transformation. On each dataset, we vary the number of records from 20% to 100% and report the average search time for each threshold. The results are shown in Figure 11. We can see that with the increasing number of records in the datasets, our method scales very well with the size of dataset and achieves near linear scalability. For example, on the LIVEJOURNAL data sets, when $k = 10$, the average search time for 20%, 40%, 60%, 80%, 100% size of dataset are 1393 ms, 2871 ms, 4523 ms, 5713 ms and 7304 ms, respectively.

Besides, we can see that Transformation is also insensitive to the variation of k. For example, on the dataset LIVEJOURNAL, for 20% size of datasets, the average search time for



(a) KOSARAK (b) LIVEJOURNAL (c) DBLP

Fig. 11. Scalability: Effect of Data Size



(a) Compare with state-of-art methods (b) I/O Cost with state-of-art methods (c) Scalability

Fig. 12. Disk-based Algorithm

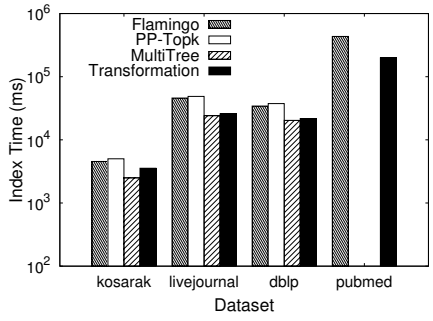


Fig. 13. Index Construction Time

$k = 5, 10, 20, 50, 100$ are 1376ms, 1393ms, 1467ms, 1498ms and 1579ms, respectively. The reason is that with the well designed dual transformation, we can group similar records into one leaf node of the R-Tree index. Then we performing KNN search, it is very likely that other dissimilar nodes are pruned and we can find the k results within very few leaf nodes.

7.6 Evaluate Disk-based Settings

Our proposed methods can also support disk-based settings. Unlike the in-memory scenario, all the index and data will be located on disk. Here we evaluate the proposed methods using PUBMED dataset, which is much larger than the other three datasets.

We first compare our proposed method with state-of-the-art method. The results are shown in Figure 12(a). Our methods perform better than Flamingo in different k values. For example, when $k = 5$, the average search time of DualTrans is 65365 ms, which is significantly less than that of Flamingo, 172615 ms. The reason is that our proposed method utilizes the R-tree structure to avoid unnecessary access of index, which significantly reduces the I/O cost compared with Flamingo. To demonstrate this, we also report the number of I/Os required by each method in Figure 12(b). The results are consistent with that in Figure 12(a). For instance, when the $k = 5$, the average number of I/O access of Transformation is 1758, while that of Flamingo is 6123. For indexing issues, the results show similar trend with the in-memory settings, as is shown in Figure 13 and Table 5.

We also show the performance of Approx disk-based settings. Besides its good efficiency, the recall rate is also promising as is shown in Table 4. Furthermore, Approx can save a great number of I/O cost as it only needs to load the buckets with overlapping one by one and does not need to traverse the R-Tree index.

Finally we report the scalability of Transformation for disk-based settings, Figure 12(c) shows that our method achieves good scalability among different numbers of records. For example, when $k = 10$, the average search time for different sizes are 11856ms, 24602ms, 38174ms, 51474ms and 66200ms, respectively.

8 CONCLUSION

In this paper, we study the problem of KNN set similarity search. We propose a transformation based framework to transform set records to fixed length vectors so as to map similar records closer to each other. We then index the representative vectors using R-Tree index and devise efficient search algorithms. We also propose approximate algorithm to accelerate the KNN search process. Experimental results show that our exact algorithm significantly outperforms state-of-the-art methods on both memory and disk settings. And our approximate algorithm is more efficient and with high recall rate at the same time. For the future work, we plan to extend our techniques to support KNN problem in other field, such as spatial and temporal data management. Besides, we would also like to conduct more probabilistic analysis on the basis of our transformation based techniques.

Acknowledgment We would like to thank all editors and reviewers for their valuable suggestions. This work was supported

by NSFC(91646202), National Key Technology Support Program of China (2015BAH13F00), National High-tech R&D Program of China(2015AA020102). Jin Wang is the corresponding author.

REFERENCES

[1] S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *SIGMOD*, pages 13–24, 1999.

[2] T. D. Ahle, R. Pagh, I. P. Razenshteyn, and F. Silvestri. On the complexity of inner product similarity join. In *PODS*, pages 151–164, 2016.

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

[4] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[5] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, pages 888–899, 2011.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

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

[8] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[9] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[10] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: data sensitive hashing for high-dimensional k-nnsearch. In *SIGMOD*, pages 1127–1138, 2014.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[13] M. Jiang, A. W. Fu, and R. C. Wong. Exact top-k nearest keyword search in large networks. In *SIGMOD*, pages 393–404, 2015.

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

[15] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[16] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[18] M. A. Soliman, I. F. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

[19] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *PVLDB*, 8(1):1–12, 2014.

[20] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[21] R. Vernica and C. Li. Efficient top-k algorithms for fuzzy search in string collections. In *KEYS*, pages 9–14, 2009.

[22] 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 *ICDE*, pages 519–530, 2015.

[23] X. Wang, X. Ding, A. K. H. Tung, and Z. Zhang. Efficient and effective KNN sequence search with approximate n-grams. *PVLDB*, 7(1):1–12, 2013.

[24] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.

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

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

[27] Z. Yang, J. Yu, and M. Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.

[28] K. Yi, X. Lian, F. Li, and L. Chen. The world in a nutshell: Concise range queries. *IEEE Trans. Knowl. Data Eng.*, 23(1):139–154, 2011.

[29] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*, pages 997–1008, 2011.

[30] 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 *ICDE*, pages 759–770, 2017.

[31] 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*, pages 915–926, 2010.

[32] B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li. Keyword-aware continuous knn query on road networks. In *ICDE*, pages 871–882, 2016.

Yong Zhang is associate professor of Research Institute of Information Technology at Tsinghua University. He received his BSc degree in Computer Science and Technology in 1997, and PhD degree in Computer Software and Theory in 2002 from Tsinghua University. From 2002 to 2005, he did his Postdoc at Cambridge University, UK. His research interests are data management and data analysis. He is a member of IEEE, and a senior member of China Computer Federation.

Jiacheng Wu is a master student in Department of Computer Science and Technology, Tsinghua University. He obtained the bachelor degree from School of Software Engineering, Nankai University in 2018. His research interest includes operating system and scalable data analysis.

Jin Wang is a PhD student in Computer Science Department, University of California, Los Angeles. Before joining UCLA, he obtained his master degree in computer science from Tsinghua University in the year 2015. His research interests include text analysis and processing, stream data management and database system.

Chunxiao Xing is professor and associate dean of Information Technology Research Institute (RIIT), Tsinghua University, and director of Web and Software RD Center (WeST). He is the director of Big Data Research Center for Smart Cities, Tsinghua University. He is also the deputy director of the Office Automation Technical Committee of China Computer Federation, a member of China Computer Federation Technical Committee on databases, big data and software engineering. He is also the member of IEEE and ACM.

APPENDIX A
PROOF OF LEMMA 1

Proof. First, we could deduce the JACCARD distance from JACCARD similarity:

$$\mathsf{JacDist}(X,Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|} \tag{12}$$

$$= 1 - \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|} \tag{13}$$

$$= 1 - (\frac{|X| + |Y|}{|X \cap Y|} - 1)^{-1} \tag{14}$$

Then we generate $\omega[X]$ and $\omega[Y]$, the representative vectors of X and Y. With the help of $\omega[X]$ and $\omega[Y]$, we can estimate the upper bound of their overlaps as: $|X \cap Y| \leq \sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y])$. Thus, we can assert that $\mathsf{TransDist}(\omega, X, Y) \leq \mathsf{JacDist}(X, Y)$, which finishes the proof. □

APPENDIX B
PROOF OF THEOREM 1

Proof. We can make a reduction of optimal transformation from 3-SAT problem. Firstly, our problem is as hard as the following one: given a set P of points, a constant S, and the dimension k of representative vector, for all records $X \in \mathcal{U}$, decide whether there exists a transformation ω s.t. $\sum_{\substack{\langle X,Y\rangle \in \mathcal{U}^2 \\ X \neq Y}} \sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y]) = S$.

Next we further add more constraints and then try to reduce above problem into vertex coloring problem. Here we assume the all records only contains one token and therefore after the transformation, all representative vectors only contains a 1 in one dimension and all 0 for other dimensions. As the real problem is more complex than the one with above assumption. We only need to prove the above one is NP-Hard.

Then we consider each record as a vertex in graph. And if we need to calculate $\sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y])$, then there is an edge between two nodes representing records X and Y. Then we regard those different types of representative vectors as different colors, i.e., the dimension k means k different colors. Thus, the transformation is just like a coloring strategy. Moreover, under this assumption, the value of $\sum_{i=1}^{m} \min(\omega_i[X], \omega_i[Y])$ is 1 iff two nodes representing records X, Y were connected by an edge has the same color and 0 otherwise.

Therefore, we can rewrite the problem as following: Given a graph $G =< V, E >$ and k colors, find a coloring strategy where sum of pairs of nodes connected by edges with the same color equals a given constant S. If S is 0, then the problem is as hard as the a known NP-hard problem k-coloring. When S is larger than 0, actually there must exist S pairs of adjacent nodes with same color, then we could merge each pairs of adjacent nodes into one node. Then the edges originally connected to those pairs of nodes now connect to those merged nodes, individually. As a result, we construct a new graph whose S equals 0. In this way, we reduce optimal transformation to the graph coloring problem. □

APPENDIX C
PROOF OF LEMMA 2

Proof. Given a query Q and a node N, we need to prove that $\mathsf{MinDist}(\omega, Q, N) \leq \min_{R \in N} \mathsf{JacDist}(Q, R)$. First we need to prove the following conclusion:

$$\min_{R \in N} \frac{\sum_{i=1}^{m}(\omega_i[Q] + \omega_i[R])}{\sum_{i=1}^{m} \min(\omega_i[Q], \omega_i[R])} \geq \frac{n(\omega, Q, N)}{d(\omega, Q, N)} \tag{15}$$

We can denote the lhs of Equation 15 as the function:
$\mathcal{F}(\omega_i[R]) = \frac{C_1 + \omega_i[Q] + \omega_i[R]}{C_2 + \min(\omega_i[Q], \omega_i[R])}$, where we have $\omega_i[R] \in [\mathcal{B}_j^{\perp}, \mathcal{B}_j^{\top}]$ with given MBR of node N for specific dimension i, and regard other $\omega_{j \neq i}[R]$ as constants. Considering the relation between $\omega_i[Q]$ and $[\mathcal{B}_j^{\perp}, \mathcal{B}_j^{\top}]$, we can rewrite the function as:

$$\mathcal{F}(\omega_i[R]) = \begin{cases} \frac{C_1 + \omega_i[Q] + \omega_i[R]}{C_2 + \omega_i[Q]} & \omega_i[Q] < \mathcal{B}_i^{\perp} \\ \frac{C_1 + \omega_i[Q] + \omega_i[R]}{C_2 + \min(\omega_i[Q], \omega_i[R])} & \mathcal{B}_i^{\perp} \leq \omega_i[Q] < \mathcal{B}_i^{\top} \\ \frac{C_1 + \omega_i[Q] + \omega_i[R]}{C_2 + \omega_i[R]} & \mathcal{B}_i^{\top} \leq \omega_i[Q] \end{cases} \tag{16}$$

Since $\omega_i[R] \in [\mathcal{B}_j^{\perp}, \mathcal{B}_j^{\top}]$, we could get the minimum value among:

$$\begin{cases} \frac{C_1 + \omega_i[Q] + \mathcal{B}_i^{\perp}}{C_2 + \omega_i[Q]} (\omega_i[R] = \mathcal{B}_i^{\perp}) & \omega_i[Q] < \mathcal{B}_i^{\perp} \\ \frac{C_1 + \omega_i[Q] + \omega_i[Q]}{C_2 + \omega_i[Q]} (\omega_i[R] = \omega_i[Q]) & \mathcal{B}_i^{\perp} \leq \omega_i[Q] < \mathcal{B}_i^{\top} \\ \frac{C_1 + \omega_i[Q] + \mathcal{B}_i^{\top}}{C_2 + \mathcal{B}_i^{\top}} (\omega_i[R] = \mathcal{B}_i^{\top}) & \mathcal{B}_i^{\top} \leq \omega_i[Q] \end{cases} \tag{17}$$

Therefore, we would like to minimize $\mathcal{F}(\omega_i[R])$ individually for each $\omega_i[R]$ to get the minimum value. Then the minimum value of $\mathcal{F}$ can be written as $\frac{n(\omega,Q,N)}{d(\omega,Q,N)}$. Thus, we complete the proof of Equation 15.

In proof of Lemma 1, we assert the $\mathsf{JacDist}(Q, R) \geq 1 - (\frac{|Q| + |R|}{\sum_{i=1}^{m} \min(\omega_i[Q], \omega_i[R])} - 1)^{-1}$. Thus,

$$\min_{R \in N} \mathsf{JacDist}(Q, R) \geq 1 - (\min_{R \in N} (\frac{|Q| + |R|}{\sum_{i=1}^{m} \min(\omega_i[Q], \omega_i[R])}) - 1)^{-1} \tag{18}$$

Also as $|Q| + |R| = \sum_{i=1}^{m}(\omega_i[Q] + \omega_i[R])$, we get that

$$\min_{R \in N} \mathsf{JacDist}(Q, R) \geq 1 - (\frac{n(\omega, Q, N)}{d(\omega, Q, N)} - 1)^{-1} = \mathsf{MinDist}(\omega, Q, N) \tag{19}$$

And thus the proof completes. □

APPENDIX D
PROOF OF THEOREM 2

Proof. We only need to prove that given the query Q and the transformation ω, if child N_c satisfies $\mathsf{MinDist}(\omega, Q, N_c) \geq \mathsf{UB}^{\mathcal{R}}$, then the child could be pruned safely. Based on Lemma 2, $\mathsf{MinDist}(\omega, Q, N_c)$ is the lower bound of JACCARD distance between Q and any record $X \in N_c$. If $\mathsf{MinDist}(\omega, Q, N_c) \geq \mathsf{UB}^{\mathcal{R}}$, the JACCARD distance between any records $X \in N_c$ and Q is no smaller than $\mathsf{UB}^{\mathcal{R}}$, which is the *upper bound of Jaccard Distance* for current KNN candidate results. Therefore, any record $X \in N_c$ cannot be the candidate of KNN results. □

APPENDIX E
PROOF OF LEMMA 3

Proof. Given records X, Y and a set of different transformations $\biguplus_{\Omega}$. Based on Lemma 1, we could say that for each $\omega_i \in \Omega$, inequality $\mathsf{TransDist}(\omega_i, X, Y) \leq \mathsf{JacDist}(X, Y)$ holds. Then the following inequality holds:

$$\max_{1 \leq i \leq |\Omega|} \mathsf{TransDist}(\omega_i, X, Y) \leq \mathsf{JacDist}(X, Y) \tag{20}$$

Therefore, $\mathsf{TransDist}(\biguplus_{\Omega}, X, Y) \leq \mathsf{JacDist}(X, Y)$.

Based on the proof, we also find that this Multiple-Transformation Distance is a tighter lower bound of JACCARD distance compared with individual transformation distance. □

APPENDIX F
PROOF OF LEMMA 4

Based on Lemma 4, it is obvious that for each $\omega_i \in \Omega$, the inequality $\mathsf{MinDist}(\omega_i, Q, N) \leq \min_{R \in N} \mathsf{JacDist}(Q, N)$ holds:

$$\begin{aligned} &\mathsf{MinDist}(\omega_i, Q, N) \leq \max_{1 \leq i \leq |\Omega|} (\mathsf{MinDist}(\omega_i, Q, N)) = \\ &\mathsf{MinDist}(\biguplus_{\Omega}, Q, N) \leq \min_{R \in N} \mathsf{JacDist}(Q, N) \end{aligned} \tag{21}$$

Therefore, based on the latter inequality, $\mathsf{MinDist}(\biguplus_{\Omega}, Q, N)$ is the lower bound of JACCARD distance; and based on the former inequality, $\mathsf{MinDist}(\biguplus_{\Omega}, Q, N)$ is tighter than individual $\mathsf{MinDist}(\omega_i, Q, N)$. □

APPENDIX G
PROOF OF THEOREM 3

Proof. We can make a reduction of optimal bucket construction from Planar 3-SAT problem.

Our problem can be rewritten as the following problem: Finding p partitions of the set U of points with p MBRs to minimize the information loss(the definition is as follows) of U. The information loss for a partition is defined as $n_i \sum_{i=1}^{m} L_i$. And the information loss of U is the summation of information loss for all partitions. The special case of above problem (when $m = 2$) has been proved to be NP-hard (See Theorem 2 in [28]) by reducing the PLANAR 3-SAT,which is an NP-complete problem [15] to this case. Hence, any instance of above problem, which is NP-hard, can be reduced to an instance of our problem. □

APPENDIX H
EXAMPLE OF ALGORITHM 3 AND ALGORITHM 4

Example 7. Here is the example of Algorithm 3 on the collections of records in Example 5. We first get the transformation $\mathcal{G}$ by applying Algorithm 1. We have: $G_1 = \{x_1, x_5, x_9, x_{13}\}$, $G_2 = \{x_2, x_6, x_{10}, x_{14}\}$, $G_3 = \{x_3, x_7, x_{11}, x_{15}\}$ and $G_4 = \{x_4, x_8, x_{12}, x_{16}\}$. We regard these groups as collections of records, apply Greedy Grouping Mechanism again on each group and collect their results into $\Im$. For instance, when applying Greedy Grouping Mechanism on G_1, then the result $\mathcal{K} = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}\}$. Therefore, $\Im = \{\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}\}, \{\{x_5\}, \{x_6\}, \{x_7\}, \{x_8\}\}, ...\}$.
Then according to Algorithm 3, we first get $\mathcal{K} = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}\}$, then for each group in $\mathcal{K}$, we allocate H_1 according to the frequency. As H_1 does not contain other group K_k, we assign all tokens in K_1 to H_1. Then we have $H_1 = \{x_1\}$. We repeat this procedure and finally get the transformation $\mathcal{H}$, $H_1 = \{x_1, x_2, x_3, x_4\}$, $H_2 = \{x_5, x_6, x_7, x_8\}$, $H_3 = \{x_9, x_{10}, x_{11}, x_{12}\}$ and $H_4 = \{x_{13}, x_{14}, x_{15}, x_{16}\}$. According to the definition of "similar transformation", we can see that transformations $\mathcal{G}$ and $\mathcal{H}$ are dissimilar.

Example 8. Figure 4 demonstrates an running example of Algorithm 4. We present the first three layer nodes of R-tree to show the process of generating buckets. Here all nodes in third layer are internal nodes, which the dotted lines under the leftmost node indicate. Here a red ellipse around nodes means a bucket. The caption shown in nodes consists of two parts, the MBR of nodes and the numbers of records in the nodes.
Here we want to generate $p = 3$ buckets in the situation where $m = 2$. At first, current $i = 1$ and then we add R_1 into b_1, and calculate the value $\Upsilon(b_1) = (6 + 20\sqrt{5})tL$. while trying to add R_2 into b_1, $MBR(b_1) = [0, 2L] \times [0, 2L]$. while the MBR of the rest area is $[2L, 6L] \times [0, 2L]$, which means $\mathcal{A}(U') = 8L^2$. Then we can get $n_1 = 4t$ and $n' = 8t$. Therefore, $\Upsilon(b_1) = 4t*2*2L + 8t*2*\sqrt{4L^2} = 48tL < (6+20\sqrt{5})tL$. Therefore, we add R_2 into b_1 and continue the iteration. Next we are trying to add R_3. As the same procedural above, $\Upsilon(b_1) = 60tL > 48tL$, which means this node cannot be added into current bucket. Therefore, b_1 only contains R_1, R_2. After that, we need to handle the next bucket b_2. With the same procedure, we add R_3, R_4, R_5 into b_2. Finally, we add the remaining nodes R_6, R_7, R_8 into b_3. Therefore, we finish the buckets constructions based on the non-leaf node from the tree.