World Wide Web (2011) 14:573–597
DOI 10.1007/s11280-011-0137-1

Efficient top-K approximate searches against a relation with multiple attributes

Wei Lu · Jinchuan Chen · Xiaoyong Du · Jieping Wang · Wei Pan

Received: 22 April 2010 / Revised: 13 June 2011 /
Accepted: 14 June 2011 / Published online: 12 August 2011

Abstract In this paper, we study the problem of efficiently identifying K records that are most similar to a given query record, where the similarity is defined as: (1) for each record, we calculate the similarity score between the record and the query record over each individual attribute using a specific similarity function; (2) an aggregate function is utilized to combine these similarity scores with weights and the aggregated value is served as the similarity of the record. After similarities of all records have been computed, K records with the greatest similarities can further be identified. Under this framework, unfortunately, the computational cost will be extremely expensive when the cardinality of relation is large as computation of similarity for each record is required. As a result, in this paper, we propose two

W. Lu · X. Du (✉)
School of Information, Renmin University of China, Beijing, 100872, China
e-mail: duyong@ruc.edu.cn

W. Lu
e-mail: uqwlu@ruc.edu.cn

W. Lu · J. Chen · X. Du
Key Labs of Data Engineering and Knowledge Engineering, Ministry of Education, Beijing, China

J. Chen
e-mail: jcchen@ruc.edu.cn

J. Wang
China Electronics Standardization Institute, Beijing, China
e-mail: wangjieping@cesi.ac.cn

W. Pan
School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, China
e-mail: panwei1002@nwpu.edu.cn

efficient algorithms, named ScanIndex and Top-Down (TD for short), to cope with this problem. With respect to ScanIndex, similarity scores that are equal to zero over individual attributes are free from computation. Based on ScanIndex, with respect to TD, similarity scores less than thresholds (rather than zero) over individual attributes are skipped, where these thresholds are improved dynamically over time. Experimental results demonstrate that, comparing with the naive approach, the performance can be improved by two orders of magnitude using ScanIndex and TD.

Keywords top-K queries · approximate search · data quality

1 Introduction

Data quality is one of the fundamental problems in the organizations' database systems. Due to a number of reasons such as misspelling, typographical and transcription errors in processing OCR data capture, multiple conversions of entity names, update anomalies, lack of integrity constraints and combination thereof, the quality problem is rampant in data management applications. Identifying records that refer to the same real-world entity as a given query record is imperative since the traditional equivalence or containment operations cannot work any further. This problem is sometimes known as approximate search in the database community.

Traditional manner of approximate searches in relational databases requires users to pre-specify a threshold over each attribute of interest [2, 6, 17, 21], and any record, whose similarity score[1] over each attribute is greater than the pre-specified threshold, is regarded to represent the same real-world entity as the query record. Here, each attribute value of a record is considered as a string, and the similarity score of a record over each individual attribute is calculated by a given similarity function, such as edit distance [16], Jaccard [30], TF/IDF cosine coefficiency [7], language model [4], etc, which are widely used to compute the similarity score between two strings. However, traditional manner of approximate searches may not be applicable in many real life scenarios since the pre-specified thresholds are usually unknown beforehand. Specifying improper thresholds might lead to either too many results or empty results. In this paper, we concentrate on how to efficiently identify K records which are most similar to the query record.

For a relation with only a single attribute, the similarity[1] of each record can be calculated using a given similarity function as each record can be taken as a single string. However, there usually exist multiple attributes in an underlying relation. Defining the similarity of each record properly is required and necessary. In [23, 24, 31, 33–35], they concatenate attribute values of the same record into a single string and the similarity of each record is defined using a given similarity function.

[1]For the sake of differentiation, the term *similarity score* refers to how similar a record and the query record over an attribute are, while the term *similarity* refers to how similar a record and the query record are.

Figure 1 A relation and a query record.

Tuple ID	Name	Address
r_1	Wei Wang	101 Cornwall St Annerley
r_2	Wei Wan	707 Cornwall Rd Annerley
r_3	Wei Wang	111 Cornwall Av Fairfield
r_4	Mei Wang	312 Springhills Duton Park
r_5	Fang Wang	102 Anne Av Sunnybank

(a) A relation R

Name	Address
Wei Wang	707 Cornwall Av Annerley

(b) A query Q

Yet, such concatenation suffers from loss of statistical information. In [21], they propose a weighted vector model to define the similarity of each record. Specifically, under this model, each attribute is pre-assigned with a weight and weights over all attributes constitute a weighted vector. Considering a relation R, symbol A refers to the set of attributes in R. Given a query record q, the similarity of a record $r \in R$, denoted as $sim(q, r)$, is defined as follows:

$$sim(q, r) = \sum_{\forall a \in A} sim(q.a, r.a) \times \omega_a, \qquad \sum_{\forall a \in A} \omega_a = 1 \tag{1}$$

where $sim(q.a, r.a)$ is the similarity score between q and r over attribute a and ω_a is the weight over attribute a.

In this paper, the similarity between a record and q is defined using the weight vector model, shown in (1). Under this model, after similarities of all records have been calculated, K records with the greatest similarities, *top-K records* for short, can further be identified.

Example 1 Figure 1a shows a relation with five records and Figure 1b shows a query record q. Suppose we want to figure out the top-2 records in R. Using Jaccard similarity function by tokenizing each string as a set of 3-grams, similarity scores of records over each attribute can be calculated, and shown in columns 2 and 3 of Figure 2. Suppose the weights over attributes name and address are set 0.4 and 0.6, respectively. Then, the similarity of each record in R can be calculated based on the (1), and shown in the 4th column of Figure 2. Eventually, we obtain that r_1,r_2 constitute the top-2 records.

Figure 2 Similarities of records with weight vector (0.4, 0.6).

Tuple ID	Sname	Saddr	Similarity
r_1	1.0	0.9	0.94
r_2	0.9	0.95	0.93
r_3	1.0	0.7	0.92
r_4	0.9	0.1	0.42
r_5	0.5	0.3	0.38

In general, extraction of the top-K records offers two advantages over the traditional manner of approximate searches. On one hand, it returns most similar records *without* assigning thresholds to the attributes of interest as assigning improper thresholds can cause either empty results (thresholds are too high) or too many results (thresholds are too low). On the other hand, in the traditional manner of approximate searches, records whose separated similarity scores cannot satisfy the thresholds, may have greater similarities than some returned records. Considering Example 1, if we pre-specify thresholds, 1.0 and 0.7, over attributes name and address, respectively, we can identify that similarity scores of both t_1 and t_3 are greater than the corresponding thresholds and are returned. However, in Example 1, t_2 has greater similarity score than t_3, but t_2 is not returned. Clearly, according to the definition, no other records can have greater similarities than any other record in the top-K records so that we should have a relatively high confidence that the results returned are most appropriate.

Unfortunately, extraction of top-K records over a large database by the above framework poses some challenges. If we calculate similarities of all records one by one and then extract the top-K records, the computational cost would be extremely expensive. Notice that K is usually much smaller than the size of relation. According to [11], there exist power law distributions of the similarity among entities in many domains, such as Internet topology and social networks, i.e., given a query record q, most records in the relation are *not* similar to q. Hence, it is possible and necessary to exploit the filtering techniques so that the number of records that need to be verified can be reduced to a minimum. The major contributions of this paper are as follows:

- To the best of our knowledge, we are the first group to study the efficiency of top-K selection problem for the approximate search in the context of a relation with multiple attributes.
- We propose two novel algorithms, ScanIndex and TD, to compute the top-K records. With respect to ScanIndex, in virtue of inverted indices, records whose similarity scores over each attribute are all zeros' will be free from computation. With respect to TD, we transform the top-K selection problem to the threshold based approximate string search problem, which is well studied in the literature. TD starts by computing a set of records and initializing the top-K records and thresholds over individual attributes using similarities of these records. In virtue of approximate string search approach, we can efficiently identify the next candidate record to update the top-K records and refine respective thresholds. This iteration goes on until for each record, we can determine whether it is in the top-K records or not. Comparing with ScanIndex, TD can further skip records whose similarity scores are not all zeros' but less than corresponding thresholds over individual attributes.
- We present a detailed performance evaluation of our proposed algorithms using two real life datasets which demonstrates our proposed work is both theoretically and practically sound.

The rest of the paper is organized as follows. Section 2 defines our problem and introduces preliminaries over approximate string search algorithms. Section 3 and 4 describe our proposed work. Experimental results are given in Section 5 and related work is shown in Section 6. Section 7 concludes this paper.

2 Preliminaries

2.1 Problem definition

Given a query record q and a relation R, our objective is to *efficiently* identify K records (termed as top-K records) which are most similar to q. Similarity between a record $r \in R$ and q is defined in (1) and $sim(q.a, r.a)$ in the equation is quantified by a specific similarity function, where $0 \leq sim(q.a, r.a) \leq 1$.

Given a relation R, we assume all attribute values in R are strings. Dealing with non-string values is beyond the scope of this paper. Given a record r, $r.a$ is used to denote the value of record r over attribute a. Without loss of generality, we assume the number of attributes in R equals to that of a query record q. If not, only the attributes, involved in q will be considered for each record while the rest of attributes are ignored. Let $T_{r.a}$ and $T_{q.a}$ be two sets of tokens for $r.a$ and $q.a$, respectively. Unless otherwise specified, we refer to substrings of a string as tokens in a generic sense such that they can be words, Q-grams [13], v-grams [23], etc. For example, consider attribute value r_4.name, "Mei Wang", which is shown in Figure 1. If r_4.name is tokenized as words, then $T_{r_4.\text{name}} = \{\text{"Mei"}, \text{"Wang"}\}$. If r_4.name is tokenized as 3-grams, then $T_{r_4.\text{name}} = \{\text{"Mei"}, \text{"ei"}, \text{"i W"}, \text{" Wa"}, \text{"Wan"}, \text{"ang"}\}$. Let $U_a = \cup_{\forall r \in R} T_{r.a}$ be a finite set of all tokens of all records over attribute a.

Like keyword search in Information Retrieval [32], for each token $t \in U_a$, an inverted list (denoted as L_t^a) is utilized to map t to a list of record identifiers where these records over attribute a contain t. As a result, we can obtain an inverted index I_a based on these inverted lists for all tokens of U_a. Notice that this inverted index can be pre-constructed with one sequential scan of the records in R. For reference, symbols listed in Table 1 will be used throughout the paper.

2.2 Similarity functions

Given a query record q and a record $r \in R$, similarity score between q and r over attribute a, $sim(q.a, r.a)$, can be quantified using a given similarity function. We

Table 1 Symbols and their definitions.

Symbol	Definition
R	a relation, $R = \{r_1, \ldots, r_n\}$
A	attributes of R, $A = \{a_1, \ldots, a_m\}$
$r.a$	value of record r over attribute a
$T_{r.a}$	tokens of $r.a$
$\lvert T_{r.a}\rvert$	the number of tokens for $r.a$
W	a weight vector of attributes, $W = \{\omega_1, \ldots, \omega_m\}$
q	a query record
$q.a$	value of q over attribute a
$T_{q.a}$	tokens of $q.a$
U_a	a finite universe of tokens over attribute a,$U_a = \{t_1, \ldots, t_{\lvert U_a\rvert}\}$
$w_a(t)$	weight of token t over attribute a
L_t^a	a record id list where attribute value of each record over a contains token t
$sim(q.a, r.a)$	similarity score between r and q over attribute a
$sim(q, r)$	similarity between r_i and q

Table 2 A variety of widely used similarity functions.

Names	Details
Intersect	$sim_{Intersect}(q.a, r.a) = \lvert T_{q.a} \cap T_{r.a} \rvert$
Dice	$sim_{Dice}(q.a, r.a) = \frac{2 \cdot \lvert T_{q.a} \cap T_{r.a} \rvert}{\lvert T_{q.a} \rvert + \lvert T_{r.a} \rvert}$
Jaccard	$sim_{Jaccard}(q.a, r.a) = \frac{\lvert T_{q.a} \cap T_{r.a} \rvert}{\lvert T_{q.a} \cup T_{r.a} \rvert}$
Cosine[a]	$sim_{Cosine}(q.a, r.a) = \frac{\mathbf{T_{q.a}} \cdot \mathbf{T_{r.a}}}{\lVert \mathbf{T_{q.a}} \rVert \lVert \mathbf{T_{r.a}} \rVert} = \frac{\lvert T_{q.a} \cap T_{r.a} \rvert}{\sqrt{\lvert T_{q.a} \rvert} \cdot \sqrt{\lvert T_{r.a} \rvert}}$

[a] $T_{q.a}$, $T_{r.a}$ are regarded as $\lvert U_a \rvert$-dimensional vectors. For ease of illustration, we ignore the weight of each token. Our proposed work can also be applied to the case, where each token is attached with a weight.

consider several commonly used similarity functions that are shown in Table 2. In general, these are token based similarity functions.

2.3 Approximate string search

Given a query record q and a relation R, when a user issues an approximate string search over attribute a with threshold δ_a, $\forall r \in R$, if $sim(q.a, r.a) \geq \delta_a$, then r will be returned, i.e., $\{r | sim(q.a, r.a) \geq \delta_a, r \in R\}$. For ease of illustration, we use Jaccard similarity function as default, and it can easily be extended to other similarity functions. In virtue of formula derivation of $sim_{Jaccard}(q.a, r.a)$ [30, 35], the necessary and sufficient condition of statement $sim(q.a, r.a) \geq \delta_a$ is:

$$\lvert T_{q.a} \cap T_{r.a} \rvert \geq \frac{\delta_a}{1 + \delta_a} (\lvert T_{q.a} \rvert + \lvert T_{r.a} \rvert) \tag{2}$$

According to the above inequality, for each record r, if the number of common tokens between $T_{q.a}$ and $T_{r.a}$ is less than $\frac{\delta_a}{1+\delta_a}(\lvert T_{q.a} \rvert + \lvert T_{r.a} \rvert)$, then $sim(q.a, r.a) < \delta_a$ and r can be skipped. Based on this idea, in [30], they propose an approach, HeapMerge, to identify records, whose similarity scores over a is not less than δ_a.

Let $T_{q.a} = \{t_1, t_2, \ldots, t_\lambda\}$ and $L_q^a = \{L_{t_1}^a, \ldots, L_{t_\lambda}^a\}$, where $\lambda = \lvert T_{q.a} \rvert$. Figure 3 shows an example of $T_{q.a}$ and L_q^a.

Pruning Rule 1 *For a record r, if r does not exist in any list of L_q^a over attribute a, then $sim(q.a, r.a) = 0$.*

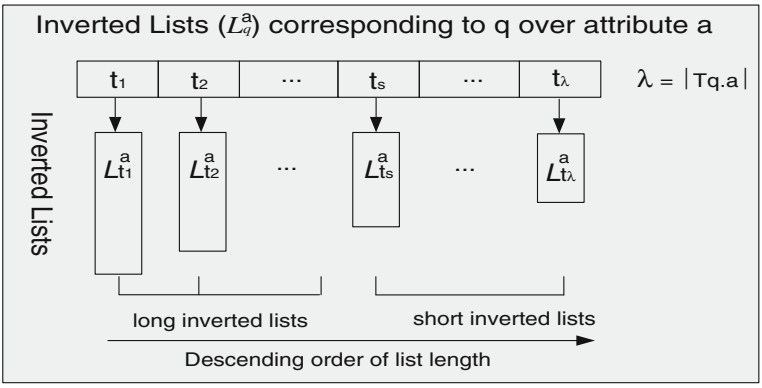


Figure 3 An example of inverted lists, which q is involved in.

Thus, based on Pruning Rule 1, we only consider these records which appear in the inverted lists L_q^a. This identification can be conducted by merging lists of L_q^a, and records with common tokens $\geq \frac{\delta_a}{1+\delta_a}(|T_{q.a}| + |T_{r.a}|)$, constitute the results. Directly merging these inverted lists is prone to suffer as $T_{q.a}$ often contains some frequent tokens that lead to large length of inverted lists. Hence, similar to the stop words in Information Retrieval [32], *HeapMerge* partitions inverted lists of L_q into two categories based on the threshold δ_a: (1) long inverted lists (corresponding to frequent tokens); and (2) short inverted lists (corresponding to non-frequent tokens).

Let $s = \lceil \frac{\delta_a}{1+\delta_a}|T_{q.a} + T_{r.a}| \rceil$ that is the right part of (2). Inverted lists in L_a^q are sorted based on the descending order of their list lengths, and $1 \leq \forall i < s$, $L_{t_i}a$ is regarded as a long inverted list, otherwise, $L_{t_i}^a$ is regarded as a short inverted list. Clearly, the number of long inverted lists is $(s-1)$.

Pruning Rule 2 *Given a query record q, for a record r, if r does not appear in any of the short inverted lists, then the maximum common items that $T_{q.a}$ and $T_{r.a}$ share cannot exceed $(s-1)$. Hence, r cannot be a candidate.*

Algorithm 1: $HeapMerge(|T_{q.a}|, L_q^a, \delta_a, |T_{r.a}|)$

```
   output: a set of records, S, whose similarity scores over a are no less than δ_a
 1 S ← ∅;   H ← empty priority queue with record identifiers;
 2 let s ← ⌈δ_a/(1+δ_a) |T_{q.a} + |T_{r.a}|⌉;
 3 for i ← s to |T_{q.a}| do
 4     push the first record identifier of list L^a_{t_i} to H;   //push the first items of the short lists into H
 5 while H is not empty do
 6     rid ← H.pop();   // obtain each record identifier of the short lists
 7     counter ← 1;
 8     while H is not empty do
 9         rid' ← H.top;
10         if rid = rid' then
11             H.pop();
12             counter + +;   // count the times that rid lies in the short lists
13         else break;
14     for each of the counter popped lists do
15         push the next identifier to H;
16     for i ← s − 1 to 1 do
17         if counter + i < T then break;
18         if rid in L_{t_i} then counter ← counter + 1 ;
19         if counter ≥ T then S ← S ∪ {rid}; break;
20 return S;
```

Based on Pruning Rule 2, we can further skip records that do not appear in any of the short inverted lists. In other words, we only scan records that appear in the short inverted lists, and when a record cannot be determined whether it is in the result or not, long inverted lists will be further searched using binary search algorithm [25]. The details of this identification are described in Algorithm 1 [30]. For the sake of brevity, we suppose that attribute values of records over attribute *a* have the same

string length, i.e., $\forall r_i, r_j \in R$, $|T_{r_i.a}| = |T_{r_j.a}|$. Coping with different lengths of strings will be illustrated in Section 4.

3 First solution: ScanIndex

Given a query record q and a relation R, the straightforward approach of extracting the top-K records, called as **ScanAll**, is to compute the similarities of records in R one by one. Using a priority queue [8], K records with greatest similarities can be identified. ScanAll is easy to implement. However, this method suffers from high computational cost as the similarity of each record needs to be computed. In order to reduce this computational cost, we propose an algorithm called ScanIndex, which is based on the inverted indexes over individual attributes.

Let $T_{q.a} = \{t_1, t_2, \ldots, t_\lambda\}$ and $L_q^a = \{L_{t_1}^a, \ldots, L_{t_\lambda}^a\}$, where $\lambda = |T_{q.a}|$. The main idea of ScanIndex is described as follows: $\forall r \in R$, over attribute a, if record identifier of r does not lie in any inverted list of L_q^a, i.e.,$T_{r.a}$ and $T_{q.a}$ share no common tokens, then $sim(q.a, r.a) = 0$; otherwise, we compute $sim(q.a, r.a)$ by scanning and merging record identifiers of L_q^a. After similarity scores of all records over individual attributes have been computed, the top-K records can be finally deserved in virtue of a priority queue.

Algorithm 2: $ScanIndex(R, L_q^a, K)$

```
   output: Top-K records: H_S
1  H_S ← ∅;
2  initialize a float array oscore with size |R|;
3  for ∀a ∈ A do
4      initialize an integer array Counter with size |R|;
5      for ∀L_t^a ∈ L_q^a do
6          for i ← 1 to |L_t^a| do
7              rid ← L_t^a.pos[i];
8              Counter[rid] ← Counter[rid] + 1;
9      for i ← 1 to |R| do
10         rid ← i;
11         if Counter[rid] > 0 then
12             compute sim(q.a, r.a) using a specified similarity function shown in Table 2;
13             oscore[rid] ← oscore[rid] + ω_a × sim(q.a, r.a);
14 for i ← 1 to |R| do
15     rid ← i;
16     if oscore[rid] > 0 then
17         if |H_S| < K then
18             H_S.push(rid);
19         else if oscore[rid] > oscore[H_S.top] then
20             H_S.pop();   H_S.push(rid);
21 return H_S;
```

3.1 Description of ScanIndex

Algorithm 2 depicts the details of ScanIndex. At first, we initialize an array *oscore* to store the overall similarity scores of all records in R (line 2). Over individual attribute

a, we sequentially scan each inverted list L_t^a in L_q^a, and count the number of common tokens sharing by each record r and q over attribute a. Array *Counter* is used to store the number of common tokens for each record (lines 4–8). After computation of the number of common tokens for all records over individual attribute a, we can compute the similarity scores of records one by one (line 12), and update their similarities correspondingly (line 13). It is worth mentioning that after computing the number of common tokens, the similarity score of each record can be calculated as the number of tokens can be pre-loaded for each record and be computed for the query at the beginning.

We use a priority queue, H_S, to maintain K records, with the greatest similarities, which we have seen. For each record r with the similarity greater then zero, if the current size of H_S is less than K, we regard r as a candidate and push it into H_S. When the current size of H_S equals K, then we need to verify whether $sim(q, r)$ is greater that the current minimum similarity of records in H_S and refine H_S properly (lines 14–20). Finally, H_S consists of K records with the greatest similarities.

3.2 Analysis of ScanIndex

Comparing with ScanAll, records that are not in any list of L_q^a, can be free from computation of their similarity scores over attribute a. Based on Algorithm 2, we conduct the space complexity and time complexity as follows:

- **Space Complexity**: In order to compute the similarity score of records over individual attribute, we use an array *Counter* to store the number of common tokens with size $|R|$. The space complexity is $O(|R|)$. In addition, we use another array *oscore* to store the overall similarities of all records with another space complexity $O(|R|)$. Finally, a priority queue H_S is maintained, with space complexity $O(K)$, for the top-K records refinement. Hence, the total space complexity of ScanIndex is $O(|R|)$, where $K \ll |R|$.
- **Time Complexity**: The time complexity of ScanIndex consists of three parts: (1) counting the number of common tokens by scanning inverted lists L_t^a in L_q^a (lines 5–8) with the time complexity $O(\sum_{a\in A}\sum_{t\in T_{q.a}}|L_t^a|)$; (2) computing the similarity score of records by scanning array *Counter* (lines 9–13) with time complexity $O(|R||A|)$; (3) deriving the final top-K records by scanning array *oscore* and maintaining a priority queue H_S (lines 14–20) with the time complexity $|R| \times logK$. To sum up, the time complexity of ScanIndex is $O(\sum_{a\in A}\sum_{t\in T_{q.a}}|L_t^a| + |R|logK + |R||A|)$.

Limitations Yet, ScanIndex still needs to scan too many records when the query contains some frequent tokens. As a result, we develop another method called Top-Down Algorithm to alleviate this problem.

4 Second solution: top-down algorithm

As described in Section 2, an important fact is that: over an attribute a, when answering an approximate string search with a given threshold, a number of records can be skipped using HeapMerge algorithm. Thus, we try to think whether we can transform the top-K selection problem to the threshold based approximate string

searches over individual attributes. In this section, we propose another interesting approach, called Top-Down algorithm (TD for short), to identify the top-K records. Without loss of generality, Jaccard (described in Table 2) is offered as the similarity function and attribute values are tokenized into a set of 3-grams.

4.1 Initialization and dynamic refinement of thresholds

TD follows the idea of bounding-pruning-refining strategy.

(1) **Bounding:** TD algorithm starts by splitting the whole dataset into a set of partitions over individual attributes. The partition phase is independent on the query. Each partition will be sequentially accessed based on a specified order which is dependent on the query. By computing similarity scores of records in first partition(s), we can initialize the top-K records (denoted as Υ_K) and identify the record (denoted as $r(K)$) in Υ_K with the minimum similarity to q. In virtue of $r(K)$, we prove that the necessary condition of a record r belonging to the final top-K records is that $\exists a \in A, sim(q.a, r.a) > sim(q, r(K))$.

(2) **Pruning:** We offer $sim(q, r(K))$ as the threshold, and issue approximate string searches using HeapMerge over the next partition. As described in Section 2, a number of record identifiers lying in the corresponding inverted indexes can be pruned. Further, as we will show later, the upper bound on similarity scores of records in the partition can be easily obtained. Hence, if we can verify that such upper bounds are not greater than $sim(q, r(K))$, then all records in the partition can be pruned.

(3) **Refining:** Over the next partition, by applying HeapMerge algorithm, records whose similarity scores exceed $sim(q, r(K))$ can be computed and considered as candidates. We update their similarities, update the top-K records, and refine the threshold $sim(q, r(K))$.

We iteratively do steps (2) and (3) until for every record, we can determine whether it is in the top-K records or not.

4.1.1 Bounding thresholds over individual attributes

Suppose we have initialized the top-K records, Υ_K, with non-zero similarities. Let $r(K)$ be the record among Υ_K with the minimum similarity.

Theorem 1 *Given a record $r \in R - \Upsilon_K$, the necessary condition that r is taken as a candidate of top-K records is $\exists a \in A$, $sim(q.a, r.a) > sim(q, r(K))$.*

Proof Suppose $\forall a \in A$, $sim(q.a, r.a) \leq sim(q, r(K))$. Based on (1), $sim(q, r) = \sum_{a \in A} \omega_a * sim(q.a, r.a) \leq sim(q, r(K)) * \sum_{a \in A} \omega_a = sim(q, r(K))$. r cannot be a candidate as $sim(q, r) \leq sim(q, r(K))$ and $r(K)$ be the record in the top-K records with the minimum similarity. Hence, in order to make r be a candidate of top-K records, there must exist an attribute $a \in A, sim(q.a, r.a) > sim(q, r(K))$. □

Based on Theorem 1, union of records whose similarity scores over any individual attribute are greater than $sim(q, r(K))$ are served as the candidates. Hence, if we can obtain the first Υ_K, then an efficient approximate string search algorithm can be applied to identify these candidates. According to Algorithm 1, the performance of

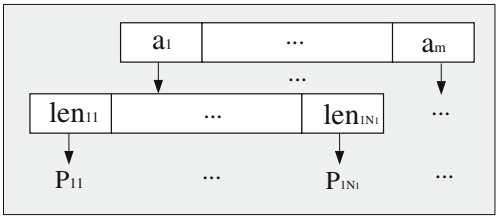


Figure 4 Data partition over individual attributes.

HeapMerge will be improved when the threshold increases. In the remainder of this section, we will illustrate how to obtain the first Υ_K.

(1) **Data Partition Over Individual Attributes:** Intuitively, we are required to select the first top-K records such that $sim(q, r(K))$ can be set as large as possible. However, obtaining the "best" first top-K records is not easy, and study on this problem may be extended to another research problem. Instead, we exploit an important observation that records similar with the query often have similar length of strings over some attributes, i.e., $\exists a \in A$, $|T_{r.a}| \approx |T_{q.a}|$ if r and q are similar. As a result, we split records, over individual attributes, into a set of partitions based on their string lengths. Formally, a partition over attribute a_i with string length len_{ij} is represented as P_{ij}, where $P_{ij} = \{r || T_{r.a_i}| = len_{ij}, r \in R\}$.

Example 2 (Partition) Given a set of records R shown in Figure 1a, R are split into seven partitions by the string lengths over individual attributes. Over attribute "Name", $len_{11} = 5, len_{12} = 6, len_{13} = 7$, and $P_{11} = \{r_2\}$, $P_{12} = \{r_1, r_3, r_4\}$, $P_{13} = \{r_5\}$.

Figure 4 shows the complete partitions of the dataset, where N_i is the number of different lengths of strings over attribute a_i of R.

(2) **Indexing Records in Each Partition:** In order to apply the approximate string search algorithm to identify candidates, we create an inverted index for each partition individually. For the sake of brevity, the inverted index that corresponds to partition P_{ij} is denoted as $I_{a_i}[j]$, and the inverted list that corresponds to token t in $I_{a_i}[j]$ is represented as $L_t^{a_i}[j]$. Given a query q, inverted lists in $I_{a_i}[j]$ that corresponds to tokens of q_{a_i} are represented as $L_q^{a_i}[j]$.

Definition 1 (Partition Similarity[2]) Given a partition P_{ij}, Partition Similarity (denoted as $sim(P_{ij}, q)$) of P_{ij} is defined as

$$sim(P_{ij}, q) = \frac{\min\{len_{ij}, |T_{q.a_i}|\}}{\max\{len_{ij}, |T_{q.a_i}|\}} \tag{3}$$

(3) **Initializing Υ_K:** As described above, in order to best improve the efficiency of approximate string search algorithm, the similarities of records in Υ_K expects to be maximized. Intuitively, this can be done by traversing these records that

[2]It is worth mentioning that using different similarity functions, definitions the partition similarity are different.

potentially pose the maximum similarity scores over individual attributes. By extending the similarity relation to partitions, we can compute the upper bound on similarity scores of all records in each partition. Definition 1 defines the partition similarity between a given partition and the query. Based on Theorem 2, $\forall r \in P_{ij}$, $sim(q.a_i, r._{a_i}) \leq sim(q, P_{ij})$. Hence $sim(q, P_{ij})$ can be taken as the upper bound on similarity scores of records in P_{ij} over a_i. In order to maximize $sim(q, r(K))$, we sort partitions over each attribute based on the descending order of their partition similarities instead of string lengths such that similarity scores of records can be potentially maximized, i.e., $sim(q, P_{ij}) \geq sim(q, P_{ij'})$ if $j \leq j'$.

Theorem 2 $\forall r \in P_{ij}$, $sim(q.a_i, r._{a_i}) \leq sim(P_{ij}, q)$.

Proof $sim(q.a_i, r._{a_i}) = \frac{T_{q.a_i} \cap T_{r.a_i}}{T_{q.a_i} \cup T_{r.a_i}} \leq \frac{\min\{T_{q.a_i}, T_{r.a_i}\}}{T_{q.a_i} \cup T_{r.a_i}} \leq \frac{\min\{T_{q.a_i}, T_{r.a_i}\}}{\max\{T_{q.a_i}, T_{r.a_i}\}} = sim(P_{ij}, q)$. □

We start by computing the similarity scores of records in P_{11} using the inverted lists $L_q^{a_1}[1]$. If the first top-K records can be derived, then thresholds over individual attributes can be bounded. Otherwise, we iteratively do the computation of records in next partitions based on the order that is described in Figure 5 until the first top-K records can be derived.

4.1.2 Pruning records using approximate string search

Given a partition P_{ij}, if partition similarity of P_{ij} is not greater than $sim(q, r(K))$, then based on Pruning Rule 3, no more records need to be traversed over attribute a_i.

Pruning Rule 3 (Partition Similarity Pruning) *Suppose we have identified the first top-K records. If* $sim(P_{ij}, q) \leq sim(q, r(K))$, *then partitions* P_{il}, $j \leq l \leq N_i$ *can be pruned.*

Proof $\forall r \in P_{ij}$, based on Theorem 2, $sim(q.a_i, r.a_i) \leq sim(P_{ij}, q) \leq sim(q, r(K))$, then r over attribute a_i cannot be the candidate. Further, $\forall r \in P_{il}$ with $N_i \geq l > j$, as $sim(q.a_i, r.a_i) \leq sim(P_{il}, q) \leq sim(P_{ij}, q) \leq sim(q, r(K))$, again, r over attribute a_i cannot be the candidate. □

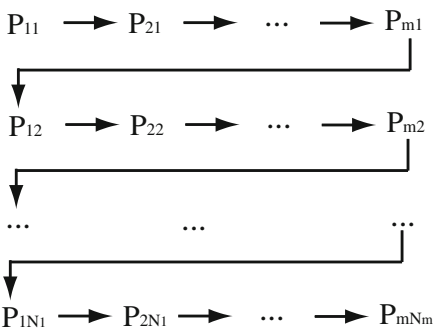


Figure 5 The order of partition traversal.

Given a partition P_{ij}, notice that $|L_q^{a_i}[j]|$ might not equal to $|T_q^a|$ as none of records in P_{ij} contain some tokens of T_q^a. Hence, we propose Pruning Rule 4. In addition, as we issue approximate string searches over individual attributes, a number of records can be further skipped.

Pruning Rule 4 (Counter Pruning) *Given a partition* P_{ij}, *if* $\frac{|L_q^{a_i}[j]|}{len_{ij}+|T_q^a|-|L_q^{a_i}[j]|} \leq sim(q, r(K))$, *then* P_{ij} *can be pruned.*

4.1.3 Refining thresholds

Over partition P_{ij}, in virtue of exploiting HeapMerge to scan inverted lists $L_q^{a_i}[j]$, we obtain a set of candidate records whose similarity scores over a_i is greater than $sim(q, r(K))$. By computing $sim(q.a, r.a)$ and refining $sim(q, r)$, if $sim(q, r) > sim(q, r(K))$, we update Υ_K, $r(K)$, and refine thresholds over individual attributes using the new value $sim(q, r(K))$.

The above pruning-refining operations will iteratively goes on until, for each record, we can determine whether it is in the top-K records or not. During the refinement of $sim(q, r(K))$, the pruning power of HeapMerge will be enhanced dynamically.

4.2 Implementation of TD

Algorithm 3 shows the details that how TD is implemented. H_S shown in line 1 maintains the top-K records dynamically. In line 2, *cand* is used to maintain candidates, and each element in *cand* poses the format $\langle rid, sscore\rangle$, where *rid* and *sscore* represent a record identifier and its similarity score over an attribute, respectively. Similar to algorithm ScanIndex that is shown in Algorithm 2, we initialize a float array *oscore* with size $|R|$ to store the overall similarities of records (line 3). As described before, in order to potentially maximize $sim(q, r(K))$ as early as possible such that the performance of HeapMerge can be best improved, we sort partitions that are involved in each attribute based on the descending order of partition similarities (line 5). At first, we traverse records lying in inverted lists $L_q^{a_i}[1]$ by setting $lPointer[i] = 1$ (line 6). Symbol $\mathcal{A}$ in line 6 is used to maintain the sequential traversal over attributes. Over attribute a_i, if all partitions have been checked, or partition similarity $sim(q, P_{ij}) \leq \theta$, then no further check on a_i is required and we remove a_i from $\mathcal{A}$ (line 10). Based on Pruning Rule 4, if $\frac{|L_q^{a_i}[j])|}{len_{ij}+|T_q^{a_i}|-|L_q^{a_i}[j]|} \leq \theta$, then no further check of partition P_{ij} is requried and then the next partition will be traversed. Otherwise, we issue an approximate string search with threshold θ[3] over inverted lists $L_q^{a_i}[j]$ to identify candidates in partition P_{ij} (line 12). Format of elements in *cand*′ is $\langle rid, sscore\rangle$ as well. Based on these candidate records in *cand*′, we update H_S and refine θ. Details of this refinement are described in Algorithm 4.

[3]Note that at the first beginning, θ is set to zero, and all records lying in the corresponding inverted lists are taken as candidates.

Algorithm 3: $TopDown(R, q, K)$

```
   output: Top-K records H_S
1  θ ← 0; H_S ← ∅;
2  initialize the candidate set cand with the element format ⟨rid, sscore⟩;
3  initialize a float array oscore with size |R|;
4  for ∀a_i ∈ A do
5      sort partitions involved in a_i based on the descending order of partition similarities;
6      lPointer[i] ← 1; 𝒜 ← 𝒜 ∪ a_i;
7  while 𝒜 is not empty do
8      for ∀a_i ∈ 𝒜 do
9          j ← lPointer[i];
10         if j > N_i || sim(q, P_ij) ≤ θ then remove a_i from 𝒜;continue;
11         if |L_q^{a_i}[j])| / (len_ij + |T_q^{a_i}| − |L_q^{a_i}[j]|) > θ then
12             cand′ ← HeapMerge(T_q^{a_i}, L_q^{a_i}[j], θ, len_ij);
13             θ ← RefineTopK(a_i, oscore, cand, cand′, H_S);
14 VerifyTopK(R, cand, H_S);
15 return H_S;
```

Algorithm 4: $RefineTopK(a_i, oscore, cand, cand', H_S)$

```
1  for ∀rid ∈ cand′ do
2      if sim(q.a_i, rid.a_i) ≤ oscore[H_S.top] then continue;
3      cand ← cand ∪ {⟨rid, sim(q.a_i, rid.a_i)⟩};
4      oscore[rid] ← oscore[rid] + ω_i * sim(q.a_i, rid.a_i);
5      if |H_S| < K then
6          if rid ∈ H_S then re-adjust H_S;
7          else H_S.push(rid);
8      else if oscore[rid] > oscore[H_S.top] then
9          if rid ∈ H_S then re-adjust H_S;
10         else H_S.pop(); H_S.push(rid);
11 return oscore[H_S.top];
```

Algorithm 5: $VerifyTopK(R, cand, H_S)$

```
1  for ∀e ∈ cand do
2      rid ← e.rid; sscore ← e.sscore;
3      if sscore > H_S then
4          compute sim(q, rid) and update H_S;
```

Due to the refinement of $sim(q, r(K))$ after traversing the inverted index, records that originally lie in the candidate set *cand*′ may not be regarded as candidate any further. Hence, for each record r in *cand*′, if we can check that $sim(q.a_i, r.a_i)$ does not exceed $sim(q, r(K))$, then r can be pruned (line 2), otherwise, r is considered as a candidate. We add pair $\langle rid, sim(q.a_i, r.a_i)\rangle$ to *cand* (line 3), refine the similarity $sim(q, r)$ by $sim(q.a_i, r.a_i)$ (line 4), and update the top-K records (lines 5–10). After all records in *cand*′ have been checked, we return $sim(q, r(K))$ as the threshold.

We iteratively do this pruning and refining until all partitions are either pruned or traversed. After completing the above iteration, we need to check and verify records in the candidate set *cand*. Details of this verification is described in Algorithm 5. As described above, due to the refinement of $sim(q, r(K))$, records that lie in *cand*

may not be considered as a new candidate. If we can check that $sim(q.a_i, r.a_i) \leq sim(q, r(K))$, then r can be pruned (line 3); otherwise, we compute similarity scores over individual attributes, and derive its similarity, and refine the top-K records (line 5). Although for some records, repetitive computation of their similarities may happen when their similarity scores exceed $sim(q, r(K))$ over multiple attributes, based on our experiments, the number of such records is extremely small.

4.3 Analysis of TD

Based on Algorithm 3–5, we conduct the space complexity and time complexity as follows:

- **Space Complexity:** By performing approximate string searches over inverted lists $L_q^{a_i}[j]$, we use $P_{ij}.cand$ to store the candidate set (line 12 of Algorithm 3). The space complexity is $O(|P_{ij}.cand|)$. In addition, we use another array *oscore* to store the similarities of all records with another space complexity $O(|R|)$ (line 1 of Algorithm 3). Further, the candidate set *cand* maintain records whose similarity scores exceed $sim(q, r(K))$ over some attributes at each iteration (line 2). The space complexity is $O(|cand|)$. Finally, a priority queue H_S is used to maintain the top-K records with space complexity $O(K)$ (line 1). Hence, the total space complexity of TD is $O(\max\{|P_{ij}.cand|\} + |R| + |cand|\})$. In general, $\max\{|P_{ij}.cand|\} \ll |R|$ and $|cand| \ll |R|$ and the space complexity can be reduced to $O(|R|)$.
- **Time Complexity:** The time complexity of TD consists of the following four parts:(1) time cost to compute the candidate set, $P_{ij}.cand$, over each partition P_{ij}. The time complexity of this part depends on the number (denoted as $L_q^{a_i}[j]$.num) of record identifiers of $L_q^{a_i}[j]$ that have been accessed using Heap-Merge (line 12 of Algorithm 3); (2) time cost to traverse records in $P_{ij}.cand$ in order to refine top-K records (line 13 of Algorithm 3). The time complexity is $O(\sum_{P_{ij}} O(|P_{ij}.cand|))$; (3) time cost to check records in *cand* with time complexity $O(|cand|)$; (4) time cost of verification of records, i.e., computation of the $sim(q, r)$ if $\forall r \in R$, $\exists a_i \in A$, $sim(q.a, r.a) > sim(q, r(K))$. Generally, as the number of verified records is quite small, the time cost of this verification can normally be ignored. In addition, $|cand| \leq \sum_{P_{ij}} |P_{ij}.cand| \leq \sum_{P_{ij}} L_q^{a_i}[j]$.num. Hence, the time complexity of TD is $O(\sum_{P_{ij}} L_q^{a_i}[j]$.num).

5 Experimental evaluation

5.1 Experiment setup

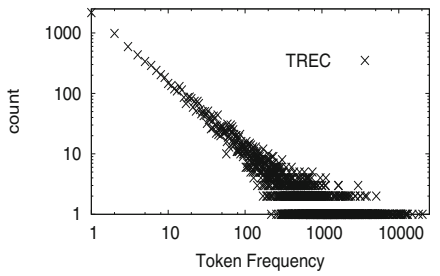
All experiments are conducted on a PC with Intel 1.66 GHz dual core CPU and 2GB memory under Windows XP operating system. All programs are implemented using JDK 1.6. We implement the following algorithms ScanAll, ScanIndex and TD as described in this paper. We use the following two real datasets for the evaluation:

- **TREC**[4] records the Broker and Sales licensees in the state of Texas. There are 145,960 records in this file. We extract six attributes which are bname (name of

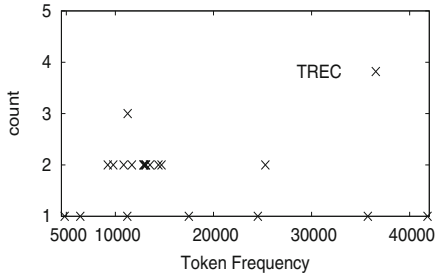
[4] http://www.trec.state.tx.us/

Dataset	A	|R|	|U|	Max	Min	Mean	S.D	size (MB)
TREC	a_1	145960	11312	25474	1	225	859	31.8
	a_2	81750	7530	20127	1	204	824	
	a_3	145943	9837	34191	1	246	958	
	a_4	145956	3151	38464	1	269	1548	
	a_5	145960	985	23678	1	445	1316	
	a_6	145960	37	14384	11638	13033	1910	
DBLP	a_1	1,749,097	16,548	201,088	1	2589	$6.4*10^7$	417
	a_2	1,768,679	44,254	844,057	1	1506	$2.1*10^8$	

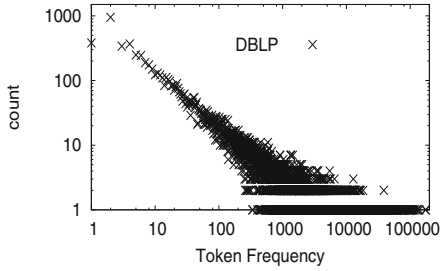
(a) Dataset Statistics



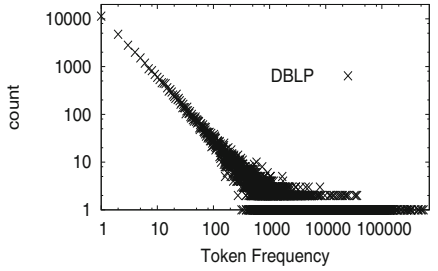
(b) Token Distribution of TREC, Attribute 1



(c) Token Distribution of TREC, Attribute 6



(d) Token Distribution of DBLP, Attribute 1



(e) Token Distribution of DBLP, Attribute 2

Figure 6 Statistics and distributions of datasets.

business), sname (name of sponsor), saddress (street address), city, zipcode, lexp (license expiration date) from this file.

– **DBLP**[5] is a snapshot of the bibliography records. We download the dataset with 1,772,301 records, and extract author names, concatenated into a single string, and the title of each publication.

Each attribute value of the dataset is tokenized as a set of 3-grams. The statistics and distribution information of the above two datasets is shown in Figure 6. Some important statistics about the datasets are listed in Figure 6a. Over the 3rd column,

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

we count the number of non-null records over each attribute. *Max*, *Min* are the maximum and minimum length of all inverted lists over each attribute. *Mean*, *S.D* are the average and standard deviation for the length of all inverted lists over the corresponding attribute. The sizes of inverted indices over individual attributes are shown in the last column of Figure 6a. For comparison, sizes of original dataset are 17.9 and 119 MB, respectively. The reason why the inverted index size does not increase too many is that we compact all inverted lists into a single file. With respect to ScanIndex, for each token t_i in the inverted index I_a, we maintain a data structure $\langle t_i, offset, length\rangle$, where *offset* pointers to the start position of inverted list $L^a_{t_i}$ of this file, and *length* represents the number of record identifiers in $L^a_{t_i}$. Figure 7 shows the details. Hence, when a query comes, we can efficiently obtain the tokens of q from the gram map, read corresponding inverted lists from the file that was pre-loaded in the memory, and then answer the top-K selection query. The case is similar for TD. With respect to TD, we also pre-load the whole dataset into the memory.

In Figure 6b and c, we show the distribution of the token frequency over attributes 1 and 6 of TREC dataset. Token frequency in x-axis represents the length of an inverted list, and count in y-axis represents the number of tokens with this frequency. From Figure 6b, we can find that the count varies inversely with the token frequency. We omit the distribution of the token frequency over attributes 2–5 as they follow similar distribution as attribute 1. Over attribute 6, there only exist 37 tokens and the frequency of each token is rather high as shown in Figure 6c. In Figure 6d and e, we show the distribution of the token frequency over attributes 1 and 2 of DBLP, which follows similar distribution with attribute 1 of TREC.

We generalize our main algorithms to the similarity functions including Jaccard and cosine. Unless otherwise specified, we set $|A| = 6$ for TREC, $|A| = 2$ for DBLP, and $\omega_i = \frac{1}{|A|}$ for each attribute by default. We randomly extract 20 records from each dataset as the queries and employ them to issue top-K selection queries. All experimental results shown in the following are the averaged values. As illustrated in the above two sections, the performance of ScanIndex and TD mainly depends on the total number of record identifiers that are accessed in corresponding inverted lists. Hence, we also show the total number of accessed record identifiers in each experiment. For TD, we also show the number of records that need to be verified, i.e., $|\{r|r \in R, \exists a \in A, sim(q.a, r.a) > sim(q, r(K))\}|$.

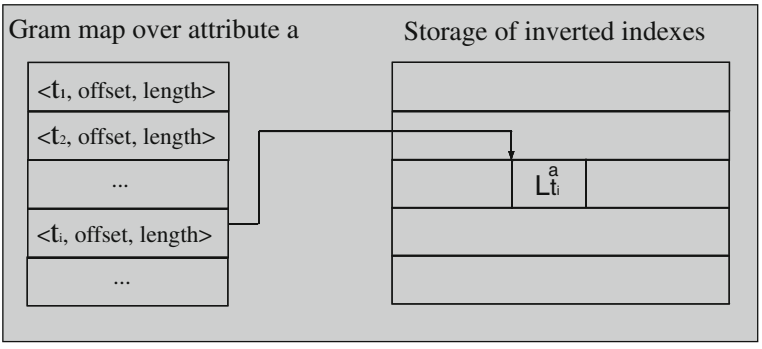


Figure 7 Storage of inverted indexes.

5.2 Overall performance evaluation of ScanIndex and TD

5.2.1 Effect of K

We evaluate the performance of identifying the top-K records on the above two datasets using Jaccard and Cosine similarity functions, where $K \in \{1, 2, 4, 6, 8, 10\}$.

Figure 8a–d show the responding time of performing top-K selection using Jaccard and Cosine, respectively. As ScanAll performs much worse than TD and ScanIndex, we do not demonstrate the results of ScanAll in these figures for the sake of clarity.

From Figure 8, we can find that both ScanIndex and TD are efficient, and TD always performs better than ScanIndex. By scanning each record identifier over inverted lists that a search query is involved in, ScanIndex counts the common tokens that are shared by each record and the query over individual attributes, and then compute the similarity scores. Although there might exist some frequent tokens that lead to long lengths of inverted lists, ScanIndex can compute similarity scores of all records only by scanning inverted lists once. In addition, when K increases, the performance almost keeps the same since time cost of ScanIndex depends on the number of record identifiers in inverted lists, but independent on K. Instead of sequentially scanning record identifiers in the long inverted lists that is the performance bottleneck of ScanIndex, TD initializes thresholds over individual attributes by carefully selecting a set of records in the dataset. After applying an approximate string search approach, HeapMerge, a number of record identifiers in the long inverted lists can be skipped while the threshold can dynamically improved. In addition, as the dataset is split into a set of partitions, Pruning Rules 3 and 4 can be applied such that more number of records can further be skipped. From the

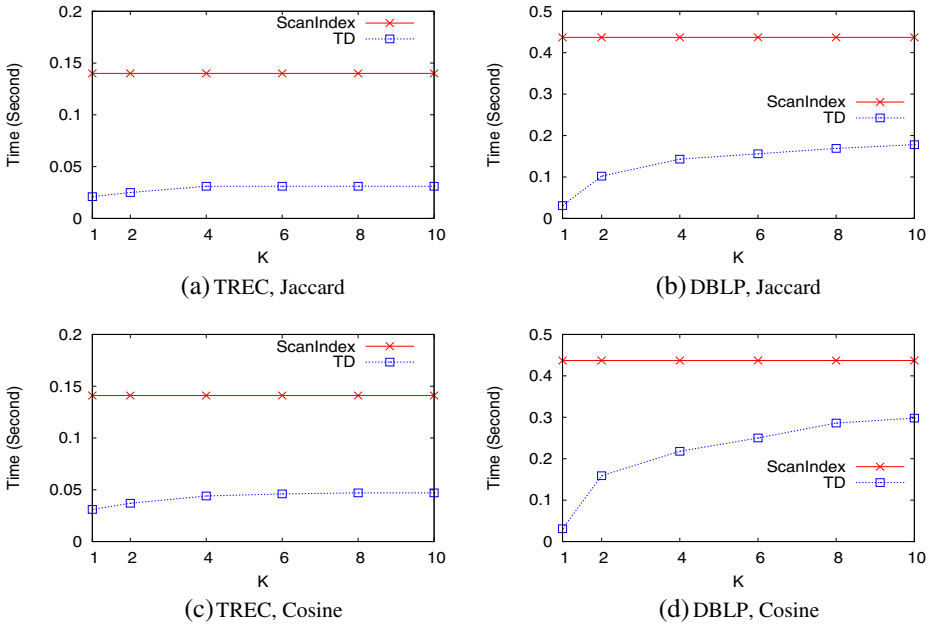


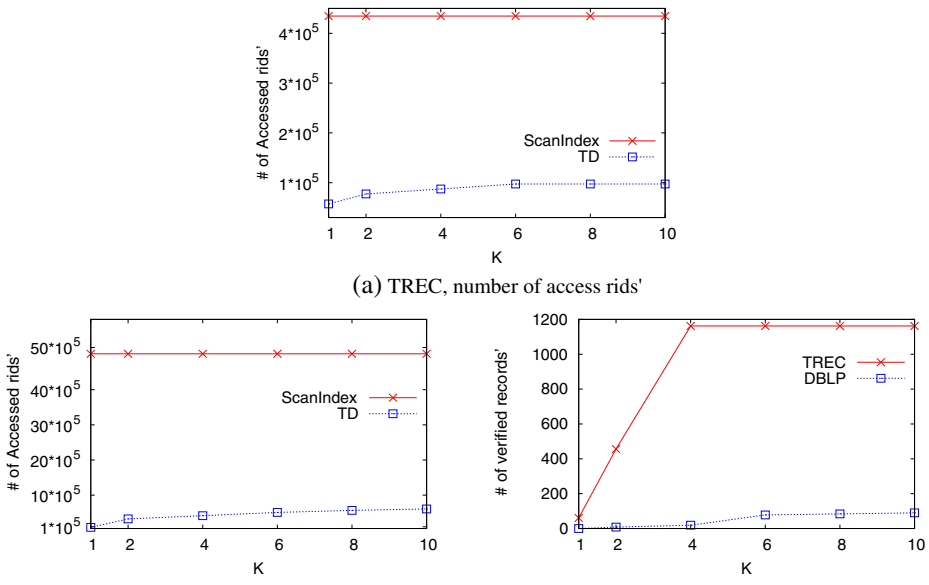
Figure 8 Responding time of varying K.

figure, we can see that the performance of TD drops when K increases. As described in Section 4, the performance of HeapMerge will be improved when the threshold increases. However, when K increases, the threshold will decrease correspondingly as we scanning the same partitions and inverted lists.

In order to further investigate the reason why TD performs better than ScanIndex, we demonstrate the number of record identifiers that are accessed during the scanning of inverted lists. As the experimental results of using Jaccard and Cosine are similar, we only show the results of using Jaccard. From Figure 9a and b, we can find that the number of accessed record identifiers in ScanIndex is much larger than that in TD. This finding verifies the effectiveness of our pruning rules. Figure 9c shows the number of records that need to be verified in TD. Interestingly, although the cardinality of DBLP is mucher larger than that of TREC, the number of verified records in DBLP is much small. By conducting analysis on two datasets, we find that in TREC, there exist a large number of duplicated values over some attributes, e.g., attributes *city* and *zipcode*. This will lead to a large size of verified records as the similarity scores of them over these attributes probably exceed the thresholds. In order to speed up the computation of similarities for verified records, we put $T_{q.a}$ in a hash over each attribute a.

5.2.2 Effect of the weight vector

We measure the performance of identifying the top-K selection on DBLP dataset using Jaccard where $K = 6$, and $\omega \in \{\{0.1, 0.9\}, \{0.2, 0.8\}, \{0.3, 0.7\}, \{0.4, 0.6\}, \{0.5, 0.5\}, \{0.6, 0.4\}, \{0.7, 0.3\}, \{0.8, 0.2\}, \{0.9, 0.1\}\}$. We do not report the results of TREC or Cosine similarity function as the findings are similar.



(a) TREC, number of access rids'

(b) DBLP, number of accessed rids'

(c) TD, number of verified records

Figure 9 Access number of varying K.

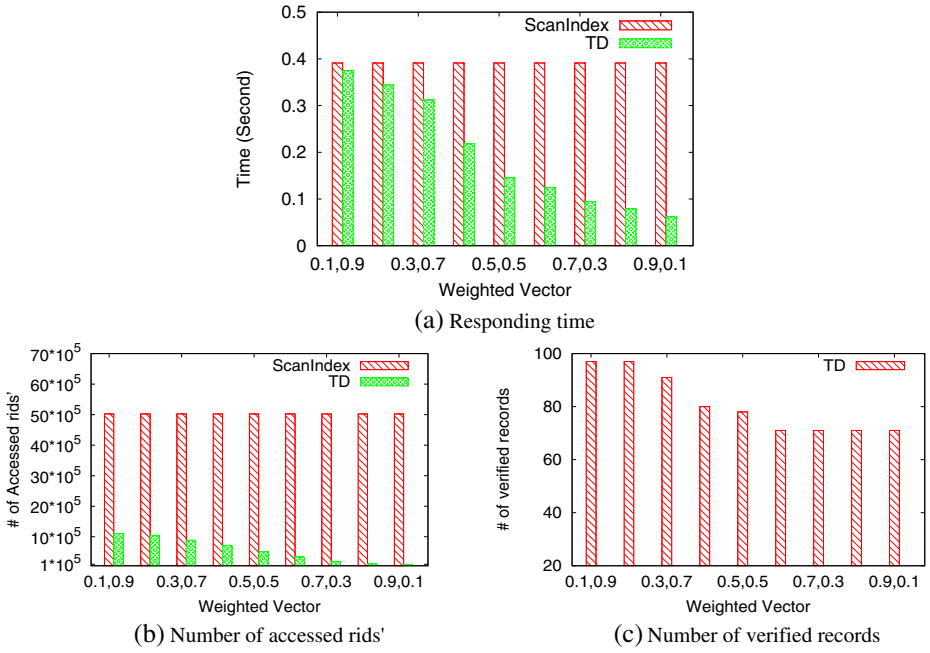
Figure 10a shows the Responding time of using ScanIndex and TD when the weight vector varies. As ScanAll performs much worse than ScanIndex and TD, we do not report it in the following experiments. Still, TD performs better than ScanIndex. When the weight vector varies, the performance of ScanIndex almost keeps the same. Interestingly, the performance of TD improves when the weight of attribute a_1 (name of authors) increases. This is because, given a query q, there could often exist a number of similar records in R over attribute *name* while it is not over attribute *title*. By increasing ω_1, the threshold $sim(q, r(K))$ can be refined such that the efficiency of TD is improved.

This finding can be verified in Figure 10b. The number of accessed record identifiers keeps the same for ScanIndex, and is reduced for TD when the weight of a_1 increases. Comparing with ScanIndex, TD reduces a large number of accessed records. The number of records that need to be verified is shown in Figure 10c. Comparing with the number of accessed record identifiers, this number is extremely small no matter the weight vector varies.

5.2.3 Effect of attributes

We investigate the performance of identifying the top-K records where $K = 6, |A| \in \{2, 3, 4, 5, 6\}$. We report the results on TREC using Jaccard similarity function.

From Figure 11a, we can find that TD performs better than ScanIndex. When the number of attributes increases, the Responding time for ScanIndex increases almost linearly. This is because, when new attributes are added, record identifiers that lies



(a) Responding time

(b) Number of accessed rids'

(c) Number of verified records

Figure 10 Effect of varying weight vector.

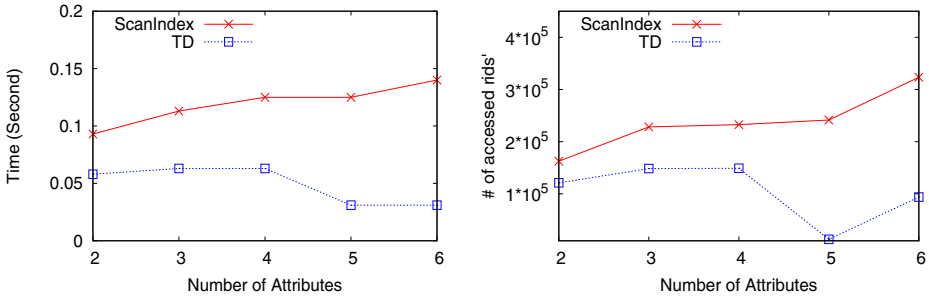
in the new inverted lists needs to be traversed additionally, and the number of record identifiers in the original inverted lists keeps the same.

Intuitively, when the number of attributes increases, the performance of TD should drop as new inverted lists need to be traversed. However, from From Figure 11a, we find that the responding time nearly keeps the same when $|A| = 4$ and more surprisingly, the Responding time decreases when $|A| = 5$. Recall that the performance of TD depends on the number of record identifiers that need to be accessed in the corresponding inverted lists. When a new attribute is added, although a number of new inverted lists are introduced, the number of record identifiers that are accessed in the original inverted lists might be reduced since the thresholds can be improved. This assertion could be verified by considering attribute a_5 (*zipcode*). As a_4 (*city*) has the positive correlation with a_5 (*zipcode*) and there exist a number of duplicated values over attributes a_4 and a_5, respectively, the thresholds can be improved a lot accordingly. To make clear comprehension of this point, we conduct the experiments on identifying the number of accessed record identifiers for both ScanIndex and TD as shown in Figure 11b. Clearly, we find that the number of accessed record identifiers of TD is much smaller than that of TD, and the relationship between the number of accessed record identifiers and the number of attributes is in accordance with that between the Responding time and the number of attributes.

5.2.4 Scalability evaluation

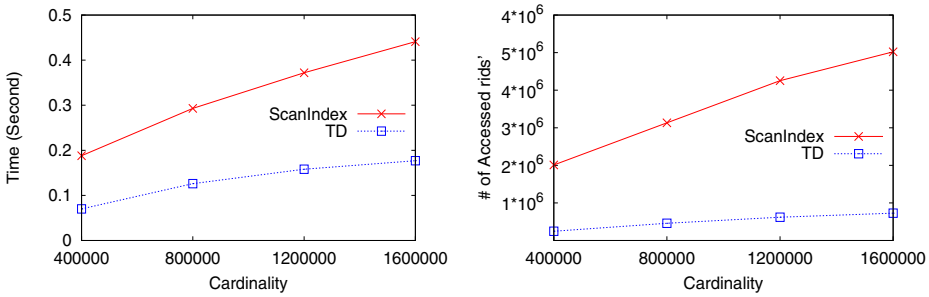
We investigate the scalability of using ScanIndex and TD to identify the top-K records on different subsets of DBLP, where $K = 6, |A| = 2, |R| \in \{400{,}000, 800{,}000, 1{,}200{,}000, 1{,}600{,}000\}$. For each dataset R, we randomly select $|R|$ records from DBLP dataset as R. By randomly extracting 20 records from different datasets as queries, we report the experimental results by using Jaccard similarity function.

From Figure 12a, we can find that the Responding time for ScanIndex increases linearly when the cardinality of the datasets varies. The reason is that when the cardinality of the datasets varies, the number of records in the inverted lists increases as well. With respect to TD, when the cardinality of the dataset varies, the responding time increases, but the incremental rate decreases. Recall that in TD, we split the dataset into a set of partitions based on their string lengths over individual attributes.



(a) Responding time (b) Number of accessed rids'

Figure 11 Effect of varying attribute.



(a) Responding time (b) Number of accessed rids'

Figure 12 scalability evaluation.

When the cardinality of the datasets increases, although the number of the records in the inverted lists increases, the number of similar records over individual attributes increases as well. Hence, the threshold $sim(q, r(K))$ can be refined earlier, which can improve the performance of TD. For the better understanding, we show the number of accessed record identifiers in Figure 12b.

6 Related work

To the best of our knowledge, this is the first study on the efficiency of top-K selection problem for the approximate search in the context of a relation with multiple attributes. Our study is related to the previous work on the approximate string search problem and top-K queries in relational databases.

6.1 Approximate string search

For a given collection of strings, an approximate string search is to identify those strings in this collection that are similar to a given search string. A variety of methods have been proposed to improve the efficiency of answering the approximate string queries. In essence, the objective of these methods is to identify these similar strings by scanning as less number of records as possible. In [1, 5, 13, 21, 23, 24, 34], the main approaches are based on the inverted indices and a variety of effective filtering techniques. In [1, 5, 24], they focus on how to skip strings as many as possible during the merging of inverted lists. In [24], they further propose an effective filter tree that integrates various filtering techniques together to skip as many records as possible. Different from the Q-gram, where the value of Q is pre-assigned, in [23], they propose a new technique called V-gram to select high quality grams with variable lengths such that the lengths of inverted lists can be decreased. As a result, the efficiency can be improved. In [12–14, 21], they study how to integrate these approaches into DBMS in a declarsive way without modifying the engine of DBMS. A more thorough investigation on approximate string searches can be found in [29].

Several existing work studies the approximate search problem in the relational databases [2, 6, 17, 19, 21], which returns records whose similarity scores over individual attributes exceeds given thresholds. In [23, 24, 31, 33–35], they concatenate

records and queries into strings, and issue approximate string queries over these strings to find similar records. As there is no single similarity function can always outperform the others. In [4], they evaluate the performances of some of the most widely used similarity functions upon a benchmark which contains some errors.

6.2 Top-K queries

The pioneering studies on top-K query processing over relational databases were conducted by Fagin [9, 10]. They propose how to efficiently extract top-K records by scanning the number of records as small as possible. Variations of Fagin's work have been explored, including keyword search in relational databases [27], top-K queries over XML DBMS [20], top-K queries in a distributed environment [3], etc. A thorough study on top-K queries can be found in the survey [18].

Similar to Fagin's work, our objective is to identify K records that are most similar to the query record. However, comparing with Fagin's work, where values over each attribute are pre-sorted, similarity scores of the records over each attribute are unknown aforehand. A straightforward idea is that we first identify top-K records over each individual attribute, and then extract the overall top-K records among them. However, this method suffers two limitations. On one hand, it is not trivial to identify top-K records over individual attributes. On the other hand, records that are supposed to be in the top-K records are not included. In order to tackle these limitations, the number of top records (rather than K) over individual attributes needs to be adjusted dynamically during the computation until there are no records that can have similarities greater than any of the overall top-K records. However, this method suffers high computational cost.

In [13, 35], they study the top-K selection problem but with single attribute, which is different from us. Guha et al. propose a voting theory model to obtain the top-K selection [15]. Unfortunately, the computational cost to obtain the top-K selection is $O(n^3)$ [28], which may not be practical for online situation. In [21], they propose a weighted vector model to calculate the similarity between a record in a relation and a query record. Yet, they do not concentrate on the top-K selection but concentrate on how to exploit a threshold over each attribute to obtain a candidate set, which is different from our motivation. It is worth mentioning that the problem of entity extraction in temporal databases [26] is similar to ours, however, their proposed approach can only be applied to temporal datasets. While handing top-K entity extraction in relational datasets, their proposed approach will be changed to ScanAll approach.

7 Conclusion

In this paper, we study the problem of efficiently identifying top-K records that are most similar to the search query. We propose two main approaches, ScanIndex and TD, that identify the top-K records without computing similarity scores of all records. In ScanIndex, for those records that appear in the inverted lists, we calculate their similarity scores by scanning and counting record identifiers of the inverted lists over individual attributes. The performance bottleneck of ScanIndex lies in long inverted lists. Comparing with ScanIndex, by splitting the whole dataset into a set

of partitions, TD can skip a large number of record identifiers in partitions that need to be accessed in ScanIndex. The superiority of our proposed algorithms are demonstrated by extensive experiments on two real datasets under a wide range of parameter settings. In general, we find that the performance of both ScanIndex and TD is mainly affected by scale, the number of attributes, the distribution of the datasets. ScanIndex is independent on the value of K, the weight vector, and the similarity functions. However, these parameters can affect the performance the TD.

Acknowledgements This work is partially supported by National Natural Science Foundation of China under Grant 60873017 and The "HGJ" Project 2010ZX0 1042-002-002-03.

References

1. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proc. VLDB, pp. 918–929 (2006)
2. Benjelloun, O., Garcia-Molina, H., Kawai, H., Larson, T.E., Menestrina, D., Su, Q., Thavisomboon, S., Widom, J.: Generic entity resolution in the SERF project. IEEE Data Eng. Bull. **29**(2), 13–20 (2006)
3. Cao, P., Wang, Z.: Efficient top-K query calculation in distributed networks. In: Proc. PODC, pp. 206–215 (2007)
4. Chandel, A., Hassanzadeh, O., Koudas, N., Sadoghi, M., Srivastava, D.: Benchmarking declarative approximate selection predicates. In: SIGMOD, pp. 353–364 (2007)
5. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: Proc. ICDE, p. 5 (2006)
6. Chaudhuri, S., Chen, B.-C., Ganti, V., Kaushik, R.: Example-driven design of efficient record matching queries. In: Proc. VLDB, pp.327–338 (2007)
7. Cohen, W.W.: Integration of heterogeneous databases without common domains using queries based on textual similarity. In: SIGMOD, pp. 201–212 (1998)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press/McGraw-Hill, Cambridge/New York, pp. 802–803 (2001)
9. Fagin, R.: Combining fuzzy information from multiple. J. Comput. Syst. Sci. **58**(1), 216–226 (1996)
10. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proc. PODS, pp. 102–113 (2001)
11. Faloutsos, M., Faloutsos, P. Faloutsos, C.: On power-law relationships of the Internet topology. In: Proc. SIGCOMM, pp. 251–262 (1999)
12. Galhardas, H., Florescu, D., Shasha, D., Simon, E., Saita, C.-A.: Declarative data cleaning: language, model, and algorithms. In: Proc. VLDB, pp. 371–380 (2001)
13. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: Proc. VLDB, pp. 491–500 (2001)
14. Gravano, L., Ipeirotis, P.G., Koudas, N., Srivastava, D.: Text joins in an RDBMS for web data integration. In: Proc. WWW, pp. 90–101 (2003)
15. Guha, S., Koudas, N., Marathe, A., Srivastava, D.: Merging the results of approximate match operations. In: Proc. VLDB, pp. 636–647 (2004)
16. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press, Cambridge (1997)
17. Hernández, M.A., Stolfo, S.J.: Real-world data is dirty: data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery **2**(1), 9–37 (1998)
18. Ilyas, I.F., Beskales, G., Soliman, M.A.: Survey of top-k query processing techniques in relational database systems. In: ACM Computing Surveys (2008)
19. Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In: Proc. WWW, pp. 371–380 (2009)
20. Kaushik, R., Krishnamurthy, R., Naughton, J.F., Ramakrishnan, R.: On the integration of structure indexes and inverted lists. In: Proc. SIGMOD, pp. 779–790 (2004)
21. Koudas, N., Marathe, A., Srivastava, D.: Flexible string matching against large databases in practice. In: Proc. VLDB, pp. 1078–1086 (2004)

22. Koudas, N., Sarawagi, S., Srivastava, D.: Record linkage: similarity measures and algorithms. In: Proc. SIGMOD, pp. 802–803 (2006)
23. Li, C., Wang, B., Yang, K.: VGRAM: improving performance of approximate queries on string collections using variable-length grams. In: Proc. VLDB, pp. 303–314 (2007)
24. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: Proc. ICDE, pp. 257–266 (2008)
25. Lu, W., Rong, C., Chen, J., Du, X., Fung, G., Zhou, X.: Efficient common item extraction from multiple sorted lists. In: Proc. Apweb (2010)
26. Lu, W., Fung, G.P., Du, X., Zhou, X., Chen, L., Deng, K.: Approximate entity extraction in temporal databases. World Wide Web J. **14**(2), 157–186 (2011)
27. Luo, Y., Lin, X., Wang, W., Zhou, X.: SPARK: Top-K keyword query in relational databases. In: Proc. SIGMOD, pp. 115–126 (2007)
28. Munkres, J.: Algorithms for the assignment and transportation problems. J. Soc. Ind. Appl. Math. **5**(1), 32–38 (1957)
29. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **3**, 31–88 (2001)
30. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD, pp. 743–754 (2004)
31. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: Proc. SIGMOD, pp. 759–770 (2009)
32. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, New York (1999)
33. Xiao, C., Wang, W., Lin, X.: Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
34. Xiao, C., Wang, W., Lin, X., Yu, J.X.: Efficient similarity joins for near duplicate detection. In: Proc. WWW, pp. 131–140 (2008)
35. Xiao, C., Wang, W., Lin, K., Shang, H.: Top-k set similarity joins. In: Proc. ICDE, pp. 916–927 (2009)