Top-k Set Similarity Joins

Chuan Xiao Wei Wang Xuemin Lin Haichuan Shang

The University of New South Wales & NICTA
{chuanx, weiw, lxue, shangh}@cse.unsw.edu.au

***Abstract*— Similarity join is a useful primitive operation underlying many applications, such as near duplicate Web page detection, data integration, and pattern recognition. Traditional similarity joins require a user to specify a similarity threshold. In this paper, we study a variant of the similarity join, termed top-k set similarity join. It returns the top-k pairs of records ranked by their similarities, thus eliminating the guess work users have to perform when the similarity threshold is unknown before hand. An algorithm, topk-join, is proposed to answer top-k similarity join efficiently. It is based on the prefix filtering principle and employs tight upper bounding of similarity values of unseen pairs. Experimental results demonstrate the efficiency of the proposed algorithm on large-scale real datasets.**

I. Introduction

Given a similarity function, a similarity join between two sets of records returns pairs of records from two sets such that their similarities are no less than a given threshold. Similar joins have a wide range of applications, including near duplicate Web page detection [1], data integration [2], record linkage [3], and data mining [4]. Consequently, there has been much interest in developing efficient algorithms for this fundamental operation.

However, the traditional form of the similarity join operation requires a user to input a similarity threshold. In many application scenarios, this threshold is not known before hand and is likely to vary according to datasets and application scenarios. An appealing alternative is to compute the most similar k pairs of records, ordered decreasingly by their similarity values. We call this top-k similarity join.

Top-k similarity join has several advantages over the traditional similarity join. Firstly, it computes most similar record pairs without the need to specify a similarity threshold. Without this top-k similarity join, users have to experiment with different threshold values, which usually leads to empty results (if the threshold chosen is too high) or a long running time and too many results (if the threshold is too low). Secondly, it supports interactive near duplicate detection applications, where users are presented with top-k most similar record pairs progressively. Thirdly, it produces most meaningful results where users perform similarity join under certain resource or time constraints. The execution of the top-k similarity join can be stopped at any time and it is guaranteed that the set of record pairs output by the algorithm have higher or equal similarity values than any of the unseen record pairs.

Top-k similarity join poses several challenges to designing efficient query processing algorithms. Although many similarity join algorithms have been proposed [5], [4], [6], [7], their efficiency heavily depends on a constant, given similarity threshold, and hence they cannot be directly applied to the top-k similarity join problem. On the other hand, the top-k similarity join problem in vector spaces with Euclidean distance functions was studied in spatial databases [8], [9], [10]. However, their algorithms are specially tailored for Euclidean distance functions and not applicable to similarity (or distance) functions such as Jaccard and cosine similarities.

In this paper, we consider the case where records are sets and similarity functions are Jaccard, cosine, and dice similarity similarities. We call this problem top-k (set) similarity joins. We propose an efficient algorithm, topk-join, that process top-k similarity joins efficiently. A naïve algorithm is to compute similarity values for all possible record pairs and then select the top k pairs. We propose an efficient algorithm that drastically reduces the number of record pairs whose similarities need to be computed. The basic idea is to carefully exploit the prefix filtering principle [5] and upper bound the similarity values of unseen pairs. Several non-trivial optimizations are also proposed to reduce the memory footprint and to achieve a tighter upper bound of the similarity scores of unseen pairs. We experimentally study the proposed method and demonstrate its efficiency for a wide range of parameter settings on real datasets.

The contributions of this paper can be summarized as follows

- We propose a novel algorithm to answer top-k similarity join queries. Unlike traditional similarity joins algorithms, the proposed algorithm can progressively compute join results without need of a prior given similarity threshold.
- We develop several new pruning and optimization techniques for top-k similarity joins by exploiting the monotonicity of maximum possible scores of unseen pairs, and the monotonicity of k-th largest similarity values seen so far. These optimizations are integrated into the basic algorithm and lead to improved upper bound estimates as well as space and time efficiency.
- The experimental results show that the new algorithm outperforms alternative algorithms in most cases and is applicable to the interactive application scenarios.

The rest of the paper is organized as follows: Section II introduces preliminaries and background on traditional similarity join algorithms. Section III introduces our basic top-k similarity join algorithm. Several optimizations of the algorithm are presented in Section IV. We discuss implementation details

and extensions to other similarity functions in Section V and VI. Experimental results are given in Section VII and related work appears in Section VIII. Section IX concludes the paper.

II. Preliminaries

A. Problem Definition

Similarity joins take two sets of record and return all pairs of records whose similarities are above a given threshold. We consider a record as a *set* of tokens taken from a finite universe $\mathcal{U} = \{ w_1, w_2, \ldots, w_{|\mathcal{U}|} \}$. A similarity function, $sim(\cdot,\cdot)$, returns a similarity value in $[0,1]$ for two records. Given two sets of records and a threshold t, a similarity join returns all pairs of records from each set, such that their similarities are no smaller than t, i.e., $\{ \langle r, s\rangle \mid sim(r,s) \geq t, r \in R, s \in S \}$. In this paper, we study a variant of the similarity join problem, termed top-k similarity join. Given two sets of records, a top-k similarity join returns k pairs of records from each set, such that their similarities are the highest among all possible pairs. For the ease of exposition, we will focus on self-join case in this paper.

We denote the *size* of a record x as $|x|$, which is the number of tokens in x. We denote the document frequency of a token w as $df(w)$, which is the number of records that contain the token. The *inverse document frequency* of a token w, $idf(w)$, is defined as $1/df(w)$. Intuitively, tokens with high idf values are rare tokens in the collection. We *canonicalize* a record by sorting its tokens according to a global ordering $\mathcal{O}$ defined on $\mathcal{U}$. An *inverse document frequency ordering* $\mathcal{O}_{idf}$ arranges tokens in $\mathcal{U}$ by the decreasing order of tokens' idf values. A record x can also be represented as a $|U|$-dimensional vector, $\mathbf{x}$, where $x_i = 1$ if $w_i \in x$ and $x_i = 0$ otherwise.

We consider several commonly used similarity functions for sets and vectors:

- *Jaccard similarity* is defined as $J(x,y) = \frac{|x \cap y|}{|x \cup y|}$.
- *Cosine similarity* is defined as $C(x,y) = \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|\cdot\|\mathbf{y}\|} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\cdot\sqrt{\sum_i y_i^2}}$.
- *Dice similarity* is defined as $D(x,y) = \frac{2\cdot|x\cap y|}{|x|+|y|}$
- *Overlap similarity* is defined as $O(x,y) = |x \cap y|$.[1]

We shall first focus on the Jaccard similarity function and later discuss other similarity functions in Section VI. In the rest of the paper, $sim(x,y)$ denotes $J(x,y)$ by default, unless otherwise stated.

The Jaccard similarity constraint $J(x,y) \geq t$ can be transformed into several equivalent forms, such as the overlap between x and y:

$$J(x,y) \geq t \Longleftrightarrow |x \cap y| \geq \alpha, \quad \text{where } \alpha = \frac{t}{1+t} \cdot (|x| + |y|) \tag{1}$$

Given a document "`the lord of the rings`", we can tokenize the document with white spaces into the following record $x = \{ A, B, C, D, E \}$, where A stands for the first "`the`", B for "`lord`", and so on. Note that we treat each subsequent occurrence of the same token as a new token [5], and hence the first "`the`" has been transformed into A and the second "`the`" into D. Assuming that $idf_B > idf_E > idf_C > idf_D > idf_A$, the record can be canonicalized according to $\mathcal{O}_{idf}$ into the array $x = [\, B, E, C, D, A \,]$.

An inverted index, I_w, is a data structure that maps a token w to a sorted list of record identifiers such that the corresponding records contain w [11]. Accordingly, $I_w[i]$ indicates the i-th entry in the inverted index of token w.

B. Prefix Filtering Methods for Similarity Joins

Several approaches to traditional similarity join problem are based on the prefix filtering principle [5], [4], [6]. The intuition is that if two records are similar, some fragments of them should overlap with each other. We formally state the prefix filtering principle below:

Lemma 1 (Prefix Filtering Principle): Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each sorted by $\mathcal{O}$. Let the p-prefix of a record x be the first p tokens of x. If $O(x,y) \geq \alpha$, then the $(|x| - \alpha + 1)$-prefix of x and the $(|y| - \alpha + 1)$-prefix of y must share at least one token.

Note that the above prefix filtering principle is a necessary but not sufficient condition for the corresponding overlap constraint.

Existing approaches usually follow the following filter-and-refine framework:

Indexing Phase Inverted indices are built for tokens that appear in the prefixes of the records.

Candidate Generation Phase The inverted indices for tokens in the prefix of each record are probed to generate a set of *candidate pairs*. The two records of a candidate pair are guaranteed to share at least one token in their prefixes. The candidate pairs are guaranteed to be a superset of the final results due to the prefix filtering principle.

Verification Phase Each candidate pair is evaluated against the similarity constraint and added to the final results if its similarity is no less than the threshold.

To ensure the prefix filtering-based approach does not miss any similarity join result, it can be shown that we need a prefix of length $|x| - \lceil t \cdot |x| \rceil + 1$ for each record x.

We now review All-Pairs [4], which is a state-of-the-art, prefix-filtering-based algorithm for processing similarity joins. The pseudo-code for All-Pairs is given in Algorithm 1. It takes as the input a collection of records already sorted in the ascending order of their sizes. It iterates through each record x, looking for candidates that intersect x's prefix (Lines 8-11). Afterwards, x and all its candidates will be verified against the similarity threshold to return the correct join results (Lines 13-15). The algorithm also employs size filtering technique [7] to reduce accesses to inverted lists by considering only candidates whose size is no less than $t \cdot |x|$ (Line 10). [2].

[1] For the ease of illustration, we do not normalize the overlap similarity to $[0,1]$.

[2] A recent improvement, ppjoin [6], employs two major optimizations over the All-Pairs algorithm. We will discuss how to integrate the two optimizations into our algorithm in Section V.

Algorithm 1: All-Pairs (R, t)

Input : R is a collection of records sorted by the increasing order of their sizes; each record has been canonicalized by a global ordering $\mathcal{O}$; a Jaccard similarity threshold t

Output : All pairs of records $\langle x, y\rangle$, such that $sim(x, y) \geq t$

```
1  S ← ∅;
2  I_i ← ∅ (1 ≤ i ≤ |U|);
3  for each x ∈ R do
4      A ← empty map from record id to int;
5      p_p ← |x| − ⌈t · |x|⌉ + 1;
6      for i = 1 to p_p do
7          w ← x[i];
8          for j = 1 to |I_w| do
9              y ← I_w[j];
10             if |y| ≥ t · |x| then
11                 A[y] ← A[y] + 1;
12         I_w ← I_w ∪ { x };
13     for each y such that A[y] > 0 do
14         if J(x, y) ≥ t then
15             S ← S ∪ {(x, y)};
16 return S
```

C. Index Reduction over the Prefix Filtering

Another significant optimization to the All-Pairs algorithm is the introduction of an index reduction technique. It was first used implicitly in [4] for cosine similarity function, and was extended to other similarity functions in [6]. We illustrate the basic idea in the following example.

Example 1: Consider the following two records x and y, and the similarity threshold of 0.8. Tokens "?" indicate that we have not accessed those tokens and do not know their contents yet.

$$x = [A, B, ?, ?, ?]$$
$$y = [B, ?, ?, ?, ?]$$

The prefix length of x is 2. If y contains the token B but not A, the maximum possible similarity of the pair $\langle x, y\rangle$ is at most $\frac{4}{5+5-4} = 0.67$. Therefore this pair cannot meet the similarity threshold though they share a common token B in their prefix.

This suggest that we do not need to index token B for x. We formally state the *index reduction principle* in Lemma 2.

Lemma 2 (Index Reduction Principle): Given a record x, All-Pairs only needs to index its $|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$-prefix to produce correct join results.

Proof: [Index Reduction Principle] Consider two records x and y, where y comes after x in the collection. We assume $J(x, y) \geq t$, and none of the tokens in the $|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$-prefix of x is contained in y. The maximum possible value of $|x \cap y|$ is $\lceil \frac{2t}{1+t} \rceil \cdot |x| - 1$. Since the records are sorted in the ascending order of their sizes, we know $|y| \geq |x|$. Therefore $J(x, y) = \frac{|x \cap y|}{|x|+|y|-|x \cap y|} \leq \frac{|x \cap y|}{|x|+|x|-|x \cap y|}$. After substituting with the maximum possible value of $|x \cap y|$, the Jaccard similarity of x and y is no greater than $\frac{\lceil \frac{2t}{1+t} \rceil \cdot |x| - 1}{\lfloor \frac{2}{1+t} \rfloor \cdot |x| + 1}$, which is less than t. This contradicts the assumption that $J(x, y) \geq t$, and hence the index reduction principle holds. ■

Lemma 2 relies on the fact that we process records in the increasing order of their sizes. The All-Pairs algorithm thus can be improved to reduce the size of inverted indices as follows. In Algorithm 1 Line 12, after x has been scanned and inserted into inverted lists, all the following records to be scanned are no shorter than x. Therefore we can apply Lemma 2 to reduce the number of x's tokens to be indexed. According to Lemma 2, we only need to index the first $|x| - \lceil \frac{2t}{1+t} \cdot |x| \rceil + 1$ tokens of x. Algorithm 2 describes the integration of the above index reduction technique into All-Pairs algorithm.

Therefore, we distinguish two kinds of prefixes for a record: the probing prefix with length p_p (Line 5 in Algorithm 1 and the indexing prefix with length p_i (Line 1 in Algorithm 2). In order to avoid ambiguity, the word "prefix" denotes probing prefix by default, unless otherwise noted.

Algorithm 2: Replacement of Line 12 in Algorithm 1

```
1  p_i ← |x| − ⌈2t/(1+t) · |x|⌉ + 1;
2  if i ≤ p_i then
3      I_w ← I_w ∪ { x };
```

III. Top-k Similarity Join Algorithms

In this section, we first give a basic algorithm to answer top-k similarity joins, followed by two optimizations exploiting the ordering that the tokens are probed to form candidate pairs.

A. Overview of the Algorithm

The main technical challenge for top-k similarity joins is that the similarity value of the k-th largest pair is unknown. If this value were known to us, we can run a state-of-the-art similarity join algorithm, e.g., All-Pairs[4] or ppjoin+[6], with the threshold and obtain correct results.

An important observation is that we can enumerate all the "necessary" similarity thresholds in the decreasing order as $t_1, t_2, \ldots$, where $t_i > t_j, \forall i < j$. By "necessary" thresholds, we mean that if we change between different thresholds, it is possible that similarity join result will change. To derive the necessary thresholds for prefix-filtering based similarity join algorithms, we note that the set of candidate pairs that the algorithms consider are solely determined by the prefixes of the records. Therefore, if we lower the current threshold t_i to the first value t such that there is a record in the database whose prefix will extend by one more token, we have to consider this t and set $t_{i+1} = t$.

A simple algorithm utilizing the above observation is as follows: for each threshold t_i, we invoke a similarity join algorithm to find all pairs with similarity values no less than t_i. The stopping condition is simply when the result size is larger than k.

The main problem with this algorithm is the redundant computation caused by the repeated invocation of the similarity join algorithm. This can be solved by using an incremental similarity join algorithm instead. Denote the prefix of for a record x for a threshold t as $prefix_t(x)$. We note that $prefix_{t_i}(x) \leq prefix_{t_j}(x)$ if $t_i > t_j$. Below we describe major modifications needed to devise an incremental version of the All-Pairs algorithm:

1) In the incremental algorithm, we *cannot* discard candidate pairs whose similarity value is less than the *current* threshold (unless $k = 1$). Nevertheless, we only need to keep the largest k pairs seen so far.
2) we need to devise a new stopping condition. We can stop the execution of the algorithm when the similarity value of the current k-th result is larger than the next similarity threshold.

In the following subsections, we will describe the algorithm in more details.

B. Temporary Results and Events

As the similarity threshold decreases, the prefixes of records will increase. We call the extension of a record's prefix by one more token w a *prefix event*, and the record in question as the *source* of the event. A prefix event implies an upper bound of the similarity value of unseen pairs, as they share no common token before w in the prefix. The incremental similarity join algorithm is then run in an event-driven manner: once a prefix event is triggered, we (1) probe the inverted list of the token w to find candidates pairs, and then (2) push its next event into a priority queue (i.e., to extend prefix to w's next token in this record).

Note that the similarity threshold which triggers a prefix event is exactly an upper bound of the similarity value of unseen pairs.

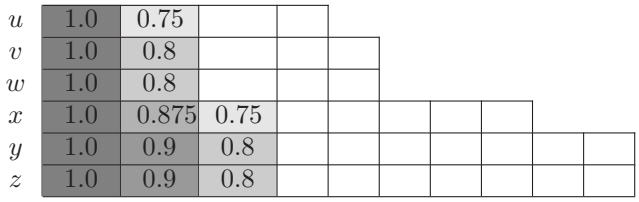


Fig. 1. Event-Driven Model

Figure 1 illustrates the event-driven idea. Each row represents a record, and each box in the row represents a token. The number in the box shows the similarity threshold to trigger a prefix event that involves the token corresponding to the box. The prefix length for each record starts with 1, indicating a similarity threshold of 1.0. The similarity threshold decreases gradually with the extending prefix. We use a max-heap to process the events in decreasing order of their associated similarity thresholds.

Once a prefix event happens, a new token w is included by the extended prefix of a record x. We probe w's inverted list in order to find all the records that share the common token w with the record x in their current prefixes. The records are paired with x to form candidate pairs, with their exact similarity values calculated by the similarity function and then added to temporary results. We keep only k temporary results with the highest similarity values. We can output a result as soon as the similarity threshold for the next prefix event is no greater than the similarity values of the current result.

For each prefix event, we need to answer the following question: "*what is the similarity threshold that triggers this prefix event*?" Answering this question will lead us to establish an upper bound for the similarity value of unseen pairs. We call this upper bound the *similarity upper bound* of a prefix event. Consider a record x with a prefix length p_x, the answer to the above question is equivalent to the *maximum* similarity threshold when x has a prefix length of p_x. According to the prefix filtering principle, this value is $1 - \frac{p_x - 1}{|x|}$. We use $\langle\, x, p_x, s_{px} \,\rangle$ to denote a prefix event, where s_{px} is the similarity upper bound for x with a prefix length of p_x.

Algorithm 3 describes this incremental similarity join algorithm. A fixed sized min-heap T is used to keep the largest k pairs seen so far. $T[k]$ gives the pair with the k-th largest similarity. For the ease of illustration, we can initialize T with any k pairs (e.g., by pairing record 1 with records 2 to $k+1$) so that it is full. The prefix length for each record is initialized as 1 first. for each record x, we insert $\langle\, x, 1, 1.0 \,\rangle$ into a max heap based on the upper bounds. The similarity upper bound for these prefix events are initialized as 1.0 (Line 1). The algorithm then iteratively pulls the next prefix event $\langle\, x, p_x, s_{px} \,\rangle$, and probes the inverted lists of tokens in the probing prefix $x[1\,..\,p_x]$. x is paired with the records that share at least one token with x in their prefixes. The pairs and their similarity values returned by the similarity function are regarded as temporary results and added to T. Next, we extend x's prefix length to p_x+1, and recalculate the similarity upper bound for x with the current prefix length of p_x+1 (Line 17). Finally, the new event $\langle x, p_x + 1, s'_{px} \rangle$ is pushed into the max heap for prefix events. The algorithm stops when the max heap for prefix events is empty or all the remaining events in the heap has a lower similarity upper bound than that of the k-th result in T.

Size filtering (Line 12) can be applied when we have k temporary results. Specifically, the lowest similarity value among the k temporary results, denoted as s_k, is regarded as a similarity threshold, and we can estimate the minimum and the maximum required sizes in order to admit only the records that can achieve a similarity value of at least s_k with the record x.

IV. Optimizations for the Incremental Algorithms

Both All-Pairs and ppjoin family algorithms exploit the ascending ordering of record sizes and the global ordering of tokens. In order to further optimize the incremental topk-join algorithm, we will exploit the ascending ordering of the lowest similarity value among k temporary results, and the descending ordering of the similarity upper bound.

Algorithm 3: Top-kSimilarityJoin (R)

Input : R is a collection of records; each record has been canonicalized by a global ordering $\mathcal{O}$
Output : Top-k pairs of records $\langle x, y \rangle$ ranked by their similarity value

1 $E \leftarrow$ InitializeEvents(R);
2 $T \leftarrow$ InitializeTempResults(R); /* Store any k pairs as temp results in T */;
3 $I_i \leftarrow \emptyset$ $(1 \leq i \leq |U|)$;
4 **while** $E \neq \emptyset$ **do**
5 $(x, p_x, s_{px}) \leftarrow E.pop()$;
6 **if** $s_{px} \leq T[k].sim$ **then**
7 **break**; /* stop here */;
8 $w \leftarrow x[p_x]$;
9 $s_k \leftarrow T[k].sim$;
10 **for** $j = 1$ to $|I_w|$ **do**
11 $y \leftarrow I_w[j]$;
12 **if** $|y| \in [s_k|x|, |x|/s_k]$ **then** /* size filtering */
13 $sim(x, y) \leftarrow$ CalcSimilarity(x, y);
14 $T.add((x, y), sim(x, y))$;
15 $s_k \leftarrow T[k].sim$;
16 $I_w \leftarrow I_w \cup \{ x \}$; /* index the current prefix */;
17 $s'_{px} \leftarrow$ SimilarityUpperBound-Probe$(x, p_x + 1)$;
18 $E.push(x, p_x + 1, s'_{px})$;
19 **return** T

Algorithm 4: InitializeEvents (R)

1 $E \leftarrow \emptyset$;
2 **for each** x *in* R **do**
3 $E.push(x, 1, 1.0)$; /* E is a max-heap */;
4 **return** E

A. Optimization for Verification

One problem with Algorithm 3 is that the similarity between a pair may be computed up to l times, if they share l common tokens in their prefixes. Like in All-Pairs, we call the computation of the similarity value of a candidate pair (Line 13 in Algorithm 3) *verification*. Obviously, repeated verifications incur unnecessary overhead.

A simple solution to address this problem is to remember *all* candidate pairs that have been verified in a hash table. However, the hash table will have too many entries. We note that pairs that have been verified only once before the algorithm stops do not need to be remembered in the hash table. To minimize the size of the hash table, we look into the contents of x and y to tell whether a pair will be identified *a second time*. This is achievable as s_k is monotonically increasing and we can use the current s_k value to calculate the *maximum prefixes* that will be accessed before the algorithm stops. We put a pair into the hash table *only if* a second common token in both maximum prefixes is found. Since the similarity function will access the contents of x and y sequentially, we can be seamlessly integrated this optimization (Lines 5–6 in Algorithm 6) into the similarity calculation (Line 4) with little overhead.

Lemma 3: Algorithm 6 guarantees that each candidate pair will be evaluated exactly once.

Algorithm 5: SimilarityUpperBound-Probe (x, p_x)

Input : A record x and a prefix length p
Output : The upper bound of the similarity value of x and another record provided that $x[p]$ is the first common token of x and the other record

1 $s_{px} \leftarrow 1 - \frac{p_x - 1}{|x|}$;
2 **return** s_{px}

Algorithm 6: Replacement of Line 13-15 in Algorithm 3

1 $\mathsf{x} \leftarrow \max(x, y)$; $\mathsf{y} \leftarrow \min(x, y)$; /* deal with (x, y) and (y, x) */;
2 **if** $(\mathsf{x}, \mathsf{y}) \notin H$ **then**
3 $l_{\mathsf{x}} \leftarrow |\mathsf{x}| - \lceil s_k \cdot |\mathsf{x}| \rceil + 1$; $l_{\mathsf{y}} \leftarrow |\mathsf{y}| - \lceil s_k \cdot |\mathsf{y}| \rceil + 1$;
4 $sim(\mathsf{x}, \mathsf{y}) \leftarrow$ CalcSimilarity(x, y);
5 $pos_{\mathsf{x}} \leftarrow$ the position of x and y's second common token in x;
6 $pos_{\mathsf{y}} \leftarrow$ the position of x and y's second common token in y;
7 **if** $pos_{\mathsf{x}} \leq l_{\mathsf{x}}$ **and** $pos_{\mathsf{y}} \leq l_{\mathsf{y}}$ **then**
8 $H \leftarrow H \cup \{ (\mathsf{x}, \mathsf{y}) \}$;
9 $T.add((\mathsf{x}, \mathsf{y}), sim(\mathsf{x}, \mathsf{y}))$;
10 $s_k \leftarrow T[k].sim$;

Theorem 1: Algorithm 3 (together with Algorithm 6) verifies the minimum number of pairs for the given similarity upper bounding function in Algorithm 5.

Proof: Assume the contrary that the number of pairs verified by our algorithm is not minimum. Then there exists a pair (x, y), which is verified by our algorithm, but is not verified by another correct top-k similarity join algorithm. Note that our algorithm generate candidates by decreasing order of similarity upper bound. All the candidate pairs generated can achieve this similarity upper bound if their unseen contents are identical. Since this similarity upper bound is no less than the final s_k, there exists a collection such that $sim(x, y) \geq s_k$, and therefore (x, y) is a join result. Any other algorithm that does not verify this pair will miss this pair in their output and is thus incorrect. ■

Note that it is not easy to upper bound the similarity values of unseens pairs and the one we used in Algorithm 5 is quite effective. Note that if we do not access part of *all* the records simultaneously, it is unlikely that we will have a useful upper bounding function — as long as two records have the same lengths and no part of their contents is accessed, we have to use 1.0 as the upper bounding score as the two might be identical.

B. Optimization for Indexing

As discussed in Section II-C, traditional similarity join algorithms benefit from the index reduction optimization by

only indexing the minimum number tokens. However, the optimization require that records are sorted in the ascending order of their sizes. Nevertheless, we manage to achieve a similar index reduction effect by exploiting the fact that *records inserted into inverted indexes are sorted by decreasing similarity upper bound* . We illustrate the idea in the example below.

Example 2: Consider the following three records. Tokens "?" indicate that we have not accessed those tokens and do not know their contents yet.

$$x = [A, \underline{C}, ?, ?, ?]$$
$$y = [\underline{C}, ?, ?, ?]$$
$$z = [B, \underline{C}, ?, ?, ?]$$

Suppose we are about to insert x into C's inverted list. According to Algorithm 5, the maximum possible similarity value that x and another record can achieve is 0.8, provided that C is their first common token. This is case for the pair (x, z) if their unseen parts are identical. However, if x is inserted into C's inverted list and is paired with another record during *subsequent* probings, can they still achieve a similarity value of 0.8? A subsequent prefix event with similarity upper bound no larger than 0.8 induces tokens contained only in the other record rather than x. Take prefix event $\langle z, 2, 0.8 \rangle$ as example, whose similarity upper bound is 0.8. Since token B is contained only in z, the maximum possible similarity value achieved by x and z is $\frac{4}{6} = 0.67$ when their unseen tokens are identical. The similarity upper bound can be achieved *if and only if* there exists another record z, $|x| = |z|$ and $\forall i \in [p_x, |x|],\ x[i] = z[i]$. In this case, the max overlap is $x - p_x + 1$ and the union size is $x + p_x - 1$. The similarity upper bound for $\langle x, 2, 0.8 \rangle$ is therefore reduced to $\langle x, 2, 0.67 \rangle$ for subsequent probings, and we do not need to insert x into C's inverted list if this upper bound is below s_k.

The following Lemma allows us to reduce the number of tokens that need to be indexed and accessed.

Lemma 4: Given a record x and a prefix length p_x, the maximum possible similarity value of x and another record is $\frac{|x|-p_x+1}{|x|+p_x-1}$ provided that $x[p_x]$ is the first common token of x and the other record, and this pair is found by probings after inserting x into $x[p_x]$'s inverted list.

We call the similarity upper bound given by Lemma 4 *indexing similarity upper bound*, while the upper bound returned by Algorithm 5 is called *probing similarity upper bound*. We use this optimization for indexing to calculate the indexing similarity upper bound of $x[p_x]$ (denoted as s_i), and then compare it with the similarity value of the k-th temporary results (denoted as s_k). If it is found to be smaller than s_k, we do not perform inverted index insertion for $x[p_x]$.

Another optimization can be derived from this tighter indexing similarity upper bound: we can stop insertion for all inverted lists when an $s_i < s_k$ is found. This is because that similarity upper bound $= \frac{|x|-p_x+1}{|x|+p_x-1} = \frac{1-\frac{p_x-1}{|x|}}{1+\frac{p_x-1}{|x|}}$; it varies monotonically with the probing similarity upper bound $1 - \frac{p_x-1}{|x|}$. Since the probing similarity upper bound is monotonically decreasing, the indexing similarity upper bound is also monotonically decreasing.

Algorithm 7 implements the optimization for indexing tokens. It computes the indexing similarity upper bound for each $\langle x, p_x, s_{px} \rangle$, and compares the indexing similarity upper bound with s_k. If the indexing similarity upper bound is smaller, a boolean flag is set as *true* to prevent any further index insertion.

Algorithm 7: Replacement of Line 16 in Algorithm 3

```
1 if f = false then        /* f is initialized as false */
2     s_i ← SimilarityUpperBound-Index(x, p_x);
3     if s_i ≥ s_k then
4         I_w ← I_w ∪ { x };
5     else
6         f ← true;      /* stop index insertion */
```

Algorithm 8: SimilarityUpperBound-Index (x, p_x)

```
1 ubound ← (|x|−p_x+1)/(|x|+p_x−1);
2 return ubound
```

C. Further Optimization for Indexing

The above optimization can be further extended to limit access to the invert lists after the index insertion is stopped. Consider the case that a prefix event $\langle x, p_x, s_{px} \rangle$ is trigger, and we are accessing $x[p_x]$'s inverted list to find the records that contain $x[p_x]$ in their indexing prefixes. When an entry in the inverted list is accessed, we ask the question: given $\langle x, p_x, s_{px} \rangle$'s probing similarity upper bound and the current entry's probing similarity upper bound when it is inserted into the inverted list, what is the maximum possible similarity between x and the current entry? Answering this question help us to stop accessing the subsequent entries in the inverted list, since the entries are inserted into $x[p_x]$'s inverted list according to *decreasing* order of probing similarity upper bound.

We solve this issue by estimating the sizes of intersection and union between two records x and y. Suppose $x[p_x]$'s probing similarity upper bound is s_{px}, and y's probing similarity upper bound when it is inserted into $x[p_x]$'s inverted list is s_{py}.[3] We distinguish two scenarios according to the maximum possible intersection of the two records:

1) $|x \cap y| = x \cdot s_{px}$, if $y \cdot s_{py} \geq x \cdot s_{px}$.
2) $|x \cap y| = y \cdot s_{py}$, if $y \cdot s_{py} \leq x \cdot s_{px}$.

[3]For the ease of computation and exposition, we use probing similarity upper bound instead of indexing similarity upper bound for the current entry in the inverted list.

- For scenario (1),

$$\begin{aligned}|x \cup y| =& |x| + |y| - |x| \cdot s_{px} \\ \geq& |x| + |x| \cdot \frac{s_{px}}{s_{py}} - |x| \cdot s_{px}\end{aligned}$$

Therefore,

$$\begin{aligned}J(x,y) =& \frac{|x \cap y|}{|x \cup y|} \\ \leq& \frac{s_{px}s_{py}}{s_{px} + s_{py} - s_{px}s_{py}}\end{aligned}$$

- For scenario (2),

$$\begin{aligned}|x \cup y| =& |x| + |y| - |y| \cdot s_{py} \\ \geq& |y| \cdot \frac{s_{py}}{s_{px}} + |y| - |y| \cdot s_{py}\end{aligned}$$

Therefore,

$$\begin{aligned}J(x,y) =& \frac{|x \cap y|}{|x \cup y|} \\ \leq& \frac{s_{px}s_{py}}{s_{px} + s_{py} - s_{px}s_{py}}\end{aligned}$$

We call the above maximum possible similarity value $\frac{s_{px}s_{py}}{s_{px}+s_{py}-s_{px}s_{py}}$ *accessing similarity upper bound*. Since it decreases monotonically with the decreasing s_{py}, access to the rest of the inverted list for token $x[p_x]$ can be stopped when its value is smaller than s_k. Furthermore, the value is also decreasing monotonically when s_{px} is decreasing. The entries starting from y till the end of the inverted list can be deleted because future probings of this inverted list will be incurred by a smaller s_{px}. The above method is implemented in Algorithm 9. [4]

Algorithm 9: Replacement of Line 11 in Algorithm 3

```
1 y ← I_w[j];
2 if SimilarityUpperBound-Access(1 − (p_x − 1)/|x|, 1 − (p_y − 1)/|y|) < s_k then
3     I_w ← I_w \ I_w[j .. |I_w|];      /* remove entries */;
4     break;
```

Algorithm 10: SimilarityUpperBound-Access (s_{px}, s_{py})

```
1 ubound ← s_px s_py / (s_px + s_py − s_px s_py);
2 return ubound
```

[4]Section V will discuss the integration of positional filtering, which eventually keeps the value of p_y

V. Implementation Details

A. Integration with Positional Filtering and Suffix Filtering

The positional filtering and suffix filtering technique proposed in [6] can be employed here so as to reduce the number of pairs to be verified for their similarity value. Each pair identified by accessing inverted lists is tested under positional filtering and the following suffix filtering before verification is performed. For positional filtering, we use the lowest similarity value of temporary results, s_k, if more than k temporary results have been obtained. The similarity value is regarded as a threshold, and the minimum required overlap between x and y is computed for each pair (x, y) to be verified. With the positions of the first common token in both x and y, we can estimate the maximum possible overlap of (x, y), and then compare with the minimum required value. A pair is admitted for verification only if the estimated maximum possible overlap is no smaller than the minimum required value. For suffix filtering, s_k is considered as a similarity threshold and converted to a Hamming distance threshold for each pair to be verified. We perform suffix filtering under the Hamming distance constraints and remove the disqualified pairs before performing verification.

B. Initialization of Temporary Results

To fill up T with k temporary results during the initialization stage (Line 3, Algorithm 3) so that we have an initial s_k, one solution is to generate k candidate pairs with high similarity value beforehand. Specifically, we examine several tokens with medium document frequency (e.g., between 10 and 100) when performing tokenization, and record their df values as well as inverted lists. To initialize T, we search among these tokens for the token w with the smallest document frequency df_w where $\binom{df_w}{2} \geq k$, and then generate candidate pairs with w's inverted list. The k candidate pairs with the highest similarities are inserted into T as temporary results.

C. Compression on Prefix Events

Since the probing and indexing similarity upper bound of a prefix event $\langle x, p_x, s_{px} \rangle$ is only dependent on the size of the record and the prefix length, the prefix events with the same record size and prefix length can be grouped together to save space and time. Specifically, we insert prefix events in the form of $\langle |x|, p_x, s_{px} \rangle$ instead of $\langle x, p_x, s_{px} \rangle$ into the priority queue E. When a prefix event $\langle |x|, p_x, s_{px} \rangle$ is popped from E, we retrieve all the records with the same size as x and perform probing actions. This requires us to sort the records in increasing order of their sizes and divide the records into blocks, each block with the same record size.

VI. Extension to Other Similarity Functions

In this section, we generalize our main algorithm to several other similarity measures. We consider the following similarity functions: overlap, cosine, and dice similarities. The major changes are related to the size filtering condition (Line 12, Algorithm 3), maximum prefix length (Line 3, Algorithm 6),

probing similarity upper bound (Line 1, Algorithm 5), indexing similarity upper bound (Line 1, Algorithm 8), and accessing similarity upper bound (Line 1, Algorithm 10).

Overlap Similarity

size filtering	$y \geq s_k$
max prefix length	$\|x\| - s_k + 1$
probing sim ubound	$\|x\| - p_x + 1$
indexing sim ubound	$\|x\| - p_x + 1$
accessing sim ubound	$\min(s_{px}, s_{py})$

Cosine Similarity

size filtering	$\|y\| \in [s_k^2 \cdot \|x\|, \|x\|/s_k^2]$
max prefix length	$\|x\| - \lceil s_k^2 \cdot \|x\| \rceil + 1$
probing sim ubound	$\sqrt{1 - \frac{p-1}{\|x\|}}$
indexing sim ubound	$1 - \frac{p-1}{\|x\|}$
accessing sim ubound	$\frac{s_{px}^2 s_{py}^2}{s_{px}^2 + s_{py}^2 - s_{px}^2 s_{py}^2}$

Dice Similarity

size filtering	$\|y\| \in [\frac{s_k\|x\|}{2-s_k}, \frac{(2-s_k)\|x\|}{s_k}]$
max prefix length	$\|x\| - \lceil \frac{s_k\|x\|}{2-s_k} \rceil + 1$
probing sim ubound	$\frac{2(\|x\|-p_x+1)}{2\|x\|-p_x+1}$
indexing sim ubound	$\frac{\|x\|-p_x+1}{\|x\|}$
accessing sim ubound	$\frac{s_{px} s_{py}}{2s_{px}+2s_{py}-3s_{px} s_{py}}$

VII. Experiments

We present our experiment results and analyses in this section.

A. Experiment Setup

We used the following algorithms in the experiment.

topk-join is the proposed algorithm with full optimizations. The positional filtering and suffix filtering [6] techniques are also employed. The `MAXDEPTH` parameter for suffix filtering is set to 2 for DBLP and TREC, and 4 for TREC-3GRAM and UNIREF-3GRAM.

pptopk serves as the baseline algorithm, where we run a state-of-the-art similarity algorithm, ppjoin+ [6] repeatedly with decreasing similarity threshold, to find the answers to top-k queries. In each round, if the size of the result computed by the ppjoin+ algorithm is smaller than k, the similarity threshold is decreased and the above process is repeated. In this study, we decrease the similarity threshold at an equal rate. More specifically, for the Jaccard similarity, we use threshold $0.95 - 0.05 \cdot i$ for the i-th round; for the cosine similarity, we use threshold $0.975 - 0.025 \cdot i$, as the cosine similarity function is looser than the Jaccard similarity function with the same threshold.

All algorithms were implemented as in-memory algorithms, with all their inputs loaded into the memory before they were run. Since ppjoin+ has been shown to outperform alternative algorithms such as All-Pairs [4] and PartEnum [7] on large-scale real datasets under a wide range of parameter settings, we didn't consider them.

All experiments were performed on a PC with Intel an Xeon X3220 2.40GHz CPU and 4GB RAM. The operating system is Debian 4.1.1-21. The algorithms were implemented in C++ and compiled using GCC 4.1.2 with the -O3 flag.

We used the following publicly available real datasets. They covered a wide range distributions and application domains.

DBLP is a snapshot of the bibliography records from the DBLP Web site.[5] It contains about 0.9M records; each record is a concatenation of author name(s) and the title of a publication. This dataset is widely used in similarity join and near-duplicate detection research [7], [4], [12], [13], [6].

TREC is from TREC-9 Filtering Track Collections.[6] It contains 0.35M references from the MEDLINE database. We extract and concatenate author, title, and abstract fields to form records.

TREC-3GRAM the same TREC dataset tokenized into 3-grams.

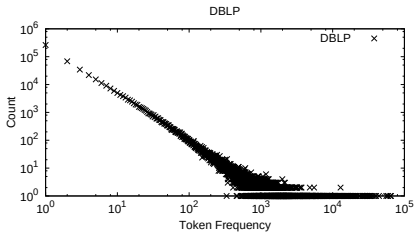
UNIREF-3GRAM denotes the UniRef90 protein sequence data from the UniProt project.[7] We extract the first 0.5M protein sequences; each sequence is an array of amino acids coded as uppercase letters. Each record is transformed into a set of 3-grams.

The datasets are transformed and cleaned by converting letters to their lowercases. White spaces and punctuations are converted into underscores before extracting q-grams to form the q-gram datasets. Exact duplicates are then removed, and the records are sorted in ascending order of size.

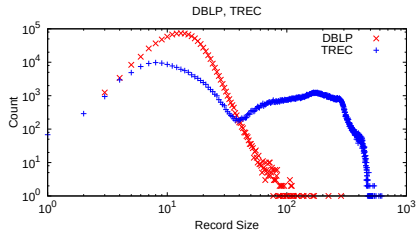
Some important statistics about the cleaned datasets are listed in Table I. Distributions of token frequency and record size are plotted in Figures 2(a)–2(c). Only the token frequency distribution for DBLP is shown because all datasets follow approximately a similar Zipf distribution. We observe that the record size distributions are quite different for DBLP, TREC, and UNIREF datasets.

The similarity measures covered by our experiments are **Jaccard** and **Cosine** similarity functions. We measure the size of candidate pairs verified by similarity functions and the running time. The running time does not include the preprocessing and loading time, as they are the same for all algorithms.[8]

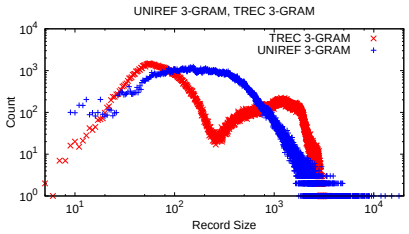
[5] `http://www.informatik.uni-trier.de/~ley/db`
[6] `http://trec.nist.gov/data/t9_filtering.html`
[7] `http://beta.uniprot.org/` (downloaded in March, 2008)
[8] The loading time is between 2 to 38 seconds.



(a) Token Frequency Distribution, DBLP



(b) Record Size Distribution, DBLP (in red) and TREC (in blue)



(c) Record Size Distribution, TREC-3GRAM (in red) and UNIREF-3GRAM (in blue)

Fig. 2. Statistics and Distributions of Datasets

TABLE I
DATASET STATISTICS

Dataset	N	$avg.size$	$\|U\|$
DBLP	855,266	14.3	509370
TREC	347,956	130.1	1066590
TREC-3GRAM	347,963	868.5	193644
UNIREF-3GRAM	500,000	372.9	158315

B. Effect of Optimization for Verification

In order to study the impact of the optimization for verification, we remove it from the topk-join algorithm, and name the resulting algorithm record-all. Hence it records in a hash table all the candidate pairs that have been verified in order to avoid repeated verification.

We measure the numbers of hash table entries for both topk-join and record-all on the TREC dataset, which were shown in Figure 3(a). Results on other datasets are similar. Both hash table sizes grow quickly with the increase of k, while applying the optimization for verification can reduce the number of hash table entries by 33

C. Effect of Optimization for Indexing

We remove the optimization for indexing in topk-join algorithm and name the resulting algorithm w/o-index-opt.

We run topk-join and w/o-index-opt algorithm on the TREC dataset and measure the peak numbers of index entries for both algorithms. Note that topk-join employs an index deletion technique to save space, so we measure the number of index entries immediately after the insertion of index has stopped but before the index deletion is performed.

The number of index entries and running time are shown in Figure 3(b) and 3(c). We observe that the number of index entries can be reduced by about 40% with the optimization, and it results in about 20% improvement on running time.

D. Comparison with pptopk

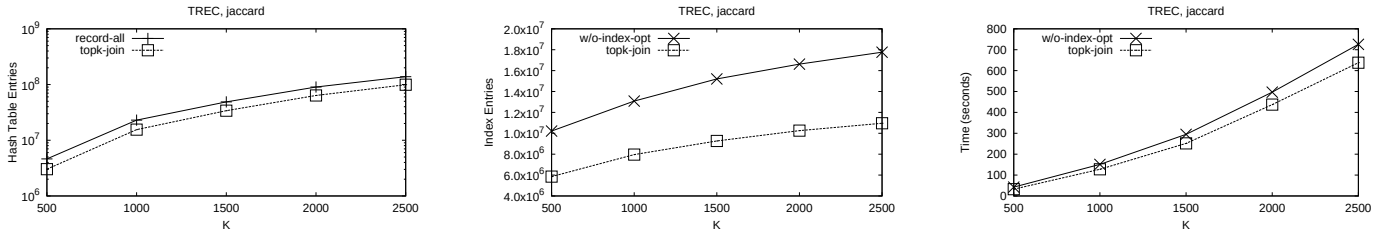
We compare the performance of our topk-join algorithm with the baseline pptopk algorithm. We measure the number of candidate pairs and running time, using the Jaccard similarity function on the DBLP and TREC datasets, and the cosine similarity function on the TREC-3GRAM dataset.

Figures 4(a)– 4(c) show the number of candidate pairs for both topk-join and pptopk algorithm with different k values. The candidate sizes for both algorithms grow with the increase of k. The difference is that the candidate size of topk-join grows smoothly with the increase of k, while the number of pptopk jumps at a certain k. This is because pptopk has to "guess" the similarity threshold for the input k. On DBLP, topk-join generates similar candidate size with pptopk, while on the other two datasets topk-join produces fewer candidates than pptopk. The candidate size is primarily dependent on two factors:

- the effect of positional filtering and suffix filtering. Both filtering techniques are more effective at high similarity thresholds. If we use the final value of s_k as the similarity threshold t for pptopk, the candidate size of pptopk is smaller than that of topk-join, because the positional filtering and suffix filtering use a constant similarity threshold t in pptopk, compared to a threshold that gradually increases to t in topk-join.
- the guess on the similarity threshold for pptopk. If the guess is "conservative", i.e. the threshold is much lower than the final s_k, pptopk may produce too many candidate pairs and join results.

The reason why topk-join produces similar number of candidate pairs with pptopk on DBLP but much fewer on TREC and TREC-3GRAM is: (a) On DBLP, the join results have higher similarity than those on the other two datasets. Positional filtering and suffix filtering are quite effective against candidate pairs and then play a more important role in determining candidate size. On TREC and TREC-3GRAM, positional filtering and suffix filtering are less effective, as the candidate sizes shown in the figures are quite large. (b) A subtle difference between the guessed similarity threshold and the final s_k might lead to a huge increase in candidate size. This is more substantial when similarity threshold is low, and thus the effect is enlarged for TREC and TREC-3GRAM.

The running time for both algorithms is shown in Figures 4(d)–4(f). topk-join outperforms pptopk in most cases. topk-join can achieve up to 1.6x speed-up on DBLP, 2x on TREC, and 3.4x on TREC-3GRAM. The main reason is that pptopk produces more results than needed for most k, while topk-join is directly optimized for the parameter k. Table II shows the number of results obtained by pptopk during each



(a) Jaccard, TREC, Hash Table Entries (b) Jaccard, TREC, Index Entries (c) Jaccard, TREC, Running Time

Fig. 3. Optimization for Verification and Indexing

round on the TREC dataset. In addition, topk-join employs an optimization technique on indexing and saves the overhead due to index construction and accesses.

TABLE II
PPTOPK'S JOIN RESULT SIZES IN EACH ROUND

t	0.95	0.90	0.85	0.80	0.75	0.70	0.65	0.60
#	34	84	187	404	725	1162	1819	3361

It is interesting to note that topk-join is faster than pptopk on DBLP although it produces more candidate pairs. This is because the average record size in DBLP is short (only 14). The similarity computation can be done efficiently, while index construction, accessing, and other overheads are more substantial. Thanks to the optimization for indexing, topk-join is able to reduce the cost on index construction and accessing, and therefore outperforms pptopk on DBLP dataset.

It is also interesting to observe that the two outliers on the TREC-3GRAM dataset, where topk-join produces fewer candidate pairs but turns out to be slower. We found that this is mainly due to the more powerful size filtering effect in pptopk than in topk-join. As pptopk scans records by the ascending order of record sizes, the entries in the inverted lists can be safely deleted if their sizes are below the required minimum value, as they cannot produce any future join results. However, in topk-join these entries cannot be deleted but only kept from forming candidates when they are accessed, as they might be identified as candidates by future index probes and produce join results. This will incur more inverted list accessing time for topk-join. The impact of size filtering is especially substantial on the TREC-3GRAM dataset, because the inverted lists for 3-grams are typically very long.

E. Verifications per Record

We measure the average number of verifications for each record running topk-join on the TREC dataset, and plot the result in Figure 5(a). We also show the number k in the figure. It is clear that the the average number of verifications per record is much smaller than k. The number is 13.3 when $k = 500$, and 397.8 when $k = 2500$.

The result is interesting in the following sense. Consider a hypothetical algorithm equipped with an Oracle such that for each record, the Oracle suggests k results that have the largest similarities. Then the algorithm can simply iterate over the records, calculating the actual similarity values for the candidates suggested by the Oracle, and retain only the largest k pairs. The average number of verifications per record will be k. Our result here shows that topk-join performs even fewer number of verifications than this hypothetical algorithm.

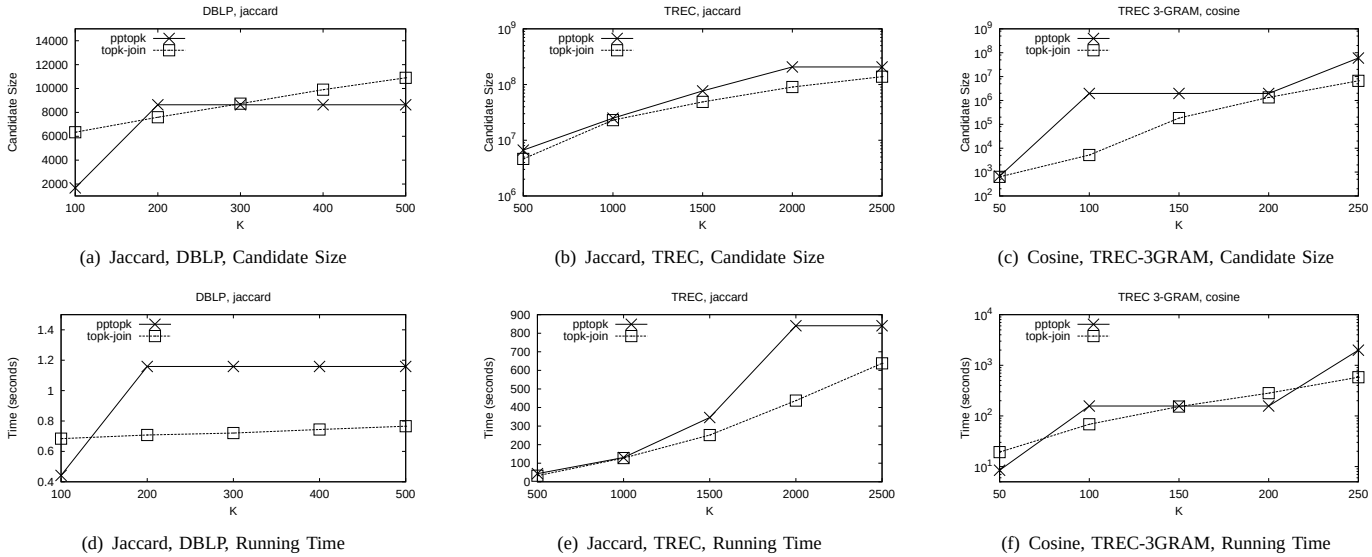
F. Answering Top-k Queries Progressively

We investigate the performance of the topk-join algorithm to output join results progressively. This feature benefits applications that require the system to output join results while the algorithm is still running. For instance, a user may input an initial $k = 100$ but terminate the execution of algorithm when she is already satisfied with the first 50 results.

We ran the experiment on the TREC-3GRAM and UNIREF-3GRAM datasets using the Jaccard similarity function with $k = 200$. The metrics measured are: the probing similarity upper bound, the similarity of the k-th temporary result, and the output time. All measures are taken with respect to each join result.

Figure 5(b) shows the probing similarity upper bound of unprocessed prefix events (denoted as TREC 3-GRAM UB and UNIREF 3-GRAM UB) and the lowest similarity value among the current k temporary results (denoted as TREC 3-GRAM Sk and UNIREF 3-GRAM Sk) when the i-th (x-axis) temporary result is confirmed as a final result and sent to the output. The probing similarity upper bounds are close to 1.0 for both datasets when the first join result is sent to the output, and decreases almost linearly with more join results are sent to the output. On the other hand, the similarity of the k-th temporary result s_k almost remains the same during the whole process, except for a slight increase on the TREC-3GRAM dataset for the first 20 results. This suggests that we can quickly fill the k temporary results whose lowest similarity value is close to the k-th final result. More than that, this strengthens the effect of the optimizations for both verification and indexing since we can find an s_k close to the final value at an early stage.

The elapsed time when each final result is sent to the output is plotted in Figure 5(c). The rate of result ouptut is slow at the beginning, but increasing as more temporary results are confirmed as final results and output. There are at least two reasons for this observation: (a) The un-indexed part of each record is long at the beginning so that it is difficult to prune candidate pairs by estimating the upper bound of the overlap.



(a) Jaccard, DBLP, Candidate Size (b) Jaccard, TREC, Candidate Size (c) Cosine, TREC-3GRAM, Candidate Size

(d) Jaccard, DBLP, Running Time (e) Jaccard, TREC, Running Time (f) Cosine, TREC-3GRAM, Running Time

Fig. 4. Experiment Results I

This renders positional filtering less effective. (b) The prefix lengths are short when the first few join results are sent to the output. Note that the tokens within each record are sorted by decreasing idf. Due to the low frequency tokens contained by the prefix, the pairs have more chance of being similar if they share such tokens. For example, two records might be quite similar if the first common token is "`ppjoin`", but less chance of being similar if the first common token is "`algorithm`". These candidate pairs may have medium similarity values and survive suffix filtering though they might not be the final results. On the contrary, the positional filtering and the suffix filtering become more effective when more join results are output, since the unseen part of tokens are smaller for each record and the first common token of a candidate pair is higher in frequency. The optimization for indexing is also working when probing similarity upper bound decreases to a certain value, which facilitates the algorithm to output the last few join results.

VIII. Related Work

Similarity Join Similarity joins have attracted much attention in several research communities. It has been studied for a wide range of applications, such as record linkage [3], merge-purge [14], data deduplication [15], and name matching [16].

Existing work on traditional similarity joins can be categorized into two groups. The first is to retrieve exact answers to the problem where all the pairs satisfying the similarity constraints are returned. [17], [5], [4], [6] are based on inverted indices and various filtering techniques. Alternatively, [7] employs the pigeon hole principle, which carefully divide the records into partitions and then hash them into signatures, with which candidate pairs are generated, followed by a post-filtering step to eliminate false positives. There has been two recent work on similarity join. [18] designs a novel framework to identify similar records with some token transformations. In [19], LSS algorithm is proposed to perform similarity join using Graphics Processing Unit (GPU).

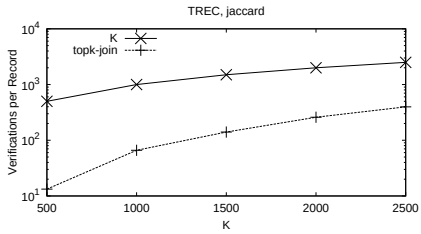
The work in the second category concentrates on retrieving approximate answers to the similarity join problem [20], [21], [22], [23]. Local Sensitive Hashing(LSH) [23] is a widely adopted technique for similarity search. [20] proposed a shingle-based approach to approximately identify near duplicate Web pages. Another synopsis-based algorithm is based on randomized projection [22].

Several existing work studies the similarity search problem [24], [25], [26], [27], which returns the records in a collection whose similarity with the query exceeds a given threshold. Based on the inverted list framework, [26] proposes an efficient principle to skip records when accessing inverted lists. For information retrieval(IR) purpose, [27] designs efficient techniques for indexing and processing similarity queries under IR-style similarity functions.

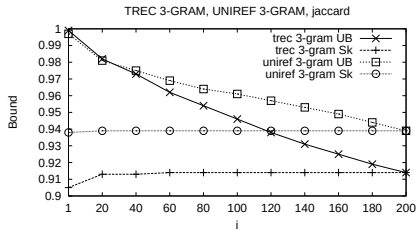
Similarity functions were studied in a variety of applications. For text documents, edit distance [28] and Jaccard similarity on q-grams [25] are widely used. In large-scale Web applications, similarities between two documents is usually measured by Jaccard or overlap similarity on small-sized synopses extracted from the documents [29], [21]. Soundex is a commonly used phonetic similarity measure for names matching tasks [30].

Top-k Queries The problem of top-k query processing has been studied by Fagin *et al* [31], [32]. Much work build upon Fagin's work for different application scenarios, e.g., ranking query results from structured databases [33], processing distributed preference queries [34] and keyword queries [35].

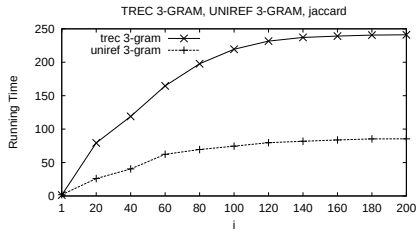
[36] studies the top-k spatial join problem, which retrieves pairs of objects that satisfy a given spatial constraint. However, as the distance function used in our problem is not in Euclidean distance and the dimensionality in our problem is much higher, existing spatial database techniques are not applicable.



(a) Jaccard, TREC, Verifications per Record



(b) Jaccard, TREC-3GRAM and UNIREF-3GRAM, Upper Bound and s_k



(c) Jaccard, TREC-3GRAM and UNIREF-3GRAM, Running Time

Fig. 5. Experiment Results II

IX. Conclusions

In this paper, we study the problem of answering similarity join queries to retrieve top-k pairs of records ranked by their similarities. Existing approaches for the traditional similarity joins with a given threshold will have to make guesses on the similarity threshold and incur much redundant calculation. We propose an efficient algorithm that computes the answers in a progressive manner. The intrinsic monotonicity of upper bound score and the k-th temporary result score are exploited to develop several optimizations to improve the space and time efficiencies of the algorithm. The superiority of our proposed algorithm is demonstrated by extensive experiments on large-scale datasets under a wide range of parameter settings.

Acknowledgment Wei Wang is supported by ARC Discovery Grants DP0987273 and DP0881779. Xuemin Lin is supported by Google Research Award and ARC Discovery Grants DP0987557, DP0881035 and DP0666428.

References

[1] M. R. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *SIGIR*, 2006.
[2] W. W. Cohen, "Integration of heterogeneous databases without common domains using queries based on textual similarity," in *SIGMOD Conference*, 1998, pp. 201–212.
[3] W. E. Winkler, "The state of record linkage and current research problems," U.S. Bureau of the Census, Tech. Rep., 1999.
[4] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007.
[5] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006.
[6] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008.
[7] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006.
[8] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," in *SIGMOD Conference*, 2000, pp. 189–200.
[9] A. Corral, M. Vassilakopoulos, and Y. Manolopoulos, "The impact of buffering on closest pairs queries using r-trees," in *ADBIS*, 2001, pp. 41–54.
[10] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Algorithms for processing k-closest-pair queries in spatial databases," *Data Knowl. Eng.*, vol. 49, no. 1, pp. 67–104, 2004.
[11] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 1st ed. Addison Wesley, May 1999.
[12] C. Li, B. Wang, and X. Yang, "VGRAM: Improving performance of approximate queries on string collections using variable-length grams," in *VLDB*, 2007.
[13] H. Lee, R. T. Ng, and K. Shim, "Extending q-grams to estimate selectivity of string matching with low edit distance," in *VLDB*, 2007, pp. 195–206.
[14] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 9–37, 1998.
[15] S. Sarawagi and A. Bhamidipaty, "Interactive deduplication using active learning," in *KDD*, 2002.
[16] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "Adaptive name matching in information integration," *IEEE Intelligent Sys.*, vol. 18, no. 5, pp. 16–23, 2003.
[17] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *SIGMOD*, 2004.
[18] A. Arasu, S. Chaudhuri, and R. Kaushik, "Transformation-based framework for record matching," in *ICDE*, 2008, pp. 40–49.
[19] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, "A fast similarity join algorithm using graphics processing units," in *ICDE*, 2008, pp. 1111–1120.
[20] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks*, vol. 29, no. 8-13, pp. 1157–1166, 1997.
[21] A. Chowdhury, O. Frieder, D. A. Grossman, and M. C. McCabe, "Collection statistics for fast duplicate document detection," *ACM Trans. Inf. Syst.*, vol. 20, no. 2, pp. 171–191, 2002.
[22] M. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002.
[23] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.
[24] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *SIGMOD Conference*, 2003, pp. 313–324.
[25] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *VLDB*, 2001.
[26] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.
[27] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast indexes and algorithms for set similarity selection queries," in *ICDE*, 2008, pp. 267–276.
[28] E. Ukkonen, "On approximate string matching," in *FCT*, 1983.
[29] A. Z. Broder, "On the resemblance and containment of documents," in *SEQS*, 1997.
[30] R. C. Russell, "Index, U.S. patent 1,261,167," April 1918.
[31] R. Fagin, "Combining fuzzy information from multiple systems," *J. Comput. Syst. Sci.*, vol. 58, no. 1, pp. 83–99, 1999.
[32] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 614–656, 2003.
[33] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis, "Automated ranking of database query results," in *CIDR*, 2003.
[34] K. C.-C. Chang and S. won Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD Conference*, 2002, pp. 346–357.
[35] Y. Luo, X. Lin, W. Wang, and X. Zhou, "SPARK: top-k keyword query in relational databases," in *SIGMOD Conference*, 2007, pp. 115–126.
[36] M. Zhu, D. Papadias, J. Zhang, and D. L. Lee, "Top-k spatial joins," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 4, pp. 567–579, 2005.