Soft Computing (2020) 24:14577–14592
https://doi.org/10.1007/s00500-020-04807-w

METHODOLOGIES AND APPLICATION

SETJoin: a novel top-*k* similarity join algorithm

Hongya Wang[1] · Lihong Yang[1] · Yingyuan Xiao[2]

Published online: 6 March 2020

Abstract
As an important operation in data cleaning, near duplicate Web pages detection and data mining, similarity joins have received much attention recently. Existing similarity joins fall into two broad categories—the *similarity-threshold-based similarity join* and *top-k similarity join* (TopkJoin). Compared with the traditional one, TopkJoin is more suitable for cases where the similarity threshold is unknown before hand. In this paper, we focus on the performance optimization problem of TopkJoin. Particularly, we observed that the state-of-the-art TopkJoin algorithm has three serious performance issues, i.e., the inappropriate application of hash table, inefficient use of suffix filtering and unnecessary evaluation of excessive unqualified candidates. To resolve these problems, we proposed a novel algorithm, SETJoin, by combining the existing event-driven framework with three simple yet efficient optimization techniques, viz., (1) reducing the cost in hashing by rearranging the orders of the candidate filtering and hash table lookup operations; (2) maximizing the pruning capability of suffix filtering by judiciously choosing the (near) optimal recursion depth; and (3) terminating join operations earlier by setting a much tighter stop condition for iteration. The experimental results show that SETJoin achieves up to 1.26*x*–3.49*x* speedup over the state-of-the-art algorithm on several real datasets.

Keywords Set similarity join · Query processing · Candidate filtering

1 Introduction

Similarity joins have a wide range of applications in domains such as data cleaning (Hernández and Stolfo 1998), near duplicate Web page detection (Bayardo et al. 2007) and data mining (Baraglia et al. 2010). For example, recommendation algorithms often need to compute pair-wise similarity among users or items and then make a recommendation to users who share similar interests. In data cleaning tasks, similarity join can serve as a primitive operation to identify different (but similar) representations of the same entity.

Similarity joins have attracted much attention in recent years (Arasu et al. 2006; Xiao et al. 2008; Lam et al. 2010; Bayardo et al. 2007; Jiang et al. 2014; Chaudhuri et al. 2006; Li et al. 2015). Existing similarity joins fall into two broad categories—the *similarity-threshold-based similarity join* (SimJoin) and *top-k similarity join* (TopkJoin). SimJoin returns pairs of records whose similarities are no less than a user-specified similarity threshold, whereas TopkJoin computes the *k* most similar record pairs under the given similarity function.

SimJoin assumes that users can issue an appropriate similarity threshold, which is, however, not always available before hand, especially when one deals with a collection of records for the first time. To this end, TopkJoin is proposed to provide the most similar pairs in an alternative way and eliminate the guess work users have to do in terms of similarity threshold. Moreover, TopkJoin is able to support interactive applications by presenting the *k* most similar pairs progressively (Xiao et al. 2009; Kim and Shim 2012).

Answering TopkJoin queries involves many challenging issues. By using an innovative event-driven framework, Xiao et al. introduced topk-join, the state-of-the-art TopkJoin algorithm, to address the self-join problem over records in the form of sets (Xiao et al. 2009). Driven by the prefix event stream, topk-join performs join operation iteratively

Communicated by V. Loia.

✉ Hongya Wang
hywang@dhu.edu.cn

Yingyuan Xiao
yyxiao@tjut.edu.cn

1 School of Computer Science and Technology, Donghua University, Shanghai, China

2 School of Computer Science and Technology, Tianjin University of Technology, Tianjin, China

till the top-k similar pairs are found. The design principles and implementation details of topk-join will be discussed in Sect. 3.

We observed that topk-join has three serious performance issues, viz. the inappropriate application of hash table, inefficient use of suffix filtering and evaluating too much totally unqualified candidates. To resolve these problems, we proposed a novel TopkJoin algorithm, SETJoin, by combining the event-driven framework with three simple yet efficient optimization techniques, namely, Switching the positions of the candidate filtering and hash table lookup operations, Enhancing the pruning capability of suffix filtering by choosing the near optimal recursion depth and Terminating the iteration earlier by setting a much tighter stop condition. The technical contributions of this paper are summarized as follows.

- In topk-join, a candidate pair may be assembled multiple times. To ensure every candidate is evaluated only once, a hash table is consulted for each newly generated candidate pair. This design, however, imposes substantial overhead in performance, which will be discussed in Sect. 3. To address this problem, we proved that the *simplified pruning conditions* of positional and suffix filtering are applicable on all occasions no matter how many times a candidate pair has been assembled, based on which we eliminated the performance bottleneck by simply rearranging the orders of lookup and filtering operations.
- Suffix filtering is an efficient pruning technique introduced in Xiao et al. (2008), which recursively compares the Hamming distance of a candidate pair with the given maximum possible Hamming distance. Its pruning capability relies heavily on the recursion depth. The conventional wisdom is that the recursion depth should not be *too large*, and thus it is often set to a small integer, say 2, in existing algorithms (Jiang et al. 2014; Wang et al. 2012; Xiao et al. 2009, 2008). We made an observation that this widely practiced method underutilizes the pruning power of suffix filtering. To this end, we developed a cost model to identify the determining factors that affect the performance. Based on this model, a thumb rule is introduced for choosing the (near) optimal recursion depth, which suggests that, counter intuitively, the recursion depth should not be *too small*.
- In topk-join, the iteration terminates when the similarity of the kth most similar candidate pair seen so far is no less than the maximum similarity upper bound of the latest pending prefix event. While this stop condition is correct, we noticed that quite an amount of time is wasted in computing the similarities of unqualified candidates. To improve the performance, we devised a much tighter stop condition. Theoretical analysis shows that the new stop condition does not affect the correctness of the algorithm and is able to cease join operations much earlier than the existing one.
- We conducted comprehensive experiments to demonstrate the efficiency of the proposed algorithm. Experimental results show that SETJoin achieves up to $1.26x$–$3.49x$ speedup over the state-of-the-art algorithm on several real datasets. Moreover, as k increases and/or the average record lengths become longer, SETJoin will outperform topk-join by a larger margin.

Table 1 Two sets of records

r_1	{conf1, is, a, DB, conference}
s_1	{conf2, is, a, IR, conference}
r_2	{conf3, is, a, DB, workshop}
s_2	{conf4, is, a, IR, workshop}

The rest of the paper is organized as follows. Problem formulation and overview of several filtering techniques are given in Sect. 2. Section 3 reviews the state-of-the-art TopkJoin algorithm. The three optimization methods are discussed in detail in Sect. 4. Experimental studies are conducted in Sect. 5. We present the related work in Sect. 6 and conclude the paper in Sect. 7.

2 Preliminaries

2.1 Problem formulation

A record is represented as a set of tokens from a finite universe $\mathcal{U}$. Given two records r and s, a similarity function $\mathsf{sim}(r, s)$ returns a value that indicates the similarity of these two records. The larger the value is, the more similar the two records are. In this paper, we consider the following similarity functions.

Definition 1 Assume r and s are two records.

- Jaccard similarity: $\mathsf{sim}_J(r, s) = \frac{|r \cap s|}{|r \cup s|}$.
- Cosine similarity: $\mathsf{sim}_C(r, s) = \frac{|r \cap s|}{\sqrt{|r| \cdot |s|}}$.
- Dice similarity: $\mathsf{sim}_D(r, s) = \frac{2 \cdot |r \cap s|}{|r| + |s|}$.
- Overlap similarity: $\mathsf{sim}_O(r, s) = |r \cap s|$.

where $|r|$ denotes the size of r.

As a running example, consider the following two sets of records $\mathcal{R} = \{r_1, r_2\}$ and $\mathcal{S} = \{s_1, s_2\}$ illustrated in Table 1.

For r_1 and s_1, we have $|r_1| = 5$, $|s_1| = 5$ and $|r_1 \cap s_1| = 3$. It is easy to see that $\mathsf{sim}_J(r_1, s_1) = \frac{3}{7}$, $\mathsf{sim}_C(r_1, s_1) = \frac{3}{5}$, $\mathsf{sim}_D(r_1, s_1) = \frac{3}{5}$ and $\mathsf{sim}_O(r_1, s_1) = 3$.

Table 2 A global ordering according to DF

Word	Token	Doc. Freq.
conf1, conf2, conf3, conf4	e_1, e_2, e_3, e_4	1
conference	e_5	2
workshop	e_6	2
DB	e_7	2
IR	e_8	2
a	e_9	4
is	e_{10}	4

Table 3 Sorted tokenized records

r_1	$\{e_1, e_5, e_7, e_9, e_{10}\}$
s_1	$\{e_2, e_5, e_8, e_9, e_{10}\}$
r_2	$\{e_3, e_6, e_7, e_9, e_{10}\}$
s_2	$\{e_4, e_6, e_8, e_9, e_{10}\}$

Definition 2 (Top-k Similarity Join) Given two sets of records $\mathcal{R}$ and $\mathcal{S}$, a similarity function sim and a user-specified parameter k, a top-k similarity join returns the k most similar record pairs, that is, $\{\langle r, s\rangle_i | \langle r, s\rangle_i \in \mathcal{R}\times\mathcal{S}, 1 \leq i \leq k\}$ such that $\forall\langle r, s\rangle_i$ $s.t.$ $\mathsf{sim}(r, s) \geq \mathsf{sim}(rr, ss)$ where $\langle rr, ss\rangle \in \mathcal{R} \times \mathcal{S}\backslash\{\langle r, s\rangle_i\}$.

Given records listed in Table 1, Jaccard similarity sim_J and $k = 2$, the TopkJoin query returns $\{\langle r_1, s_1\rangle, \langle r_2, s_2\rangle\}$ since $\mathsf{sim}_J(r_1, s_1)$ and $\mathsf{sim}_J(r_2, s_2)$ are both equal to $\frac{3}{7}$, which are greater than $\mathsf{sim}_J(r_1, s_2) = \frac{1}{4}$ and $\mathsf{sim}_J(r_2, s_1) = \frac{1}{4}$.

To minimize the number of candidates, elements in $\mathcal{U}$ are often sorted according to the increasing order of their document frequencies (DF) (Bayardo et al. 2007; Xiao et al. 2009, 2008). Table 2 lists a global ordering of elements and tokens of all records. Table 3 shows the tokenized records which have been sorted based on the global ordering.

Similar to Xiao et al. (2009) and Bayardo et al. (2007), we concentrate on the self-join problem, viz., $\mathcal{R} = \mathcal{S}$, in this paper. Consider the inherent relation between different similarity functions in Definition 1, we focus on Jaccard similarity in the rest of this paper, unless otherwise stated.

2.2 Overview of several filtering techniques

Existing similarity join algorithms usually employ a filter-and-verification framework. In the candidate generation phase, the inverted indices are probed and a large number of candidate pairs are generated. To avoid computing the exact similarities of all candidate pairs, efficient filtering techniques are solicited to prune those that cannot be similar. Thereafter, the remaining candidates (verification pairs) are passed down to the final verification phase. In this section, we will give a brief overview of three prevailing filtering methods that are closely related to our work.

2.2.1 Prefix filtering

The basic idea of *prefix filtering* is that, instead of indexing all tokens for each record, one only needs to index and examine the first few tokens of each record during the candidate generation phase. In this way, the index size and the number of candidates can be both reduced dramatically. Lemma 1 gives a formal description of the prefix filtering principle.

Lemma 1 (Prefix Filtering Principle) *Consider a set of records, each of which consists of tokens sorted according to a global ordering $\mathcal{O}$ over the token universe $\mathcal{U}$. If $\mathsf{sim}_O(r, s) \geq t$, then the $(|r| - t + 1)$-$prefix$ of r and the $(|s| - t + 1)$-$prefix$ of s must share at least one common token, where the p-$prefix$ of r is the first p tokens of r.*

Consider records in Table 3 and suppose $t = 4$, the 2-$prefix$es are $\{e_1, e_5\}$, $\{e_2, e_5\}$, $\{e_3, e_6\}$ and $\{e_4, e_6\}$ for r_1, r_2, s_1 and s_2, respectively. Since the 2-$prefix$es of r_1 and s_2 (r_2 and s_1) have no common token, $\langle r_1, s_2\rangle$ ($\langle r_2, s_1\rangle$) can be pruned safely without checking the remaining tokens.

While the prefix filtering principle is formulated using Overlap similarity, extensions can be made to other similarity functions listed in Definition 1. Particularly, if $\mathsf{sim}(r, s) \geq \theta$, we can transform this constraint into its equivalent form using Overlap similarity as follows.

- If $\mathsf{sim}_J(r, s) \geq \theta$, then $|r \cap s| \geq |r| \cdot \theta$, thus $t = \lceil |r| \cdot \theta \rceil$.
- If $\mathsf{sim}_C(r, s) \geq \theta$, then $|r \cap s| \geq \sqrt{|r|} \cdot \theta$, thus $t = \lceil \sqrt{|r|} \cdot \theta \rceil$.
- If $\mathsf{sim}_D(r, s) \geq \theta$, then $|r \cap s| \geq |r| \cdot \frac{\theta}{2-\theta}$, thus $t = \lceil |r| \cdot \frac{\theta}{2-\theta} \rceil$.

2.2.2 Positional filtering

The similarity threshold mapping between Overlap and other similarity functions discussed in Sect. 2.2.1 is over pessimistic. During the index construction phase, this treatment is inevitable because, for any record r, we have no idea which other records r will be paired with. In the candidate generation phrase, however, one could obtain a larger similarity threshold by taking the lengths of both records into consideration. The refined similarity threshold mappings are as follows: (based on which a more powerful filtering technique, i.e., *positional filtering*)

- If $\mathsf{sim}_J(r, s) \geq \theta$, then $|r \cap s| \geq (|r| + |s|) \cdot \frac{\theta}{1+\theta}$, thus $t = \lceil (|r| + |s|) \cdot \frac{\theta}{1+\theta} \rceil$.
- If $\mathsf{sim}_C(r, s) \geq \theta$, then $|r \cap s| \geq \sqrt{|r| \cdot |s|} \cdot \theta$, thus $t = \lceil \sqrt{|r| \cdot |s|} \cdot \theta \rceil$.
- If $\mathsf{sim}_D(r, s) \geq \theta$, then $|r \cap s| \geq \frac{|r|+|s|}{2} \cdot \theta$, thus $t = \lceil \frac{|r|+|s|}{2} \cdot \theta \rceil$.

Positional filtering exploits the position information of the common token(s) between two records to further reduce the number of candidates. To elaborate the key idea of positional filtering, an illustrative example borrowed from Xiao et al. (2008) is given below.

Example 1 Consider two records r and s and suppose the similarity threshold is 0.8.

$$r = \{\underline{e_1}, \underline{e_2}, e_3, e_4, e_5\}$$
$$s = \{\underline{e_2}, \underline{e_3}, e_4, e_5, e_6\}$$

As discussed earlier, the refined overlap similarity threshold t for $\langle r, s\rangle$ is equal to $\lceil (5+5)\cdot \frac{0.8}{1+0.8}\rceil = 5$. Obviously, $\langle r, s\rangle$ does not meet this constraint and thus should be pruned as early as possible. However, since the prefixes of r and s (underlined tokens) share a common token, e_2, $\langle r, s\rangle$ will be still assembled during the candidate generation phrase. Please note that the prefix length here is determined by $\lceil |r|\cdot\theta\rceil$ instead of $\lceil (|r|+|s|)\cdot\frac{\theta}{1+\theta}\rceil$

If we look carefully into r and s, one can see that the position of the first common token e_2 actually matters in estimating the overlap similarity of $\langle r, s\rangle$. Specifically, the maximum possible overlap between r and s is the sum of the current overlap value and the minimum number of the unseen tokens between them, that is, $1 + min(3, 4) = 4$, which is less than $t = 5$. Therefore, $\langle r, s\rangle$ can be pruned by exploiting the position information of their tcommon tokens.

Lemma 2 generalizes the idea of positional filtering.

Lemma 2 (Xiao et al. 2008) *Consider a set of records, each of which consists of tokens sorted according to a global ordering $\mathcal{O}$ over the token universe $\mathcal{U}$. Let token $e = r[i]$, r is partitioned by e into the left part $r_l[e] = r[1\cdots i]$ and right part $r_r[e] = r[i+1\cdots |r|]$. If $\mathsf{sim}_O(r, s) \geq t$, then for every token $e \in r\cap s$, $\mathsf{sim}_O(r_l(e), s_l(e)) + \mathsf{min}(|r_r(e)|, |s_r(e)|) \geq t$.*

According to Lemma 2, given an Overlap similarity threshold t, a candidate pair can be pruned if Eq. (1) holds.

$$\mathsf{sim}_O(r_l(e), s_l(e)) + \mathsf{min}(|r_r(e)|, |s_r(e)|) < t \tag{1}$$

2.2.3 Suffix filtering

The candidate size might still grow quadratically with the number of records even if positional filtering is used. To further reduce the cost of exact similarity computation, *suffix filtering* is introduced in Xiao et al. (2008) based on two important insights.

The first insightful observation is that overlap constraint can be converted to the equivalent Hamming distance constraint as shown in Eq. (2).

$$\mathsf{sim}_O(r, s) \geq t \Longleftrightarrow \mathrm{dis}_H(r, s) \leq |r| + |s| - 2t \tag{2}$$

where $\mathrm{dis}_H(r, s)$ denotes the Hamming distance between r and s.

Let r_p and r_s denote the prefix and suffix of r. The Hamming distance of $\langle r_s, s_s\rangle$ can be bounded using Eq. (3) if $\mathsf{sim}_O(r, s) \geq t$.

$$\mathrm{dis}_H(r_s, s_s) \leq H_{\max} = |r| + |s| - 2t - (|r_p| + |s_p| - 2\mathsf{sim}_O(r_p, s_p)) \tag{3}$$

where $(|r|+|s|-2t)$ is the maximum possible Hamming distance of $\langle r, s\rangle$ and $(|r_p|+|s_p|-2\mathsf{sim}_O(r_p, s_p))$ is the exact Hamming distance of $\langle r_p, s_p\rangle$. For any $\langle r, s\rangle$, if $\mathrm{dis}_H(r_s, s_s)$ is greater than $H_{\max}$, it can be pruned safely.

The second observation is that the Hamming distance of $\langle r_s, s_s\rangle$ can be estimated efficiently in a recursive way. Figure 2 gives an illustrative example where $r=\{e_1, e_3, e_5, e_7, e_8\}$ and $s = \{e_2, e_3, e_4, e_5, e_6\}$. The estimation algorithm starts by choosing one record as the probing record. In Fig. 2, r is selected. Tokens in r can be examined one at a time in an arbitrary order. As shown in Fig. 2a, suppose the $3rd$ token in r (e_5) is chosen first. Using e_5, r is divided into two sections. Similarly, we divide s into two sections as well using e_5. While the contents of the left and right sections of r and s are still unknown (marked with "?"), the Hamming distance between r and s can be lower bounded by summing up the difference of the left sections and that of the right sections.

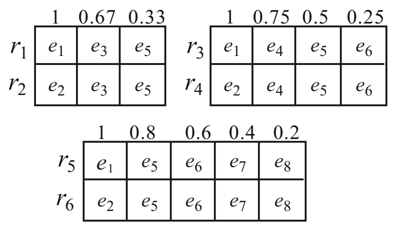
To improve the accuracy of estimation, one can probe the left and right sections of r and s recursively. As shown in Fig. 2b, e_7 splits the right section of r into an empty subsection (left) and a subsection with two tokens (right). Similarly, the right section of s is partitioned into a single-token subsection and an empty subsection. Having these subsections, the Hamming distance between the right sections of r and s is lower bounded by 3, which is much larger than 1 as shown in Fig. 2a.

Actually, if all tokens in r are probed in a similar vein, we can get the exact Hamming distance of r and s as shown in Fig. 2c. In a nutshell, this estimation method provides much higher estimation accuracy at the cost of larger probing and computation time.

3 Topk-Join review

In this section, we will review $\mathsf{topk\text{-}join}$, the state-of-the-art TopkJoin algorithm and discuss its performance issues.

$\mathsf{topk\text{-}join}$ uses an event-driven framework to perform join operations incrementally till the k most similar pairs are found. The key notion in this framework is the *prefix event*, which is defined as a 3-tuple (r, p_r, ub_{p_r}). For record r, p_r denotes the position of the token to be processed and



1 0.67 0.33
r_1 e_1 e_3 e_5
r_2 e_2 e_3 e_5
1 0.75 0.5 0.25
r_3 e_1 e_4 e_5 e_6
r_4 e_2 e_4 e_5 e_6
1 0.8 0.6 0.4 0.2
r_5 e_1 e_5 e_6 e_7 e_8
r_6 e_2 e_5 e_6 e_7 e_8

Fig. 1 Prefix events

ub_{p_r} represents the similarity upper bound between r and the remaining (unseen) records w.r.t. $r[p_r]$. Equation (4) shows how ub_{p_r} is calculated (Xiao et al. 2009).

$$ub_{p_r} = 1 - \frac{p_r - 1}{|r|} \tag{4}$$

Figure 1 illustrates the prefix events of six records r_1–r_6. For each token of a record, the number on top of it represents the associated similarity upper bound. Take r_1 as an example, the prefix events associated with e_1, e_3 and e_5 are $(r_1, 1, 1)$, $(r_1, 2, 0.67)$ and $(r_1, 3, 0.33)$, respectively.

Conceptually, topk-join processes prefix events as follows.

1. Sort all prefix events in descending order of ub_{p_r}.
2. Fetch the prefix event with the greatest ub_{p_r}. If $ub_{p_r} > \text{sim}_k$, probe the inverted index using $r[p_r]$ and generate candidates. Otherwise, the whole procedure terminates. sim_k is the similarity of the kth most similar pairs seen so far.
3. Remove the prefix event that has been processed from the sorted list and go back to Step 2.

The pseudo-code of topk-join is shown in Algorithm 1. The max-heap E_h is used to sort prefix events in descending order of ub_{p_r}. For each record r, the prefix event $(r, 1, 1)$ is added into E_h at the initialization stage (Line 1). Then, the prefix event with the highest ub_{p_r} is iteratively popped out (Line 3). Using the token in a prefix event, say (r, p_r, ub_{p_r}), the index is probed and all candidate pairs that share the common token $r[p_r]$ are assembled (Line 7).

Since the number of candidates is often prohibitively large, positional and suffix filtering are used to prune most of the unqualified candidates (Line 9). For candidates that cannot be pruned (verification pairs), their exact similarities are computed (Line 10). The min-heap Top_h is used to store the k most similar pairs that have been seen so far. The inverted index is updated to accommodate r in the inverted list of token $r[p_r]$ (Line 12). At the end of each iteration, the next prefix event of r is pushed onto E_h for further processing (Line 13–14). The whole iteration terminates if ub_{p_r} is no greater than the similarity of the kth element in Top_h (Line 5–6), which means that no more eligible candidates exist.

As an illustration, Fig. 3 depicts the main steps in evaluating the top-1 similarity join query over dataset shown in Fig. 1. Each step here consists of four operations, i.e., popup of a prefix event from E_h, maintaining the inverted index, updating the result set and pushing a new prefix event onto the max-heap. In Step 1 and Step 2, the result set is empty because no matching record can be found via index searching. Starting with step 3, new candidate pairs are gradually generated and the similarity value of the most similar pair keeps increasing. The whole procedure stops in step 11, where ub_{p_r} of the prefix event $(r_2, 2, 0.67)$ is equal to the similarity of $\langle r_5, r_6 \rangle$.

In topk-join, the similarity of a verification pair might be computed multiple times if these two records share more than one common token. To address this problem, Algorithm 1 uses a hash table $\mathcal{H}$ to avoid the repeated evaluation of the same verification pair. Particularly, for each candidate pair $\langle r, s \rangle$, $\mathcal{H}$ is consulted first to see whether $\langle r, s \rangle$ has been processed (Line 8). The further processing is carried out only if $\langle r, s \rangle$ appears for the first time (Line 10).

Although topk-join has employed a few optimization techniques to reduce the number of candidates and avoid repeated verifications, we observed that there were three serious performance problems in Algorithm 1.

1. While being introduced to improve the performance, the hash table itself actually becomes a serious performance bottleneck.
2. The pruning capability of suffix filtering has not been fully exploited.
3. The stop condition is not tight enough, and too much time is wasted in evaluating totally unqualified candidates.

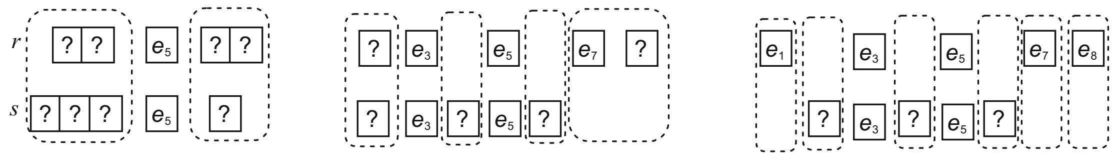
In the next section, we will discuss the main causes of these problems and provide three simple yet efficient solutions to improve the performance.

4 Optimizing top-*k* similarity join

In this section, we present a novel top-k similarity join algorithm, SETJoin, by combining the event-driven framework with three optimization techniques, which will be discussed in Sects. 4.1, 4.2 and 4.3, respectively.

4.1 Switch the positions of hash table lookup and filtering operations

Recall that topk-join utilizes a hash table to guarantee that each verification pair is evaluated only once. Although it seems that the performance could be improved by eliminating



(a) The Hamming distance is estimated as (3-2)+(2-1)=2 in the first round **(b)** The Hamming distance is estimated as (1-1)+(1-0)+(1-0)+(2-0)=4 in the second round **(c)** The Hamming distance is estimated as 1+1+1+(1-0)+(2-0)=6 in the third round

Fig. 2 An illustrative example of suffix filtering

Events	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Step9	Step10	Step11
E_h.pop( )	$(r_1, 1, 1)$	$(r_2, 1, 1)$	$(r_3, 1, 1)$	$(r_4, 1, 1)$	$(r_5, 1, 1)$	$(r_6, 1, 1)$	$(r_5, 2, 0.8)$	$(r_6, 2, 0.8)$	$(r_3, 2, 0.75)$	$(r_4, 2, 0.75)$	$(r_1, 2, 0.67)$
Inverted Index	$e_1 \rightarrow \{r_1\}$	$e_1 \rightarrow \{r_1\}$ $e_2 \rightarrow \{r_2\}$	$e_1 \rightarrow \{r_1 r_3\}$ $e_2 \rightarrow \{r_2\}$	$e_1 \rightarrow \{r_1 r_3\}$ $e_2 \rightarrow \{r_2 r_4\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4 r_6\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4 r_6\}$ $e_5 \rightarrow \{r_5\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4 r_6\}$ $e_5 \rightarrow \{r_5 r_6\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4 r_6\}$ $e_4 \rightarrow \{r_3\}$ $e_5 \rightarrow \{r_5 r_6\}$	$e_1 \rightarrow \{r_1 r_3 r_5\}$ $e_2 \rightarrow \{r_2 r_4 r_6\}$ $e_4 \rightarrow \{r_3 r_4\}$ $e_5 \rightarrow \{r_5 r_6\}$	
Result			$<r_1,r_3,0.4>$	$<r_1,r_3,0.4>$	$<r_3,r_5,0.5>$	$<r_3,r_5,0.5>$	$<r_3,r_5,0.5>$	$<r_5,r_6,0.67>$	$<r_5,r_6,0.67>$	$<r_5,r_6,0.67>$	
E_h.push( )	$(r_1, 2, 0.67)$	$(r_2, 2, 0.67)$	$(r_3, 2, 0.75)$	$(r_4, 2, 0.75)$	$(r_5, 2,0.8)$	$(r_6, 2, 0.8)$	$(r_5, 3, 0.6)$	$(r_6, 3, 0.6)$	$(r_3, 3, 0.5)$	$(r_4, 3, 0.5)$	

Fig. 3 An illustrative example of how topk-join works

```
Algorithm 1: Topk-Join($\mathcal{R}$,$k$)
Input: $\mathcal{R}$ is a collection of records; $k$ is a user-specified parameter
Output: Top-$k$ similar pairs
1  Initialize prefix event and result heaps $E_h$ and $Top_h$;
2  while $E_h \neq \emptyset$ do
3      $(r, p_r, ub_{p_r}) \leftarrow E_h.pop()$;
4      $sim_k = Top_h[k].sim$;
5      if $ub_{p_r} \leq sim_k$ then
6          break;
7      Generate candidate pairs $\langle r, s\rangle$ via index searching;
8      if $\langle r, s\rangle \notin \mathcal{H}$ then
9          Perform positional and suffix filtering;
10         Compute the similarity of $\langle r, s\rangle$;
11         $\mathcal{H} \leftarrow \mathcal{H} \cup \langle r, s\rangle$;
12         Update the inverted index and $Top_h$;
13         Calculate new similarity upper bound $ub_{p_r}$ for $r$;
14         $E_h.push(r, p_r + 1, ub_{p_r})$
```

the repeated computation, the fact is that the hash table itself becomes a new performance bottleneck.

Figure 4 compares the total running time and the cost of hash table operations by running a top-500 similarity join query over TREC dataset.[1] The hashing cost is obtained by performing insertion and lookup operations alone using the real trace collected during the query execution. As depicted in Fig. 4, the overhead in hashing accounts for more than 20% of the total query time.

We found that this problem is mainly due to the huge number of lookup operations. Particularly, topk-join often generated a great many candidates[2] and, for every assembled candidate, the hash table has to be consulted once in Algorithm 1. Despite the overhead for performing a single lookup is cheap, the total hashing cost could become unbearable if the number of lookup operations is prohibitively large.

Thanks to positional and suffix filtering, only a small fraction of candidates (2.4 million in the aforementioned example) could enter the final verification phase . In other words, most of candidates have no chance to become the verification pairs. This observation suggests that if the filtering operation is performed before the lookup operation, i.e., swapping Line 8 and Line 9 in Algorithm 1, a vast majority of lookup operations could be avoided and the hashing cost would be reduced dramatically.

Although the idea is simple, moving the filtering operation forward is not as trivial as it appears. To be specific, recall that a candidate pair can be pruned by positional filtering if Eq. (1) is satisfied (Lemma 2). In topk-join, Eq. (1) is replaced with Eq. (5) in which $\mathrm{sim}_O(r_l(e), s_l(e))$, the overlap of the left parts of r and s, is set to 1. This *simplified pruning condition* is correct here because (1) only the first instance of $\langle r, s\rangle$ can enter the filtering stage in Algorithm 1, and (2) $\mathrm{sim}_O(r_l(e), s_l(e))$ is equal to 1 when $\langle r, s\rangle$ is generated for the first time.[3]

$$1 + \min(|r_r(e)|, |s_r(e)|) < t \tag{5}$$

[1] Will be discussed in Sect. 5 in more detail.

[2] For instance, during the execution of the top-500 query, over *two hundred million* candidate pairs are generated.

[3] We do not present the details of prefix and positional filtering in Algorithm 1 for the sake of conciseness.

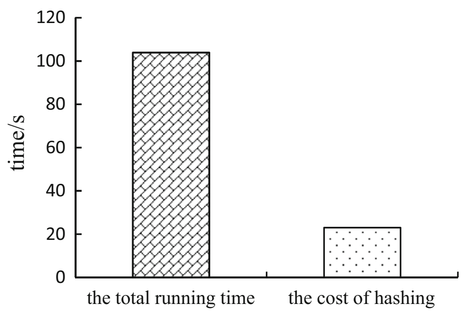


Fig. 4 The running time versus the cost of hashing

Provided positional filtering was switched ahead of the lookup operation, we would have to resort to Eq. (1) again because it is general enough to handle the mth duplicate of $\langle r, s\rangle$ $(m > 1)$. Assume Eq. (1) is employed, then $\mathsf{sim}_O(r_l(e), s_l(e))$ of *ALL* candidates must be either computed and stored or calculated multiple times in order to evaluate this inequality, which will definitely degrade the performance considering the huge number of candidate pairs.

Fortunately, we have the following observation that can help us out of the dilemma.

Lemma 3 *In positional filtering, using Eq.* (5) *as the filtering condition provides correct answers on all occasions no matter how many times a candidate pair is assembled, and Eq.* (5) *is actually more efficient than Eq.* (1) *in pruning capability.*

Proof 1 We first prove the case in which a candidate pair appears for the first time. Suppose e is the first common token shared by r and s, then $\mathsf{sim}_O(r_l(e), s_l(e)) = 1$ and the maximum possible overlap of $\langle r, s\rangle$ is equal to $1+\mathsf{min}(|r_r(e)|, |s_r(e)|)$. It is direct from Lemma 2 that $\langle r, s\rangle$ can be pruned if Eq. (5) holds. Please note that in topk-join $t = \lceil(|r| + |s|)\cdot\frac{sim_k}{1+sim_k}\rceil$ as the exact threshold is unknown before the algorithm terminates.

The mth duplicate of $\langle r, s\rangle$ $(m > 1)$ could be pruned only if $m + \mathsf{min}(|r_r(e)|, |s_r(e)|)$ is less than t according to Lemma 2, where e is the mth common token. As $m + \mathsf{min}(|r_r(e)|, |s_r(e)|)$ is always greater than $1 + \mathsf{min}(|r_r(e)|, |s_r(e)|)$, candidates satisfying Eq. (1) will definitely be pruned when Eq. (5) is used. In other words, Eq. (5) is more strict than Eq. (1).

Equation (5) may prune eligible candidates. This, however, has no impact on the correctness of final results because it doesn't matter whether the mth duplicate of $\langle r, s\rangle$ is wrongly handled as long as its first instance is processed correctly. The by-product of using Eq. (5) as the pruning condition is that more candidate pairs could be pruned compared with Eq. (1). We prove the Lemma. □

Similarly, Lemma 4 shows that, for suffix filtering, Eq. 6 (as a replacement of Eq. 3) is also applicable in all cases.

$$dis_H(r_s, s_s) > H_{\max} = |r| + |s| - 2t - (|r_p| + |s_p| - 2) \quad (6)$$

Lemma 4 *In suffix filtering, using Eq.* (6) *as the filtering condition provides correct answers on all occasions no matter how many times a candidate pair is assembled, and Eq.* (6) *is actually more efficient than Eq.* (3) *in pruning capability.*

Proof 2 The proof strategy is similar to that used in Lemma 3. Suppose e is the first common token in the prefixes of r and s, then $\mathsf{sim}_O(r_p, s_p) = 1$ and the maximum possible Hamming distance between r and s is $|r| + |s| - 2t - (|r_p| + |s_p| - 2)$, where $t = \lceil(|r| + |s|)\cdot\frac{sim_k}{1+sim_k}\rceil$. As discussed in Sect. 2.2.3, if the estimated Hamming distance is greater than this upper bound (Eq. (6) holds), it can be pruned safely.

The mth duplicate of $\langle r, s\rangle$ $(m > 1)$ could be pruned only if the estimated Hamming distance is greater than $|r| + |s| - 2t - (|r_p| + |s_p| - 2m)$ according to Eq. (3). Since $|r| + |s| - 2t - (|r_p| + |s_p| - 2)$ is less than $|r| + |s| - 2t - (|r_p| + |s_p| - 2m)$, it will be much easier for a candidate to be pruned using Eq. (6) instead of Eq. (3). The correctness of suffix filtering, however, is still guaranteed because the top-k similar pairs will definitely be included in the final result set as long as the first instance of $\langle r, s\rangle$ is correctly processed. We prove this lemma. □

Lemmas 3 and 4 suggest that, while the simplified pruning conditions (Eqs. (5) and (6)) are only correct for the first instance of a candidate pair theoretically, they actually work very well on all occasions and are even more efficient than Eqs. (1) and (3) in pruning power. With the help of these two lemmas, we can simply switch the positions of filtering and hash table lookup operations in Algorithm 1 without any concern of the correctness issues.

4.2 Enhance the pruning capability of suffix filtering

Suffix filtering is an efficient pruning method introduced in Xiao et al. (2008). As discussed in Sect. 2.2.3, by comparing the estimated Hamming distance of the suffixes of $\langle r, s\rangle$ and $H_{\max}$ calculated using Eq. (6), one may determine whether $\langle r, s\rangle$ should be discarded.

The pseudo-code of suffix filtering is sketched in Algorithm 2.[4] The function Partition splits a record into two partitions (Lines 4–5), and the estimated Hamming distance H in current recursion level is calculated in Line 6. If the Hamming distance is large enough to prune $\langle r, s\rangle$, H is returned (Line 7–8). Otherwise, SuffixFilter is invoked recursively with smaller $H_{\max}$ and greater d (Line 10–13). There

[4] Please note that the suffixes of two records are passed to r and s when SuffixFilter is invoked

Algorithm 2: SuffixFilter(r, s, $H_{\max}$, d)

```
    Input: $r$ and $s$ are two records; $H_{\max}$ is the upper bound of
           Hamming distance; $d$ is the current recursive depth
    Output: The estimated Hamming distance between $r$ and $s$
 1  if $d >$ maxdepth then
 2      return abs($|r| - |s|$);
 3  $mid \leftarrow \lceil \frac{|s|}{2} \rceil$; $e \leftarrow s[mid]$ ;
 4  $(s_l, s_r, diff) \leftarrow$ Partition($s, e$) ;
 5  $(r_l, r_r, diff) \leftarrow$ Partition($r, e$) ;
 6  $H \leftarrow$ abs($|r_l| - |s_l|$) + abs($|r_r| - |s_r|$) + $diff$ ;
 7  if $H > H_{\max}$ then
 8      return $H$
 9  else
10      $temp_1 = H_{\max}$ − abs($|r_l| - |s_l|$) − $diff$ ;
11      $temp_2 = H_{\max}$ − abs($|r_r| - |s_r|$) − $diff$ ;
12      $H_l \leftarrow$ SuffixFilter($r_l, s_l, temp_2, d + 1$);
13      $H_r \leftarrow$ SuffixFilter($r_r, s_r, temp_1, d + 1$);
14      return $H_l + H_r + diff$
```

are two stop conditions for the recursion: (1) the estimated Hamming distance is greater than $H_{\max}$ (Line 7); (2) the depth of recursion is greater than the given threshold maxdepth (Line 1–2).

Since $H_{\max}$ is fixed for a given candidate pair, the pruning capability of suffix filtering relies heavily on the recursion depth maxdepth. The conventional wisdom is that the recursion depth should not be *too large* because, intuitively, the larger maxdepth is, the more running time filtering operations will cost. For instance, in Jiang et al. (2014), Wang et al. (2012), Xiao et al. (2009) and Xiao et al. (2008), it is stated that "for efficiency reason maxdepth is set to 2" without any justification or explanation.

Although this simple rule works in some cases, we believe that an in-depth analysis will be more helpful in understanding how and why the performance of topk-join varies with maxdepth. To this end, we introduced a cost model to quantitatively analyze which factors in suffix filtering really matters. Based on this model, we presented a counter intuitive guiding principle, with the aid of which the (near) optimal maxdepth could be obtained.

Given a set of candidate pairs, let C_f^i, C_e, N_f^i and N_r^i denote the average filtering cost, the average cost for calculating the exact similarity, the number of pruned candidates and the number of remaining candidates at the ith level of recursion, respectively. The total cost of suffix filtering and exact similarity computation in the case of maxdepth $= x$ can be formulated as follows.

$$C_T^x = \sum_{i=1}^{x-1} N_f^i C_f^i + N_r^x (C_f^x + C_e) \quad (7)$$

The difference between C_T^{x-1} and C_T^x indicates whether the overall running time can be reduced if the recursion depth increases by 1. To compute $C_T^{x-1} - C_T^x$, we first need to prove the following lemma.

Table 4 Sample statistics collected while running ppjoin+ over TREC dataset with $\theta = 0.8$

x	N_f^x	N_r^x	C_T^x (s)
2	1,844,993	354,211	9.106
3	344,203	10,008	8.552
4	9358	623	8.545
5	99	524	8.640

Lemma 5 *Given a fixed set of candidate pairs, C_f^i and N_f^i (if defined) are invariable for invocations of SuffixFilter with different maxdepth.*

Proof 3 The argument is true because SuffixFilter is deterministic and the stop condition (Line 7–8) guarantees that, for any prunable candidate pair, SuffixFilter will terminate at the same recursion level no matter how much maxdepth is. As a result, C_f^i and N_f^i do not vary with maxdepth. □

With Lemma 5 and the fact $N_r^{i-1} = N_f^i + N_r^i$ as per definition, Eq. (7) can be calculated as follows.

$$\begin{aligned} &C_T^{x-1} - C_T^x \\ &= N_r^{x-1}(C_f^{x-1} + C_e) - N_f^x C_f^x - N_r^x (C_f^x + C_e) \\ &= N_f^x (C_f^{x-1} + C_e) + N_r^x (C_f^{x-1} + C_e) \\ &\quad - N_f^x C_f^x - N_r^x (C_f^x + C_e) \\ &= N_f^x (C_e + C_f^{x-1} - C_f^x) - N_r^x (C_f^x - C_f^{x-1}) \end{aligned} \quad (8)$$

Equation (8) suggests that, as maxdepth increases, the overall running time would drop if $N_f^x (C_e + C_f^{x-1} - C_f^x)$ is greater than $N_r^x (C_f^x - C_f^{x-1})$. Otherwise, the performance would become even worse.

Since $(C_e + C_f^{x-1} - C_f^x)$ and $(C_f^{k+1} - C_f^k)$ are constant for a given set of candidate pairs, how to choose an optimal maxdepth solely depends on N_f^x and N_r^x. As an illustration, Table 4 depicts the overall running time, N_f^x and N_r^x while running ppjoin+[5] over TREC dataset with a similarity threshold of 0.8.

As we can see, C_T^x keeps decreasing with x until $x = 4$. This is because N_f^x is far greater than N_r^x when x falls in between 1 and 4. In the case of x of 5, N_f^x is much less than N_r^x. Therefore, increasing the recursion depth would not help improving the performance according to Eq. (8). In addition, it easy to see that $|\ C_T^{x-1} - C_T^x\ |$ becomes very small as x grows. The reason is that N_f^x and N_r^x both decline rapidly

[5] ppjoin+ is the state-of-the-art SimJoin algorithm proposed in Xiao et al. (2008).

Table 5 Sample statistics collected while running topk-join over TREC dataset with $k = 500$

x	N_f^x	N_r^x	C_T^x (s)
2	8,572,295	4,173,994	82.382
3	3,178,453	995,541	62.542
4	897,702	97,839	55.904
5	86,837	11,002	55.374
6	4266	6736	55.513
7	322	6414	55.564

with x, which renders that $| C_T^{x-1} - C_T^x |$ decreases in a similar way. For example, in the case of $k \geq 4$, $| C_T^{x-1} - C_T^x |$ is less than 0.1 second. All these observations prove the validity of our cost model.

When we turn to topk-join, the picture is somewhat different. As show in Table 5, where statistics are collected by running topk-join with $k = 500$ over TREC dataset, N_f^x and N_r^x are much greater than their counterparts in Table 4. Please note that this comparison is reasonable because the similarity of the $500th$ most similar record pair is approximately equal to 0.8.

The reasons for such a remarkable difference are two folds: (1) In topk-join, the same candidate might be assembled multiple times, and (2) At the early stage of the execution of topk-join, sim_k is far less than the similarity of the kth most similar pair, which makes that the overlap threshold $t = \lceil (|r| + |s|) \cdot \frac{sim_k}{1+sim_k} \rceil$ is smaller than how much it should be. As a result, the pruning capabilities of positional and suffix filtering are crippled in this case.

Table 5 indicates that the pruning power of suffix filtering has not been fully exploited considering maxdepth is only set to 2 in topk-join. Theoretically, the optimal maxdepth can be obtained by comparing C_T^{x-1} and C_T^x for all possible recursion depths. This method, however, is impractical because we cannot collect all necessary statistics needed in Equation (8) without running SuffixFilter first. Moreover, the performance of suffix filtering depends on k, which makes it hard to find a global optimal maxdepth.

In our preliminary experiments, which is exemplified in Table 5, we observed that

- $C_T^{x_{\text{opt}}}$ is almost equal to the optimal total running time, where x_{opt} is the recursion depth in which N_f^x is less than N_r^x for the first time (x_{opt} is equal to 6 in Table 5).
- C_T^x grows very slowly with the increase in the recursion depth after the total running time reaches the minimum.
- The differences between the optimal maxdepth for different k that we experimented with are very small (no greater than 2).

Please note that these experimental results agree with the proposed cost model since N_f^x and N_r^x are very small for large x, which means neither pruning candidates nor computing the exact similarities would not cost much time. Based on these observations, we could use the thumb rule below to find the (near) optimal maxdepth in practice.

Rule of Thumb When a TopkJoin query is issued over a dataset for the first time, one could set a *large* maxdepth, say 10, and then record the corresponding x_{opt}. For successive queries, $maxdepth$ can be simply set to $x_{\text{opt}} + 1$.

4.3 Terminate the iteration earlier

Recall that topk-join utilizes an event-driven framework to perform the join operation incrementally and terminates if the similarity upper bound of the latest pending prefix event is no greater than the similarity of the kth most similarity pair seen so far. While the stop condition is correct, we observed that the time instant sim_k reaches its maximum is much earlier than the time in which the stop condition is satisfied. In other words, a large amount of time is wasted in checking totally unqualified candidates. This observation leads us to ask whether the stop condition could be somehow tightened.

Next, we will introduce a new stop condition that is able to terminate the algorithm earlier. Our findings are motivated by an observation due to Xiao et al. (2009).

Observation 1 (Xiao et al. 2009) *Given two arbitrary records r and s, suppose the similarity upper bound associated with $r[p_r]$ ($s[p_s]$) is ub_{p_r} (ub_{p_s}), then the following inequality holds*

$$\text{sim}_J(r, s) = \frac{|r \cap s|}{|r \cup s|} \leq \frac{ub_{p_r} ub_{p_s}}{ub_{p_r} + ub_{p_s} - ub_{p_r} ub_{p_s}} \tag{9}$$

The right hand of Eq. (9) is called the *accessing similarity upper bound*, which has two important properties: (1) it decreases monotonically with ub_{p_r} (ub_{p_s}) when ub_{p_s} (ub_{p_r}) is fixed; (2) its value is no less than ub_{p_r} (ub_{p_s}).

With these properties, we consider to replace ub_{p_r} in Algorithm 1 (Line 4) with $\frac{ub_{p_r} ub_{\max}}{ub_{p_r} + ub_{\max} - ub_{p_r} ub_{\max}}$, where $ub_{\max}$ is the maximum ub_{p_s} seen so far. Theorem 1 shows that this replacement has no effect on the correctness of the event-driven framework.

Theorem 1 *Assume $\langle r, p_r, ub_{p_r} \rangle$ is the latest pending prefix event and $ub_{\max}$ is the greatest similarity upper bound seen so far. Substituting $\frac{ub_{p_r} ub_{\max}}{ub_{p_r} + ub_{\max} - ub_{p_r} ub_{\max}}$ for ub_{p_r} in Algorithm 1 (Line 4) does not affect the correctness of this algorithm.*

Proof 4 As discussed in Sect. 3, all prefix events are popped out from the max-heap in descending order of their similarity upper bounds. If the new condition holds, i.e.,

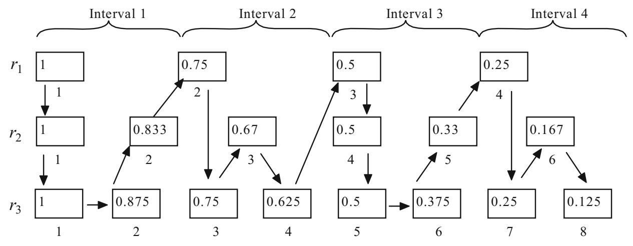


Fig. 5 An illustration of how to choose smaller $ub_{\max}$

$\frac{ub_{p_r} ub_{\max}}{ub_{p_r}+ub_{\max}-ub_{p_r} ub_{\max}} \le \text{sim}_k$, then for every upcoming candidate pair $\langle rr, ss\rangle$ we have $\frac{ub_{p_{rr}} ub_{p_{ss}}}{ub_{p_{rr}}+ub_{p_{ss}}-ub_{p_{rr}} ub_{p_{ss}}} \le \frac{ub_{p_r} ub_{\max}}{ub_{p_r}+ub_{\max}-ub_{p_r} ub_{\max}}$ according to the aforementioned two properties and the fact that $ub_{p_{rr}}$ and $ub_{p_{ss}}$ are both no greater than ub_{p_r} and $ub_{\max}$. As a result, we have $\frac{ub_{p_{rr}} ub_{p_{ss}}}{ub_{p_{rr}}+ub_{p_{ss}}-ub_{p_{rr}} ub_{p_{ss}}} \le \text{sim}_k$, which implies $\text{sim}_J(rr, ss) \le \text{sim}_k$ by Observation 1. We proved the theorem. □

The efficiency of the new stop condition is determined by how small $ub_{\max}$ could be. For example, if we take all processed prefix events into account, $ub_{\max}$ will be equal to 1, which renders $\frac{ub_{p_r} ub_{\max}}{ub_{p_r}+ub_{\max}-ub_{p_r} ub_{\max}} = ub_{p_r}$ and thus no performance gain could be achieved.

We observed that, however, using a smaller $ub_{\max}$ is feasible. Take Fig. 5 as an example, there are three records r_1, r_2 and r_3 of lengths 4, 6 and 8, respectively. Each rectangle represents a prefix event and the number in it gives the associated similarity upper bound. The lines with arrows show in which order these prefix events are processed. In view of the length of r_1 being 4, we divide the similarity range of [0,1] into four equally spaced intervals, i.e., [1,0.75),[0.75,0.5), [0.5,0.25), [0.25,0]. In each interval, one can see that, for every $r_i, i = 1, 2, 3$, at least one prefix event is processed and r_1 always owns the maximum similarity upper bound.

This observation suggests that we don't have to use the globally maximal similarity upper bound, i.e, 1, in Eq. (9). Instead, a local maximum will suffice for all upcoming prefix events. To be specific, the latest $ub_{p_{r_1}}$ could serve as $ub_{\max}$ for the corresponding interval. For example, suppose the prefix event $(r_2, 4, 0.5)$ is under processing. Since it lies in the third interval, we could let $ub_{\max} = 0.5$. By Eq. (9), the new stop condition would be $0.33 \le \text{sim}_k$, which is much tighter than the original one, i.e., $0.5 \le \text{sim}_k$. The following theorem generalizes the preceding discussion.

Theorem 2 *Given a set of records, suppose r is of the shortest length and ub_{p_r} comes from the most recently processed prefix event of r, using ub_{p_r} as $ub_{\max}$ in the stop condition guarantees the correctness of the algorithm.*

Proof 5 Obviously, the argument is true for the first interval. Suppose $\langle r, p_r, ub_{p_r}\rangle$ and $\langle s, p_s, ub_{p_s}\rangle$ lie in the mth interval ($m > 1$), and $\langle s, p_s, ub_{p_s}\rangle$ is under processing. For any prefix events $\langle rr, p_{rr}, ub_{p_{rr}}\rangle$ and $\langle rr, p_{rr}, ub_{p_{rr}}\rangle$ coming after $\langle s, p_s, ub_{p_s}\rangle$, $\frac{ub_{p_{rr}} ub_{p_{ss}}}{ub_{p_{rr}}+ub_{p_{ss}}-ub_{p_{rr}} ub_{p_{ss}}}$ is no greater than $\frac{ub_{p_r} ub_{p_s}}{ub_{p_r}+ub_{p_s}-ub_{p_r} ub_{p_s}}$ because all prefix events are processed in decreasing order of their associated similarity upper bounds. As ub_{p_r} is the local maximum in each interval, we have $\text{sim}_J(rr, ss) \le \text{sim}_k$ if $\frac{ub_{p_r} ub_{p_s}}{ub_{p_r}+ub_{p_s}-ub_{p_r} ub_{p_s}} \le \text{sim}_k$ (the stop condition holds). Please note that the prefix event of any record in the dataset appears at least once in each interval. Therefore, the above analysis is general enough to cover all possible candidates that will be paired during the mth interval. We proved this theorem. □

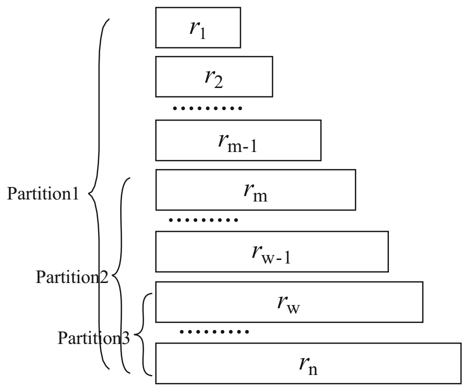


Fig. 6 A sample dataset with three partitions

There is still one problem to solve before we could enjoy the benefit brought by the new stop condition: how do we deal with datasets in which the minimal record lengths are very short? For example, if the size of the shortest record, say r, is 1, $ub_{\max}$ will be always equal to 1.

Our solution requires that all records are ordered in the ascending order of their lengths, which can be easily done during the preprocessing phase (Bayardo et al. 2007). Then, the records are divided (conceptually) into multiple overlapped partitions as shown in Fig. 6. Starting with the record with the largest size (r_n in this example), each small partition is a subset of a large partition, which then is contained by a larger partition, and so on and so forth. In each partition, $ub_{p_{r_i}}$ of the most recently processed prefix event out of the shortest record, e.g., r_1, r_m and r_w in Fig. 6, is used as $ub_{\max}$ for all records in this partition. For every candidate pair $\langle r, s\rangle$, we first need to find the minimal partition r and s belong to by comparing their ids with those of r_1, r_m and r_w, and then evaluate the stop conditions of the corresponding partitions. Put it another way, we use multiple $ub_{\max}$ and stop conditions for candidates in different partitions. Once the stop condition of some partition is satisfied, all upcoming candidates belonging to this partition could be pruned.

In our preliminary experiments, we noticed that the number of partitions should not be too large since the cost of checking which partition a candidate belongs to rises with the increase in the number of partitions. Experiment results show that splitting the dataset into 4 partitions achieves the best performance in most cases.

By combining these three optimizations with the existing event-driven framework, we propose a novel top-*k* similarity join algorithm SETJoin. As shown in Algorithm 3, the hash table lookup and filtering operations are switched (Line 10–11), the depth of recursion in suffix filtering is set to the (near) optimal value using the thumb rule (Line 10), and a new stop condition is employed (Line 7). For clarity of presentation, we do not show the pseudo-code for the dataset partitioning approach.

Algorithm 3: SETJoin($\mathcal{R}$,k)

```
Input: $\mathcal{R}$ is a collection of records sorted by the ascending order
       of their lengths; $k$ is a user-specified parameter
Output: Top-$k$ similar pairs
1  Initialize prefix event and result heaps $E_h$ and $Top_h$;
2  while $E_h \neq \emptyset$ do
3      $(r, p_r, ub_{p_r}) \leftarrow E_h.pop()$;
4      if $r = r_1$ then
5          $ub_{\max} = ub_{p_r}$;
6      $\text{sim}_k = Top_h[k].\text{sim}$;
7      if $\frac{ub_{p_r} ub_{\max}}{ub_{p_r} + ub_{\max} - ub_{p_r} ub_{\max}} \leq \text{sim}_k$ then
8          break;
9      Generate candidate pairs $\langle r, s\rangle$ via index search;
10     Perform positional filtering and suffix filtering with the (near)
       optimal maxdepth ;
11     if $\langle r, s\rangle \notin \mathcal{H}$ then
12         Compute the similarity of $\langle r, s\rangle$;
13         $\mathcal{H} \leftarrow \mathcal{H} \cup \langle r, s\rangle$;
14         Update the inverted index and $Top_h$;
15         Calculate new similarity bound $ub_{p_r}$ for $r$;
16         $E_h.push(r, p_r + 1, ub_{p_r})$
```

5 Experimental study

We conduct extensive experiments to verify the efficiency of SETJoin in comparison with the state-of-the-art algorithm.

5.1 Experiment setup

The following algorithms are implemented and compared.

topk-join is the state-of-the-art TOPKJOIN algorithm proposed in Xiao et al. (2009). Similar to Xiao et al. (2009), maxdepth is set to 2 for DBLP, TREC and ENRON-3GRAM datasets, and 4 for TREC-3GRAM dataset.

SETJoin(v1∼v3) are three algorithms implemented to clearly demonstrate the efficiency of the three optimization techniques. Particularly, hash optimization is enabled in SETJoin-v1 and maxdepth is set to the (near) optimal value in SETJoin-v2. In SETJoin-v3, all three optimizations are incorporated in.

Table 6 Dataset statistics

Datasets	Size	avg_len	max_len	min_len
DBLP	861,567	14.3	284	3
TREC	345,969	114.4	609	24
TREC-3GRAM	345,969	387.8	1202	50
ENRON-3GRAM	245,557	524.1	24,592	3

We performed the experiments with the following publicly available datasets. DBLP is a snapshot of the bibliography records from the DBLP Web site.[6] It contains about 0.9M records, each of which is a concatenation of author name(s) and the title of a publication. TREC is from TREC-9 Filtering Track Collections.[7] It contains 0.35M references from the MEDILINE database. We extract and concatenate author, title and abstract fields to form the records. ENRON-3GRAM is from the Enron email collection.[8] It contains about 0.25M emails from about 150 users, mostly senior management of Enron. We tokenized it into 3-g to form ENRON-3GRAM. TREC-3GRAM is the same as TREC dataset except that each record is transformed into a set of 3-grams.

In the preprocessing of datasets, letters are converted into the lowercases, and white spaces and punctuation are substituted with underscores before extracting 3-grams. Exact duplicates are removed, and all records are sorted in ascending order of size. The dataset statistics are listed in Table 6.

Similar to Xiao et al. (2009), our experiments covered Jaccard and Cosine similarities. The preprocessing and loading time are not included in the total running time . The experiments were run on a machine with a 1.9GHz Xeon(R) E5-2420 CPU and 16G RAM. The operating system is Ubuntu 12.04, and all algorithm were implemented in C++ and compiled using gcc-4.5.

5.2 Experiment results and analysis

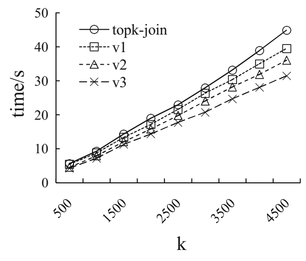
5.2.1 Experiment 1

In this experiment, we compared SETJoin(v1∼v3) with topk-join over DBLP and ENRON-3GRAM datasets. The similarities of the top-*k* pairs are very high in these two datasets, and the average record length of DBLP is short, whereas ENRON-3GRAM is mainly composed of long records. maxdepth is set to 6 in SETJoin(v2∼v3) according to the thumb rule. Each dataset is divided into 4 partitions for early termination,

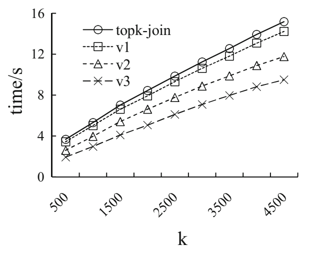
[6] http://www.informatik.uni-trier.de/ ley/db.

[7] http://trec.nist.gov/data/t9-filtering.html.

[8] http://www.cs.cmu.edu/ enron.



(a) ENRON-3GRAM, Jaccard



(b) ENRON-3GRAM, Cosine

Fig. 7 The runtime for `ENRON-3GRAM` dataset

and the sizes of large partitions are multiples of that of the smallest one. Figures 7 and 8 report the runtime on `DBLP` and `ENRON-3GRAM` datasets. As one can see, the runtimes of all algorithms grow with the increase in k and SETJoin outperforms topk-join consistently. For Jaccard similarity, SETJoin-v3 achieves 1.2x to 2.7x speedup on `DBLP` and 1.3x to 1.47x speedup on `ENRON-3GRAM`. For Cosise similarity, SETJoin-v3 is up to 1.8 and 1.9 times faster than topk-join on `DBLP` and `ENRON-3GRAM`, respectively. In addition to the obvious superiority of SETJoin, several interesting observations are worth mentioning.

1. In Fig. 8a, hash optimization dominates the performance improvement on `DBLP` under Jaccard similarity. As shown in Fig. 9a, the number of hash table lookups is reduced by two orders of magnitude in SETJoin-v1, which leads to a significant decrease in the total running time. In contrast, suffix filtering optimization didn't contribute much while the number of verification pairs has dropped by one order of magnitude as shown in Fig. 10a. This is mainly because (1) it takes very short time to compute the exact similarity of a single pair for short records and (2) the number of pruned candidates due to this optimization is still not large enough. Figure 8a shows that the early termination optimization didn't work well either. The reason is that the short prefix lengths make $ub_{\max}$ of the four partitions equal to 1 in most cases. In Fig. 8c, suffix filtering optimization and the early termination strategy begin to perform better as k grows, but their contributions are still incomparable with that of hash optimization.

2. Figure 8b, d shows that suffix filtering optimization works much better under Cosine similarity on `DLBP` dataset. The truth, however, is that the performance gain brought by hash optimization becomes less significant. Table 7 lists the similarities of the kth most similar pair on `DLBP` and `ENRON-3GRAM` datasets under Cosine and Jaccard similarity functions for different k values. Apparently, Cosine similarity thresholds are higher than those under Jaccard similarity for the same k. Consequently, as we can see in Fig. 9, the number of candidates under Cosine similarity is much smaller than its counterpart for Jaccard similarity,[9] which makes the hash optimization strategy less effective. At the same time, the performance improvement due to suffix filtering optimization remains almost the same because, as shown in Fig. 10, the numbers of verification pairs under Jaccard and Cosine similarities are both reduced by around the same order of magnitude in SETJoin-v2. The net result is that suffix filtering optimization looks comparable with the first optimization strategy.

Early termination performs better as well under Cosine similarity. Besides the reason discussed above, the other subtle cause is that the accessing similarity bound is $\frac{ub_{pr}^2 ub_{\max}^2}{ub_{pr}^2+ub_{\max}^2-ub_{pr}^2 ub_{\max}^2}$ for Cosine similarity (Xiao et al. 2009), which decreases faster than $\frac{ub_{pr} ub_{\max}}{ub_{pr}+ub_{\max}-ub_{pr} ub_{\max}}$ w.r.t. ub_{pr} and $ub_{\max}$. Therefore, the corresponding stop condition is much tighter than the one under Jaccard similarity.

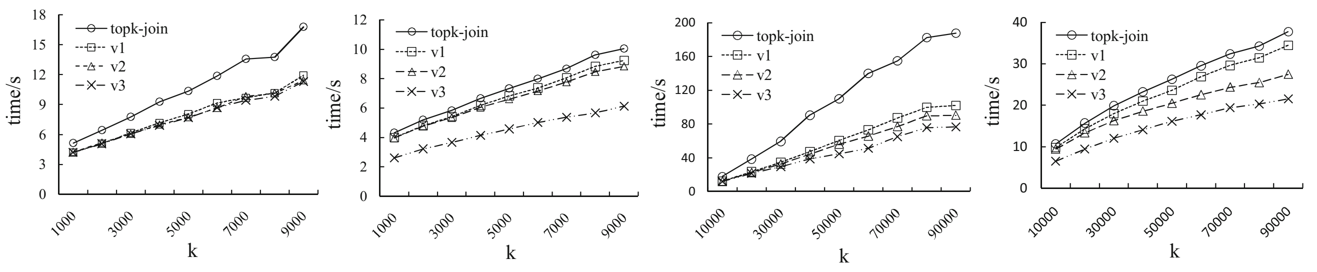
3. Suffix filtering optimization performs pretty well on `ENRON-3GAM` dataset as shown in Fig. 7. This is because the long average record length leads to a dramatic increase in the cost of exact similarity computation. Early termination also benefit from the long average record (prefix) length since there are more chances for $ub_{\max}$ to be less than 1.

5.2.2 Experiment 2

In this section, we compared SETJoin with topk-join over `TREC` and `TREC-3GRAM` datasets. The similarities of the top-k pairs are relatively low in these two datasets, and `TREC-3GRAM` has much longer average record length than `TREC`. maxdepth is set to 7 in SETJoin. Each dataset is divided into 4 partitions. Figure 11 reports the runtime on `TREC` and `TREC-3GRAM` datasets. As we can see, the runtimes of all algorithms grow with k and SETJoin beats topk-join in all cases. For `TREC` dataset, SETJoin-v3 achieves 2.54x to 3.46x speedup under Jaccard similarity and 2.92x to 3.06x speedup w.r.t. Cosine similarity. For `TREC-3GRAM` dataset, `SETJoin-v3` is up to 1.73 times faster than topk-join under Jaccard similarity and 1.63 times faster w.r.t. Cosine similarity. Also, we have the following observations.

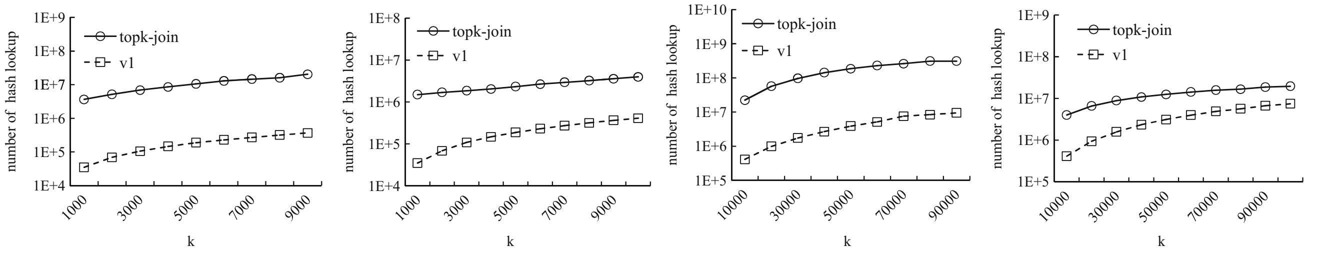
1. For `TREC` dataset, suffix filtering optimization dominates the performance gain under both Jaccard and Cosine similarities, whereas for `TREC-3GRAM` dataset, increasing maxdepth seems to be less effective. The reason is that, as shown in Fig. 12, the number of verification pairs decreases by three orders of magnitude for `TREC` dataset but only one order of magnitude for `TREC-3GRAM` dataset. Such a significant difference results from the different settings of maxdepth—in topk-join, maxdepth is set to 2 for `TREC` dataset while is already set to 4 for `TREC-3GRAM` dataset.

[9] Please note that the number of hash lookup operations is equal to the number of generated candidates in topk-join.



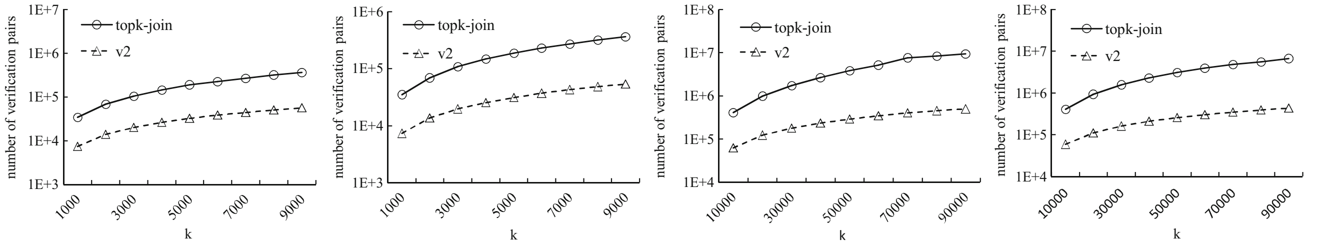
(a) DBLP,Jaccard, small k **(b)** DBLP, Cosine, small k **(c)** DBLP,Jaccard, large k **(d)** DBLP,Cosine, large k

Fig. 8 The runtime for DBLP dataset



(a) DBLP,Jaccard, small k **(b)** DBLP, Cosine, small k **(c)** DBLP,Jaccard, large k **(d)** DBLP,Cosine, large k

Fig. 9 The number of hash table lookup operations for DBLP dataset

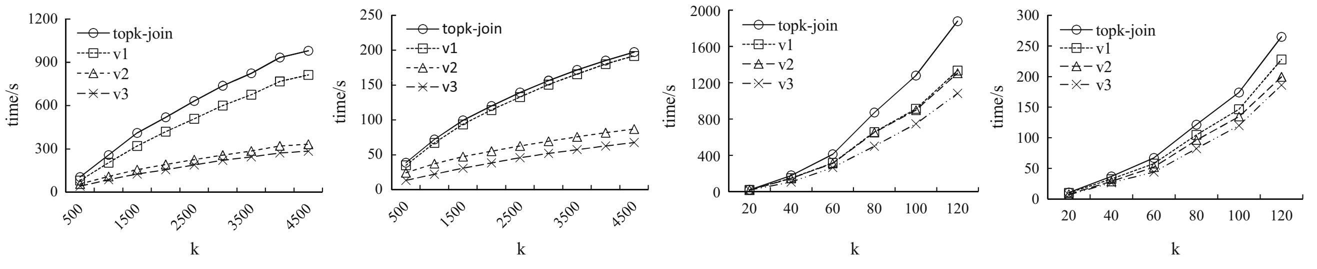


(a) DBLP,Jaccard, small k **(b)** DBLP, Cosine, small k **(c)** DBLP,Jaccard, large k **(d)** DBLP,Cosine, large k

Fig. 10 The number of verification pairs for DBLP dataset

Table 7 Similarities of the *k*th most similar pairs for different *k* values

Datasets/Similarity function	Top-1000	Top-5000	Top-10,000
DBLP/Jaccard	0.909	0.833	0.769
DBLP/Cosine	0.952	0.905	0.859
ENRON-3GRAM/Jaccard	0.991	0.974	–
ENRON-3GRAM/Cosine	0.995	0.987	–



(a) TREC, Jaccard **(b)** TREC, Cosine **(c)** TREC-3GRAM, Jaccard **(d)** TREC-3GRAM, Cosine

Fig. 11 The runtime for TREC and TREC-3Gram datasets

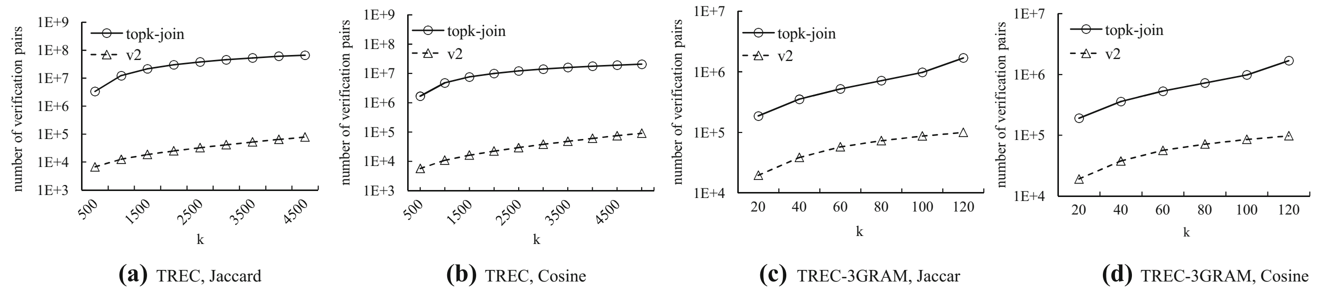


Fig. 12 The number of verification pairs for `TREC` and `TREC-3Gram` datasets

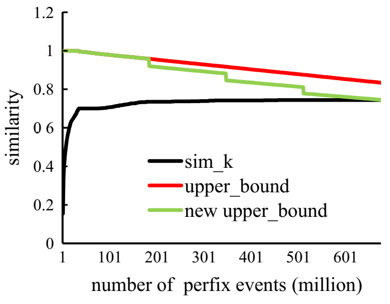


Fig. 13 An illustration of how does the early termination strategy works

Table 8 The number of pruned candidates before and after enabling hash optimization

Algorithm/Method	Top-500	Top-1000	Top-1500
topk-join/Suffix	25M	52M	72M
SETJoin-v1/Suffix	27M	56M	79M
topk-join/Positional	52M	121M	185M
SETJoin-v1/Positional	54M	127M	197M

2. As a typical example, Fig. 13 depicts how sim_k (sim_k), ub_{p_r} (upper_bound) and $\frac{ub_{p_r} ub_{\max}}{ub_{p_r}+ub_{\max}-ub_{p_r} ub_{\max}}$ (new upper_bound) vary over time in SETJoin-v3. The statistics is collected by running a top-500 query over `TREC` dataset under Jaccard similarity. As one can see, sim_k (ub_{p_r}) declines (rises) smoothly as the number of processed prefix events increases. On the contrary, $\frac{ub_{p_r} ub_{\max}}{ub_{p_r}+ub_{\max}-ub_{p_r} ub_{\max}}$ experienced frequent sharp declines because the $ub_{\max}$ factor is included in the new stop condition, which is fulfilled (the intersection of black and blue lines) before the original one. Obviously, this optimization is able to help SETJoin terminate much earlier than topk-join.

3. Table 8 lists the number of pruned candidate pairs for topk-join and SETJoin-v1. As expected, the pruning capabilities of suffix and positional filtering are both improved when hash optimization is enabled, which bears out the analysis made in Lemma 3 and Lemma 4.

6 Related work

Similarity joins have received much attention in recent years. Generally speaking, similarity joins can be categorized into two classes—the similarity-threshold-based one and top-k similarity join.

SimJoin has been widely studied in Arasu et al. (2006), Xiao et al. (2008), Lam et al. (2010), Bayardo et al. (2007), Jiang et al. (2014), Chaudhuri et al. (2006), Sarawagi and Kirpal (2004). In Chaudhuri et al. (2006), a primitive operator is proposed based on prefix filtering. Bayardo et al. utilizes the ordering of vectors and filtering techniques tailored for Cosine similarity function to find similar pairs (Bayardo et al. 2007). Arasu et al. employ the pigeon hole principle to divide the records into partitions and then eliminate false positive in a post filtering step (Arasu et al. 2006). A novel framework is designed to identify similar records using token transformations (Arasu et al. 2008). More recently, some researchers studied how to perform efficient similarity joins on a map-reduce platform (Metwally and Faloutsos 2012; Vernica et al. 2010; Baraglia et al. 2010; Sarma et al. 2014; Deng et al. 2014; Huang et al. 2014; Fries et al. 2014). Instead of providing the exact answers, some other studies focus on retrieving approximate similar pairs (Charikar 2002; Broder et al. 1998; Gravano et al. 2001; Behm et al. 2011; Jestes et al. 2010).

TopkJoin returns the k most similar pairs. The top-k join problem has been addressed in spatial database community (Corral et al. 2000; Zhu et al. 2005). For records in the form of sets in high dimensional space, Xiao et al. proposed the first top-k similarity join algorithm based on an event-driven framework. Kim and Shim (2012) proposed to use a map-reduce framework to support top-k similarity join.

Mann et al. (2016) compared 7 threshold-based set similarity join algorithms such as ALL, PPJ, PPJ+, MPJ, PEL, ADP, GRP. Quirino et al. (2018) uses GPU to speed up the similarity join algorithm. Wang et al. (2017) leveraging Set Relations to speed up similarity search algorithms. However, all of them belong to the threshold-based join algorithms. In the top-k similarity join topic, there are mostly application papers, such as SriUsha et al. (2018), Hu et al. (2016), Willi et al. (2017) and Xiong et al. (2015). These papers show the algorithm has broad applications in practice.

7 Conclusion

In this paper, we studied the problem of top-*k* similarity join over sets. In view of the performance bottlenecks in the state-of-the-art TopkJoin algorithm, we proposed a novel algorithm SETJoin. To be specific, 1. we proved that the simplified pruning conditions of positional and suffix filtering are applicable on all occasions no matter how many times a candidate pair has been assembled, based on which we eliminated the performance bottleneck by simply rearranging the orders of look up and filtering operations. 2. We developed a cost model to identify the determining factors that affect the performance. Based on this model, a thumb rule is introduced for choosing the (near) optimal recursion depth, which suggests that, counter intuitively, the recursion depth should not be too small. 3. We devised a much tighter stop condition. Theoretical analysis shows that the new stop condition does not affect the correctness of the algorithm and is able to cease join operations much earlier than the existing one. We have implemented our algorithm and compared with topk-join. For DBLP and ENRON-3GRAM data set, SETJoin-v3 is up to $1.8x$ and $1.9x$ times faster than topk-join, respectively. For TREC dataset, SETJoin-v3 achieves $2.54x$ to $3.46x$ speedup. For TREC-3GRAM dataset, SETJoin-v3 is up to $1.63x$ to $1.73x$ times faster than topk-join. Experimental results show that our algorithm outperforms substantially over the existing one on several real datasets.

Acknowledgements The work reported in this paper is partially supported by NSFC under Grant Numbers 61370205, NSF of Shanghai under Grant Number 13ZR1400800 and the Fundamental Research Funds for the Central Universities.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Human participants or animals rights This article does not contain any studies with human participants or animals performed by any of the authors.

References

Arasu A, Ganti V, Kaushik R (2006) Efficient exact set-similarity joins. In: VLDB, pp 918–929

Arasu A, Chaudhuri S, Kaushik R (2008) Transformation-based framework for record matching. In: ICDE, pp 40–49

Baraglia R, Morales GDF, Lucchese C (2010) Document similarity self-join with mapreduce. In: Webb GI, Zhang C, Gunopulos D, Wu X (eds) ICDM. IEEE Computer Society, Washington, pp 731–736

Bayardo RJ, Ma Y, Srikant R (2007) Scaling up all pairs similarity search. In: WWW, pp 131–140

Behm A, Li C, Carey MJ (2011) Answering approximate string queries on large data sets using external memory. In: ICDE, pp 888–899

Broder AZ, Charikar M, Frieze AM, Mitzenmacher M (1998) Min-wise independent permutations (extended abstract). In: STOC, pp 327–336

Charikar M (2002) Similarity estimation techniques from rounding algorithms. In: STOC, pp 380–388

Chaudhuri S, Ganti V, Kaushik R (2006) A primitive operator for similarity joins in data cleaning. In: ICDE, p 5

Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M (2000) Closest pair queries in spatial databases. In: SIGMOD, pp 189–200

Deng D, Li G, Hao S, Wang J, Feng J (2014) Massjoin: a mapreduce-based method for scalable string similarity joins. In: ICDE, pp 340–351

Fries S, Boden B, Stepien G, Seidl T (2014) Phidj: parallel similarity self-join for high-dimensional vector data with mapreduce. In: ICDE, pp 796–807

Gravano L, Ipeirotis PG, Jagadish HV, Koudas N, Muthukrishnan S, Srivastava D (2001) Approximate string joins in a database (almost) for free. In: VLDB, pp 491–500

Hernández MA, Stolfo SJ (1998) Real-world data is dirty: data cleansing and the merge/purge problem. Data Min Knowl Discov 2(1):9–37

Hu H, Li G, Bao Z, Feng J, Wu Y, Gong Z, Xu Y (2016) Top-k spatio-textual similarity join. IEEE Trans Knowl Data Eng 28(2):551–565

Huang J, Zhang R, Buyya R, Chen J (2014) MELODY-JOIN: efficient earth mover's distance similarity joins using mapreduce. In: ICDE, pp 808–819

Jestes J, Li F, Yan Z, Yi K (2010) Probabilistic string similarity joins. In: SIGMOD, pp 327–338

Jiang Y, Li G, Feng J, Li W (2014) String similarity joins: an experimental evaluation. PVLDB 7(8):625–636

Kim Y, Shim K (2012) Parallel top-k similarity join algorithms using mapreduce. In: ICDE, pp 510–521

Lam HT, Dung DV, Perego R, Silvestri F (2010) An incremental prefix filtering approach for the all pairs similarity search problem. APWeb 2010:188–194

Li G, He J, Deng D, Li J (2015) Efficient similarity join and search on multi-attribute data. In: SIGMOD, pp 1137–1151

Mann W, Augsten N, Bouros P (2016) An empirical evaluation of set similarity join techniques. Proc VLDB Endow 9(9):636–647

Metwally A, Faloutsos C (2012) V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. PVLDB 5(8):704–715

Quirino RD, Ribeiro-Junior S, Ribeiro LA,Martins WS (2018) Efficient filter-based algorithms for exact set similarity join on GPUs. In: Hammoudi S, Śmiałek M, Camp O, Filipe J (eds) Enterprise information systems. ICEIS 2017. Lecture notes in business information processing, vol 321. Springer, Cham, pp 74–95

Sarawagi S, Kirpal A (2004) Efficient set joins on similarity predicates. In: SIGMOD, pp 743–754

Sarma AD, He Y, Chaudhuri S (2014) Clusterjoin: a similarity joins framework using map-reduce. PVLDB 7(12):1059–1070

SriUsha I, Choudary KR, Sasikala T et al (2018) Data mining techniques used in the recommendation of e-commerce services. In: second international conference on electronics, communication and aerospace technology (ICECA). IEEE, pp 379–382

Vernica R, Carey MJ, Li C (2010) Efficient parallel set-similarity joins using mapreduce. In: SIGMOD, pp 495–506

Wang J, Li G, Feng J (2012) Can we beat the prefix filtering? An adaptive framework for similarity join and search. In: SIGMOD, pp 85–96

Wang X, Qin L, Lin X, Zhang Y, Chang L (2017) Leveraging set relations in exact set similarity join. Proc VLDB Endow 10(9):925–936

Willi M, Augsten N, Jensen CS (2017) Swoop: top-k similarity joins over set streams. arXiv: Databases

Xiao C, Wang W, Lin X, Yu JX (2008) Efficient similarity joins for near duplicate detection. In: WWW, pp 131–140

Xiao C, Wang W, Lin X, Shang H (2009) Top-k set similarity joins. In: ICDE, pp 916–927

Xiong Y, Zhu Y, Yu PS (2015) Top-k similarity join in heterogeneous information networks. IEEE Trans Knowl Data Eng 27(6):1710–1723

Zhu M, Papadias D, Zhang J, Lee DL (2005) Top-k spatial joins. IEEE Trans Knowl Data Eng 17(4):567–579