The VLDB Journal
DOI 10.1007/s00778-014-0367-9

REGULAR PAPER

A unified framework for approximate dictionary-based entity extraction

Dong Deng · Guoliang Li · Jianhua Feng · Yi Duan · Zhiguo Gong

Received: 12 November 2013 / Revised: 28 April 2014 / Accepted: 11 July 2014

Abstract Dictionary-based entity extraction identifies predefined entities (e.g., person names or locations) from documents. A recent trend for improving extraction recall is to support *approximate* entity extraction, which finds all substrings from documents that approximately match entities in a given dictionary. Existing methods to address this problem support either token-based similarity (e.g., Jaccard Similarity) or character-based dissimilarity (e.g., Edit Distance). It calls for a unified method to support various similarity/dissimilarity functions, since a unified method can reduce the programing efforts, the hardware requirements, and the manpower. In this paper, we propose a unified framework to support various similarity/dissimilarity functions, such as jaccard similarity, cosine similarity, dice similarity, edit similarity, and edit distance. Since many real-world applications have high-performance requirement for approximate entity extraction on data streams (e.g., Twitter), we focus on devising efficient algorithms to achieve high performance. We find that many substrings in documents have overlaps, and we can utilize the shared computation across the overlaps to avoid unnecessary redundant computation. To this end, we propose efficient filtering algorithms and develop effective pruning techniques. Experimental results show our method achieves high performance and outperforms state-of-the-art studies significantly.

Keywords Approximate entity extraction · Unified framework · Filtering algorithms · Pruning techniques

D. Deng (✉) · G. Li · J. Feng
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
e-mail: buaasoftdavid@gmail.com

G. Li
e-mail: liguoliang@tsinghua.edu.cn

J. Feng
e-mail: fengjh@tsinghua.edu.cn

Y. Duan
School of Software, Beihang University, Beijing, China
e-mail: windream1991@hotmail.com

Z. Gong
University of Macau, Macau, China
e-mail: fstzgg@umac.mo

1 Introduction

Dictionary-based entity extraction identifies all the substrings from documents that match the predefined entities in a given dictionary. For example, consider a tweet document "*I'm in the mood for something Coka-Cola flavored but I don't want an actual Coke*", and a dictionary with two entities "`Coca Cola`" and "`Coke`". Dictionary-based entity extraction finds the predefined entity "`Coke`" from the document. This problem has many real applications in the fields of information retrieval, molecular biology, bioinformatics, and natural language processing [5].

However, the document may contain typographical or orthographical errors and the same entity may have different representations [31]. For example, the substring "`Coka-Cola`" in the above document has orthographical errors. The traditional (exact) entity extraction cannot find this substring from the document, since the substring does not *exactly* match the predefined entity "`Coca Cola`". Approximate entity extraction is a recent trend to address this problem, which finds all substrings from the document that *approximately* match the predefined entities.

To improve extraction recall, in this paper, we study the problem of approximate dictionary-based entity extraction, which, given a dictionary of entities and a document, finds

Published online: 05 August 2014

all substrings from the document *similar* to some entities in the dictionary. Many similarity/dissimilarity functions have been proposed to quantify the similarity between two strings, such as jaccard similarity, cosine similarity, dice similarity, edit similarity, and edit distance. For instance, in the above example, suppose we use edit distance and the threshold is 3. Approximate dictionary-based entity extraction can find the substring "*Coka-Cola*" which is similar to the entity "`Coca Cola`".

Although there have been some studies on approximate entity extraction [4,31], they support either token-based similarity (e.g., Jaccard Similarity) or character-based dissimilarity (e.g., Edit Distance). It calls for a unified method to support various similarity/dissimilarity functions, since the unified method can reduce not only the programing efforts, but also the hardware requirements and the manpower needed to maintain the codes for different functions. Moreover, many real-world applications have high-performance requirement on approximate entity extraction. For example, the high performance is apparently very important for approximate entity extraction on data streams (e.g., Twitter and online advertisements). For instance, product analysts (e.g., *Coca Cola* staff) want to be informed of relevant tweets that comment on their products in Twitter. In this case, product analysts register some entities as their interests, and an approximate entity extraction system should efficiently deliver tweets to relevant product analysts. We observe that many substrings in the document have overlaps and we can share computation across substrings in the document. For example, considering the above document, many substrings such as "*Coka-Cola*", "*oka-Cola*", and "*Cola flavored*" have overlap "*Cola*". We can utilize this feature to avoid the redundant computation across overlaps of different substrings. For instance, we can share the computation on the common substring "*Cola*" for different substrings to improve the performance.

To address these problems, we propose a unified filtering framework for approximate dictionary-based entity extraction, called "Faerie", which can support various similarity/dissimilarity functions. To avoid redundant computation across overlaps of different substrings, we devise efficient filtering algorithms to achieve high performance and make the following contributions.

- We propose a unified framework to support many similarity/dissimilarity functions, such as jaccard similarity, cosine similarity, dice similarity, edit similarity, and edit distance.
- We devise two effective filtering algorithms, the multi-heap-based algorithm and the single-heap-based algorithm, which utilize the shared computation across the overlaps of multiple substrings of the document.
- We propose a hybrid-based algorithm to combine the single-heap-based algorithm and the multi-heap-based algorithm to improve the performance
- We develop efficient pruning techniques and devise efficient algorithms to improve the performance.
- We have implemented our proposed techniques, and the experimental results show that our method achieves high performance and outperforms state-of-the-art approaches significantly.

The remainder of this paper is organized as follows. We propose a unified framework to support various similarity functions in Sect. 2. Section 3 introduces a heap-based filtering algorithm to utilize shared computation. We develop pruning techniques in Sect. 4. We propose a hybrid method in Sect. 5. We conduct extensive experimental studies in Sect. 6. Related works are provided in Sect. 7. Finally, we conclude the paper in Sect. 8.

2 A unified framework

We first formulate the problem of approximate entity extraction (Sect. 2.1), and then introduce a unified method to support various similarity/dissimilarity functions (Sect. 2.2). Finally, we introduce a concept of "*valid substrings*" to prune unnecessary substrings (Sect. 2.3).

2.1 Problem formulation

Definition 1 (*Approximate Entity Extraction*) Given a dictionary of entities $E = \{e_1, e_2, \ldots, e_n\}$, a document D, a similarity function, and a threshold, it finds all "**similar**" pairs $\langle s, e_i \rangle$ with respect to the given function and threshold, where s is a substring of D and $e_i \in E$.

The similarity between two strings is usually quantified by similarity/dissimilarity functions, and in this paper, we focus on token-based similarity, character-based similarity, and character-based dissimilarity.

Token-based similarity The token-based similarity takes a string as a set of tokens. The representative functions include Jaccard Similarity (JAC), Cosine Similarity (COS), and Dice Similarity (DICE). Given two strings r and s, let $|r|$ denote the number of tokens in r. $\text{JAC}(r, s) = \frac{|r \cap s|}{|r \cup s|}$, $\text{COS}(r, s) = \frac{|r \cap s|}{\sqrt{|r| \cdot |s|}}$, and $\text{DICE}(r, s) = \frac{2|r \cap s|}{|r| + |s|}$. For example, $\text{JAC}(\text{“}\texttt{vldb journal 2013}\text{”}, \text{“}\texttt{vldb journal}\text{”}) = \frac{2}{3}$, $\text{COS}(\text{“}\texttt{vldb journal 2013}\text{”}, \text{“}\texttt{vldb journal}\text{”}) = \frac{2}{\sqrt{6}}$, and $\text{DICE}(\text{“}\texttt{vldb journal 2013}\text{”}, \text{“}\texttt{vldb journal}\text{”}) = \frac{4}{5}$.

Character-based dissimilarity The character-based dissimilarity takes a string as a sequence of characters. The rep-

Table 1 A dictionary of entities and a document

ID	Entity e	len(e)	$\lvert e\rvert$ (# of q-grams with $q=2$)
(a) Dictionary E			
1	`kaushik ch`	10	9
2	`chakrabarti`	11	10
3	`chaudhuri`	9	8
4	`venkatesh`	9	8
5	`surajit ch`	10	9
(b) Document D			
an efficient filter for approximate membership checking. venkaee shga kamunshik kabarati, dong xin, surauijt chadhurisigmod			

resentative function is Edit Distance. The edit distance of strings r and s, denoted by $\text{ED}(r, s)$, is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform r to s. For example, ED("`surajit`", "`surauijt`")=2.

Character-based similarity: The representative function is Edit Similarity. The edit similarity between two strings r and s is defined as $\text{EDS}(r, s) = 1 - \frac{\text{ED}(r,s)}{\max(\text{len}(r),\text{len}(s))}$, where $\text{len}(s)$ is the length of s. For example, EDS("`surajit`", "`surauijt`") $= 1 - \frac{2}{8} = \frac{3}{4}$.

In this paper, two strings are said to be *similar*, if their (jaccard, cosine, dice, edit) similarity is no smaller than a given similarity threshold δ, or their edit distance is no larger than a given edit-distance threshold τ. For instance, consider the document D and dictionary E in Table 1. Suppose the edit-distance threshold $\tau = 2$. ⟨"*venkaee sh*", "`venkatesh`"⟩, ⟨"*surauijt ch*", "`surajit ch`"⟩, and ⟨"*chadhuri*", "`chaudhuri`"⟩ are three example results. Especially, although the substring "*chadhurisigmod*" misses a space between "*chadhuri*" and "*sigmod*" (a typographical error), our method still finds "*chadhuri*" (similar to entity "`chaudhuri`").

It has been shown that approximate entity extraction can improve recall [31]. For example, the recall can increase from 65.4 to 71.4 % when performing protein name recognition. In this paper, we emphasize on improving the performance. We focus on extracting textual entities. We assume the thresholds (δ and τ) are pre-defined and how to select such thresholds is orthogonal to this work.

2.2 A unified framework

In this section, we propose a unified framework to support various similarity/dissimilarity functions.

We model both the entity and document as a set of tokens. Especially for edit distance and edit similarity, we take q-grams of an entity as tokens. A q-gram of a string s is a substring of s with length q. The q-gram set of s, denoted by $G(s)$, is the set of all of s's q-grams. For example, the 2-gram set of "`surajit_ch`" is {`su`, `ur`, `ra`, `aj`, `ji`, `it`, `t_`, `_c`, `ch`}. If the context is clear, we use token to denote token/gram; for edit distance and edit similarity, we use e to denote $G(e)$, $e \cap s$ to denote $G(r) \cap G(s)$, and $|e|$ to denote $|G(e)|$ (i.e., $|e| = \text{len}(e) - q + 1$).

Given an entity e and a substring s, we transform different similarities/dissimilarities to the *overlap similarity* ($|e \cap s|$) and use the *overlap similarity* as a unified filtering condition: if e and s are similar, then $|e \cap s|$ cannot be smaller than a threshold $T > 0$, where T can be computed as follows.

- Jaccard Similarity: $T = \lceil (|e| + |s|) * \frac{\delta}{1+\delta} \rceil$.
- Cosine Similarity: $T = \lceil \sqrt{|e| \cdot |s|} * \delta \rceil$.
- Dice Similarity: $T = \lceil (|e| + |s|) * \frac{\delta}{2} \rceil$.
- Edit Distance: $T = \max(|e|, |s|) - \tau * q$.
- Edit Similarity:

$$T = \lceil \max(|e|, |s|) - \big(\max(|e|, |s|) + q - 1\big) * (1 - \delta) * q \rceil.$$

The correctness of these thresholds is stated in Lemma 1.

Lemma 1 *Given an entity e and a substring s, we have,*[1]

- Jaccard Similarity: *If* $\text{JAC}(e, s) \geq \delta$, $|e \cap s| \geq \lceil (|e| + |s|) * \frac{\delta}{1+\delta} \rceil$.
- Cosine Similarity: *If* $\text{COS}(e, s) \geq \delta$, $|e \cap s| \geq \lceil \sqrt{|e| \cdot |s|} * \delta \rceil$.
- Dice Similarity: *If* $\text{DICE}(e, s) \geq \delta$, $|e \cap s| \geq \lceil (|e| + |s|) * \frac{\delta}{2} \rceil$.
- Edit Distance: *If* $\text{ED}(e, s) \leq \tau$, $|e \cap s| \geq \max(|e|, |s|) - \tau * q$.
- Edit Similarity: *If* $\text{EDS}(e, s) \geq \delta$,

$$|e \cap s| \geq \lceil \max(|e|, |s|) - \big(\max(|e|, |s|) + q - 1\big) * (1 - \delta) * q \rceil.$$

Proof (1) Jaccard Similarity: As $\text{JAC}(e, s) = \frac{|e \cap s|}{|e \cup s|} = \frac{|e \cap s|}{|e| + |s| - |e \cap s|} \geq \delta$, we have $|e \cap s| \geq (|e| + |s|) * \frac{\delta}{1+\delta}$. As $|e \cap s|$ is an integer, we have $|e \cap s| \geq \lceil (|e| + |s|) * \frac{\delta}{1+\delta} \rceil$.
(2) Cosine Similarity: $\text{COS}(e, s) = \frac{|e \cap s|}{\sqrt{|e| \cdot |s|}} \geq \delta$. Thus $|e \cap s| \geq \lceil \sqrt{|e| \cdot |s|} * \delta \rceil$.
(3) Dice Similarity: $\text{DICE}(e, s) = \frac{2|e \cap s|}{|e| + |s|} \geq \delta$. Thus $|e \cap s| \geq \lceil \frac{|e| + |s|}{2} * \delta \rceil$.
(4) Edit Distance: It is obvious as two strings r and s are similar only if they share enough common q-grams [13].

[1] In this paper, we omit the proof due to space constraints.

(5) Edit Similarity: As $\text{EDS}(e,s) = 1 - \frac{\text{ED}(e,s)}{\max(\mathsf{len}(e),\mathsf{len}(s))} \geq \delta$, $\text{ED}(e,s) \leq \max(\mathsf{len}(e),\mathsf{len}(s)) * (1-\delta)$. Based on Edit Distance, we have $|e \cap s| \geq \max(|e|,|s|) - \max(\mathsf{len}(e),\mathsf{len}(s)) * (1-\delta) * q$, thus $|e \cap s| \geq \lceil \max(|e|,|s|) - (\max(|e|,|s|) + q - 1) * (1-\delta) * q \rceil$. □

Accordingly, we can transform various similarities or dissimilarities to the overlap similarity and develop a unified filtering condition: if $|e \cap s| < T$, we prune the pair $\langle e, s \rangle$. Note that given any similarity function and a threshold, if we can deduce a lower bound for the overlap similarity of two strings, our method can apply to this function. Specially, the five similarity/distance functions we studied are commonly used in information extraction and record linkage [4,31].

2.3 Valid substrings

We have an observation that some substrings in D will not have any similar entities. For instance, consider the dictionary and document in Table 1. Suppose, we use edit distance and $\tau = 1$. Consider substring "*surauijt chadhurisigmod*" with length 23. As the lengths of entities in the dictionary are between 9 and 11, the substring cannot be similar to any entity. Next we discuss how to prune such substrings.

Given an entity e and a substring s, if s is similar to e, the number of tokens in s ($|s|$) should be in a range $[\perp_e, \top_e]$, that is $\perp_e \leq |s| \leq \top_e$, where $\perp_e$ and $\top_e$ are, respectively, the lower and upper bound of $|s|$, computed as below:

- Jaccard Similarity: $\perp_e = \lceil |e| * \delta \rceil$ and $\top_e = \lfloor \frac{|e|}{\delta} \rfloor$.
- Cosine Similarity: $\perp_e = \lceil |e| * \delta^2 \rceil$ and $\top_e = \lfloor \frac{|e|}{\delta^2} \rfloor$.
- Dice Similarity: $\perp_e = \lceil |e| * \frac{\delta}{2-\delta} \rceil$ and $\top_e = \lfloor |e| * \frac{2-\delta}{\delta} \rfloor$.
- Edit Distance: $\perp_e = |e| - \tau$ and $\top_e = |e| + \tau$.
- Edit Similarity: $\perp_e = \lceil (|e| + q - 1) * \delta - (q-1) \rceil$ and $\top_e = \lfloor \frac{|e|+q-1}{\delta} - (q-1) \rfloor$.

where δ is the similarity threshold and τ is the edit-distance threshold. The correctness of the bounds is stated in Lemma 2.

Lemma 2 *Given an entity e, for any substring s, we have*

- Jaccard Similarity: *if* $\text{JAC}(e,s) \geq \delta$, $\lceil |e| * \delta \rceil \leq |s| \leq \lfloor \frac{|e|}{\delta} \rfloor$.
- Cosine Similarity: *if* $\text{COS}(e,s) \geq \delta$, $\lceil |e| * \delta^2 \rceil \leq |s| \leq \lfloor \frac{|e|}{\delta^2} \rfloor$.
- Dice Similarity: *if* $\text{DICE}(e,s) \geq \delta$, $\lceil |e| * \frac{\delta}{2-\delta} \rceil \leq |s| \leq \lfloor |e| * \frac{2-\delta}{\delta} \rfloor$.
- Edit Distance: *if* $\text{ED}(e,s) \leq \tau$, $|e| - \tau \leq |s| \leq |e| + \tau$.
- Edit Similarity: *if* $\text{EDS}(e,s) \geq \delta$,

$$\lceil (|e| + q - 1) * \delta - (q-1) \rceil \leq |s| \leq \lfloor \frac{|e| + q - 1}{\delta} - (q-1) \rfloor.$$

Proof (1) Jaccard Similarity: As $|e| \geq |e \cap s|$, $\frac{|e|}{|s|} \geq \frac{|e|}{|e|+|s|-|e \cap s|} \geq \frac{|e \cap s|}{|e|+|s|-|e \cap s|} \geq \delta$. Thus, we have $|s| \leq \lfloor \frac{|e|}{\delta} \rfloor$. As $|s| \geq |e \cap s|$, $\frac{|s|}{|e|} \geq \frac{|s|}{|e|+|s|-|e \cap s|} \geq \frac{|e \cap s|}{|e|+|s|-|e \cap s|} \geq \delta$. Thus, we have $|s| \geq \lceil |e| * \delta \rceil$. Hence, $\lceil |e| * \delta \rceil \leq |s| \leq \lfloor \frac{|e|}{\delta} \rfloor$.

(2) Cosine Similarity: As $|e| \geq |e \cap s|$, $\frac{|e|}{\sqrt{|e| \cdot |s|}} \geq \frac{|e \cap s|}{\sqrt{|e| \cdot |s|}} \geq \delta$. Thus, we have $|s| \leq \lfloor \frac{|e|}{\delta^2} \rfloor$. As $|s| \geq |e \cap s|$, $\frac{|s|}{\sqrt{|e| \cdot |s|}} \geq \frac{|e \cap s|}{\sqrt{|e| \cdot |s|}} \geq \delta$. Thus $|s| \geq \lceil |e| * \delta^2 \rceil$. Hence $\lceil |e| * \delta^2 \rceil \leq |s| \leq \lfloor \frac{|e|}{\delta^2} \rfloor$.

(3) Dice Similarity: As $|e| \geq |e \cap s|$, $\frac{2|e|}{|e|+|s|} \geq \frac{2|e \cap s|}{|e|+|s|} \geq \delta$. Thus $|s| \leq \lfloor |e| * \frac{2-\delta}{\delta} \rfloor$. As $|s| \geq |e \cap s|$, $\frac{2|s|}{|e|+|s|} \geq \frac{2|e \cap s|}{|e|+|s|} \geq \delta$. Thus $|s| \geq \lceil |e| * \frac{\delta}{2-\delta} \rceil$. Hence $\lceil |e| * \frac{\delta}{2-\delta} \rceil \leq |s| \leq \lfloor |e| * \frac{2-\delta}{\delta} \rfloor$.

(4) Edit Distance: As $\big||e| - |s|\big| \leq \tau$, $|e| - \tau \leq |s| \leq |e| + \tau$.

(5) Edit Similarity: As $\text{EDS}(e,s) = 1 - \frac{\text{ED}(e,s)}{\max(\mathsf{len}(e),\mathsf{len}(s))} \geq \delta$, $\text{ED}(e,s) \leq \max(\mathsf{len}(e),\mathsf{len}(s)) * (1-\delta)$. If $|e| \leq |s|$, $|\mathsf{len}(s) - \mathsf{len}(e)| \leq \text{ED}(e,s) \leq \max(\mathsf{len}(e),\mathsf{len}(s)) * (1-\delta) = \mathsf{len}(s) * (1-\delta)$. Thus $\mathsf{len}(s) \leq \frac{\mathsf{len}(e)}{\delta}$ and $|s| \leq \lfloor \frac{|e|+q-1}{\delta} - (q-1) \rfloor$. If $|e| > |s|$, $\mathsf{len}(e) - \mathsf{len}(s) \leq \text{ED}(e,s) \leq \max(\mathsf{len}(e),\mathsf{len}(s)) * (1-\delta) = \mathsf{len}(e) * (1-\delta)$. $\mathsf{len}(s) \geq \mathsf{len}(e) * \delta$ and $|s| \geq \lceil (|e| + q - 1) * \delta - (q-1) \rceil$. Thus $\lceil (|e| + q - 1) * \delta - (q-1) \rceil \leq |s| \leq \lfloor \frac{|e|+q-1}{\delta} - (q-1) \rfloor$. □

Based on Lemma 2, given an entity e, only those substrings with token numbers between $\perp_e$ and $\top_e$ could be similar to entity e, and others can be pruned. Especially, let $\perp_E = \min\{\perp_e | e \in E\}$ and $\top_E = \max\{\top_e | e \in E\}$. Obviously, the substrings in D with token numbers between $\perp_E$ and $\top_E$ may have a similar entity in the dictionary E, and others can be pruned. Based on this observation, we introduce the concept of "valid substring."

Definition 2 (*Valid Substring*) Given a dictionary E and a document D, a substring s in D is a valid substring for an entity $e \in E$ if $\perp_e \leq |s| \leq \top_e$. A substring s in D is a valid substring for dictionary E if $\perp_E \leq |s| \leq \top_E$.

For instance, consider the dictionary and document in Table 1. Suppose we use edit similarity, and $\delta = 0.8$ and $q = 2$. Consider entity e_5 = "`surajit ch`". We have

$$\perp_{e_5} = \lceil (|e_5| + q - 1) * \delta - (q-1) \rceil = 7,$$
$$\top_{e_5} = \lfloor \frac{|e_5| + q - 1}{\delta} - (q-1) \rfloor = 11.$$

Only the valid substrings with token numbers between 7 and 11 could be similar to entity e_5. As $\perp_E = 7$ and $\top_E = 12$, only the valid substrings with token numbers between 7 and 12 could have similar entities in the dictionary, and all other substrings (e.g., "*surauijt chadhurisigmod*") can be pruned.

A naive method to solve the approximate entity extraction problem first enumerates every valid substring of the document, and then for each valid substring and each entity, it calculates their similarity/dissimilarity and outputs the similar pairs. However, this naive method requires to enumerate large numbers of pairs and thus leads to low performance. To this end, we employ a filter-and-verify framework. In the filter step, we generate *the candidate pairs* of a valid substring in document D and an entity in dictionary E, whose overlap similarity is no smaller than a threshold T; and in the verify step, we verify the candidate pairs to get the final results, by computing the real similarity/dissimilarity. As the verification step is very easy to address by using a merge-join-based algorithm, in this paper we focus on the filter step.

3 Heap-based filtering algorithms

The filter-and-verify framework relies on effective index structures and efficient filtering algorithms to prune dissimilar pairs based on the index structures. To this end, in this section, we first introduce an index structure (Sect. 3.1), and then propose two heap-based filtering algorithms, the multi-heap-based algorithm (Sect. 3.2) and the single-heap-based algorithm (Sect. 3.3).

3.1 An inverted index structure

A valid substring is similar to an entity only if they share enough common tokens. To efficiently count the number of their common tokens, we build an inverted index for all entities, where entries are tokens (for jaccard similarity, cosine similarity, and dice similarity) or q-grams (for edit similarity and edit distance), and each entry has an inverted list that keeps the ids of entities that contain the corresponding token/gram, sorted in ascending order. For example, Fig. 1 gives the inverted list for entities in Table 1 using q-grams with $q = 2$.

For each valid substring s in D, we first get its tokens and the corresponding inverted lists. Then, for each entity in these inverted lists, we count its *occurrence number* in the inverted lists, i.e., the number of inverted lists that contain the entity. Obviously, the occurrence number of entity e is exactly $|e \cap s|$.[2] For each entity, e with occurrence number no smaller than T ($|e \cap s| \geq T$), $\langle s, e \rangle$ is a candidate pair.

ka→1→4 au→1→3 us→1 sh→1→4 hi→1 ik→1
k_→1 _c→1→5 ch→1→2→3→5 ha→2→3 ak→2 kr→2
ra→2→5 ab→2 ba→2 ar→2 rt→2 ti→2
ud→3 dh→3 hu→3 ur→3→5 ri→3 ve→4
en→4 nk→4 at→4 te→4 es→4 su→5
aj→5 ji→5 it→5 t_→5

Fig. 1 Inverted indexes for entities in Table 1

For example, consider a valid substring "*surauijt ch*". We first generate its token set {su, ur, ra, au, ui, ij, jt, t_, _c, ch} and get the inverted lists (the italic ones in Fig. 1). Suppose we use edit distance and $\tau = 2$. For entity e_5, $T = \max(|e_5|, |s|) - \tau * q = 6$. As e_5's occurrence number is 6, ⟨"*surauijt ch*", e_5="surajit ch"⟩ is a candidate pair.

For simplicity, given an entity e and a valid substring s, we use e's occurrence number in s (or s's inverted lists) to denote e's occurrence number in the inverted lists of tokens in s. To efficiently count the occurrence numbers, we propose heap-based filtering algorithms in the following sections.

3.2 Multi-heap-based method

In the filter step of the filter-and-verify framework, we focus on generating the candidate pairs of a valid substring in document D and an entity in dictionary E, whose overlap similarity is not smaller than a threshold T. Using the inverted indexes, we want to count the occurrence number of the entity on the inverted lists of the valid substring and output the pair of the entity and the valid substring as a candidate if the occurrence number is not smaller than T. To facilitate computing the occurrence number and avoid enumerating every pairs, we propose a multi-heap-based method.

We first enumerate the valid substrings in D (with token number between $\perp_E$ and $\top_E$). Then for each valid substring, we generate its tokens and construct a min-heap on top of the non-empty inverted lists of its tokens. Initially, we use the first entity of every inverted list to construct the min-heap. For the top entity on the heap, we count its occurrence number on

[2] In this paper, we take e and s as multisets, since there may exist duplicate tokens in entities and substrings of the document. Even if they are taken as sets, we can also use our method for extraction.

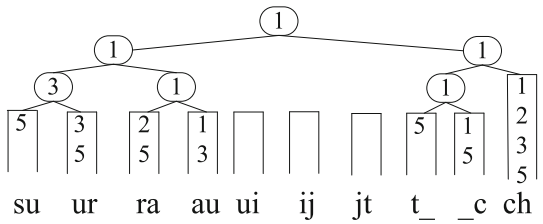


Fig. 2 A heap structure for "surauijt ch"

Table 2 Complexity of multi-heap based methods

(a) Space complexity	
Maximum Heap	$\mathcal{O}(\top_E)$
(b) Time complexity	
Heap Construction	$\mathcal{O}\left(\sum_{l=\perp_E}^{\top_E}(\lvert D\rvert - l + 1) * l\right)$
Heap Adjustment	$\mathcal{O}\left(\sum_{l=\perp_E}^{\top_E} log(l) * l * N\right)$

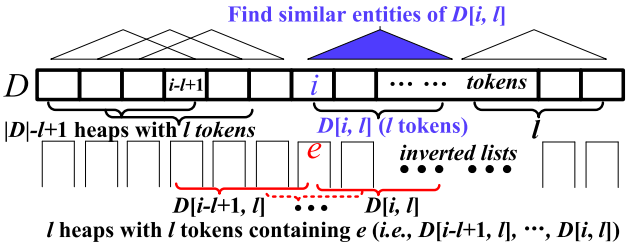


Fig. 3 A multi-heap structure

the heap (i.e., the number of inverted lists that contain the entity). If the number is not smaller than T, the pair of this valid substring and the entity is a candidate pair. Next, we pop the top entity, add the next entity of the inverted list from which the top entity is selected into the heap, adjust the heap, and count the occurrence number of the new top entity. Iteratively, we find all candidate pairs.

For example, consider a valid substring "*surauijt ch*". We first generate its token set and construct a min-heap on top of the first entities of every inverted list (Fig. 2). Next, we iteratively adjust the heap and get the entities {1, 1, 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5} in ascending order. We count the occurrence numbers of each entry. For example, the occurrence numbers of e_1, e_2, e_3, and e_5 are, respectively, 3, 2, 3, and 6. Suppose we use edit distance and $\tau = 2$. For entity e_5, $T = \max(|e_5|, |s|) - \tau * q = 6$. The pair of the substring and entity e_5 is a candidate pair. Finally, we verify the candidate pair and get the final result.[3]

Complexity For a valid substring with l tokens, its corresponding heap contains at most l non-empty inverted lists. Thus, the space complexity of the heap is $\mathcal{O}(l)$. As we can construct heaps one by one, the space complexity is the space of the maximum heap, i.e., $\mathcal{O}(\top_E)$ (Table 2a).

The time complexity for constructing a heap of a valid substring with l tokens is $\mathcal{O}(l)$. As there are $|D| - l + 1$ valid substrings with l tokens, the heap construction complexity for such valid substrings is $\mathcal{O}\big((|D| - l + 1) * l\big)$, and the total heap construction complexity is $\mathcal{O}\big(\sum_{l=\perp_E}^{\top_E}(|D| - l + 1) * l\big)$ (Table 2b). In addition, for each entity, we need to adjust the heap containing the entity. Consider such heap with l inverted lists. The time complexity of adjusting the heap once is $\mathcal{O}(log(l))$. There are l such heaps that contain the entity (Fig. 3). Thus, for each entity, the time complexity of adjusting the heaps is $\mathcal{O}\big(\sum_{l=\perp_E}^{\top_E} log(l) * l\big)$. Suppose N is the total numbers of entities in inverted lists of tokens in D. The total time complexity of adjusting the heaps is $\mathcal{O}\big(\sum_{l=\perp_E}^{\top_E} log(l) * l * N\big)$ (Table 2b).

It is worth noting that although we can scan all the inverted lists to compute the occurrences, this method requires high space complexity to keep the occurrences of each entity and the space complexity is $\mathcal{O}(n)$ where n is the number of entities in the dictionary. The space complexity of the multi-heap-based method is $\mathcal{O}(1)$. Moreover, we can utilize more efficient filtering techniques to improve the heap-based methods [21] which can void scanning all the entities in the inverted lists. In this paper, we propose effective algorithms to simultaneously find similar entities for *multiple* substrings (with large number of overlaps), which are orthogonal to the heap-merge algorithms [21].

3.3 Single-heap-based method

The multi-heap-based method needs to access the inverted lists multiple times and does large numbers of heap-adjustment operations. To address this issue, we propose a single-heap-based method which accesses every inverted list only once in this section.

We first tokenize the document D and get a list of tokens. For each token, we retrieve the corresponding inverted list. We use $token[i]$ to denote the i-th token, and $\mathcal{IL}[i]$ to denote the inverted list of the i-th token. We construct a single min-heap on top of non-empty inverted lists of all tokens in D, denoted by H, and use the heap to find candidate pairs.

For ease of presentation, we use a two-dimensional array $V[1 \cdots |D|][\perp_E \cdots \top_E]$ to count an entity's occurrence numbers in every valid substring's inverted lists. Formally, let $D[i, l]$ denote a valid substring of D with l tokens starting with the i-th token. Given an entity e, we use $V[i][l]$ to count e's *occurrence number* in $D[i, l]$'s inverted lists, i.e., $V[i][l] = |e \cap D[i, l]|$. We compute $V[i][l]$ as follows. First for each entity, $V[i][l]$ is initialized as 0 for $1 \le i \le |D|$ and $\perp_E \le l \le \top_E$.

For the top entity e on the heap selected from the i-th inverted list, we increase the values of *relevant entries* in the array by 1 as follows. Without loss of generality, firstly consider the heap with l tokens. Obviously, only $D[i - l + 1, l], \ldots, D[i, l]$ contain the i-th inverted list (Fig. 4), thus $V[i - l + 1][l], \ldots, V[i][l]$ are relevant entries. Similarly, for $\perp_E \le l \le \top_E$, $V[i - l + 1][l], \ldots, V[i][l]$ are relevant entries. We increase the value of each relevant entry by 1. If $V[i][l] \ge T$, $\langle D[i, l], e\rangle$ is a candidate pair. Then, we pop the

[3] For ease of presentation, we use a loser tree to represent a heap structure in our examples.

Table 3 Complexity of single-heap based methods

(a) Space complexity	
Single heap	$\mathcal{O}(\|D\|)$
Counting occurrence numbers	$\mathcal{O}(\|D\| - \perp_E + 1\|)$
(b) Time complexity	
Heap construction	$\mathcal{O}(\|D\|)$
Heap adjustment	$\mathcal{O}\big(log(\|D\|) * N\big)$
Counting occurrence numbers	$\mathcal{O}\big(N * \max\{\sum_{l=\perp_e}^{\top_e} l \| e \in E\}\big)$

top entity, add the next entity in $\mathcal{IL}[i]$ into the heap, adjust the heap and get the next entity, and count the occurrence number of the new entity. We repeat the above steps, and iteratively, we can find all candidate pairs.

Actually, for entity e, only the valid substrings with token numbers between $\perp_e$ and $\top_e$ could be similar to entity e.[4] Thus, we only need to maintain the array V.

Next, we use a running example to walk through the single-heap-based method. For example, in our running example, consider a document "`venkaee shga kamunshi`". We construct a single heap on top of the document as shown in Fig. 5. Suppose we use edit distance and $\tau = 2$. $\perp_E = 6$ and $\top_E = 12$. For the entity e_4 selected from the first token, we only need to increase its occurrence numbers in valid substrings $D[1, l]$ for $\perp_E \leq l \leq \top_E$, i.e., $D[1, 6], \ldots, D[1, 12]$. We increase the values of $V[1][6], \ldots, V[1][12]$ by 1. For the next entity e_4 selected from the second token, we increase its occurrence numbers in valid substrings $D[1, l]$, $D[2, l]$ for $\perp_E \leq l \leq \top_E$. Similarly, we can count all occurrence numbers. For instance, the occurrence number of entity e_4 ("`venkatesh`") in $D[1, 9]$ is 5. As the occurrence number is no smaller than $T = \max(|e_4|, |D[1, 9]|) - \tau * q = 9 - 2 * 2 = 5$, the pair of $D[1, 9]$ ("`venkaee sh`") and entity e_4 ("`venkatesh`") is a candidate pair. Actually, as $\perp_{e_4} = 6$ and $\top_{e_4} = 10$, we only need to consider the entries in $V[1 \cdots 20][6 \cdots 10]$.

Complexity The space complexity of the single heap is $\mathcal{O}(|D|)$ (Table 3a). To count the occurrence numbers of an entity, we do not need to maintain the array and propose an alternative method. We first pop all entities with the same id from the heap (with $|D|$ space to store them). Suppose the entity is e. Then, we increase e's occurrence numbers in $V[1 \cdots |D| - l + 1][l]$ by varying l from $\perp_e$ to $\top_e$. In this way, we only need to maintain a one-dimensional array. Thus, the space complexity for counting the occurrence number is $\mathcal{O}(\max\{|D| - \perp_e + 1 | e \in E\}) = \mathcal{O}(|D| - \perp_E + 1)$ (Table 3a).

[4] Note that, we can get entity e's token number $|e|$ using a hash map, which keeps the pair of an entity and its token number, thus we can get the token number of an entity in $\mathcal{O}(1)$ time complexity.

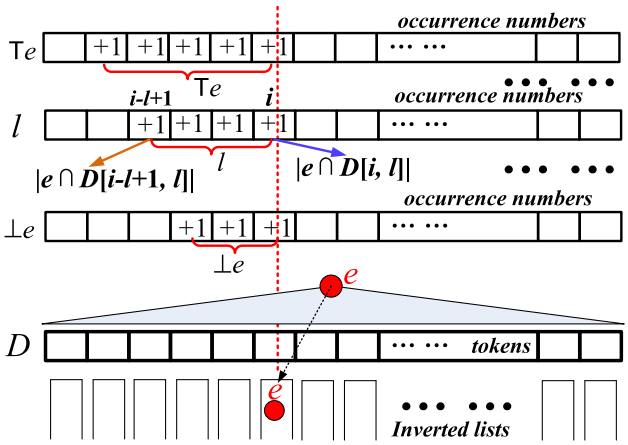


Fig. 4 A single-heap structure

The time complexity of heap construction is $\mathcal{O}(|D|)$ (Table 3b). To compute the occurrence numbers of each entity, we need to adjust the heap, and the total time complexity of adjusting the heap is $\mathcal{O}(log(|D|) * N)$, where N is the total number of elements in every inverted list. In addition, for each entity, we need to increase its occurrence numbers. For entity e, there are $\sum_{l=\perp_e}^{\top_e} l$ entries needed to be increased by 1 (Fig. 4), and the maximum number of such entries (for any entity) is $\max\{\sum_{l=\perp_e}^{\top_e} l | e \in E\}$. Thus, the total time complexity is $\mathcal{O}\big(N * \max\{\sum_{l=\perp_e}^{\top_e} l | e \in E\}\big)$ (Table 3b).

Note that the single-heap-based method has used the shared computation across the overlaps (tokens) of different valid substring, since it only needs to scan each inverted list once. It has much lower time complexity than multi-heap-based method and achieves much higher performance (Sect. 6).

4 Improving the single-heap-based method

The single-heap-based method still requires to compute the occurrence number of every entity. If there are large numbers of entities, this method is still rather expensive. To address this issue, in this section, we improve the single-heap-based method and propose effective techniques to skip many irrelevant entities.

4.1 Pruning techniques

In this section, we propose several pruning techniques by estimating the lower bounds of $|e \cap s|$.

Lazy-count pruning We find that if the occurrence number of an entity in the heap is small enough, we do not need to count its occurrence numbers for every valid substring immediately. Instead, we count the occurrence numbers in a lazy manner.

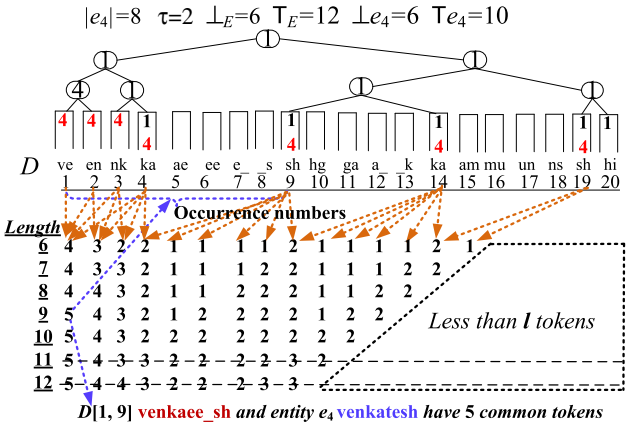


Fig. 5 An example for the single-heap-based method on a document "venkaee shga kamunshi"

Formally, given an entity e, we use a sorted position list $P_e = [p_1, \cdots, p_m]$ to keep its occurrences in the heap (in ascending order). Each element in P_e is the position of the corresponding token whose inverted list contains entity e. We can easily get the position list using the heap structure. Then, we count e's occurrence number in the heap, i.e., the number of elements in P_e ($|P_e|$). If the number is smaller than a threshold, denoted as T_l, we prune the entity; otherwise we count its occurrence number in its valid substrings with token numbers between $\perp_e$ and $\top_e$ (Sect. 3.3). For example, in Fig. 5, $P_{e_1} = [4, 9, 14, 19, 20]$ and $|P_{e_1}| = 5$.

Next, we discuss how to compute the threshold T_l. Recall the threshold T for the overlap similarity (Sect. 2.2). T depends on both $|e|$ and $|s|$. To derive a lower bound of T which only depends on $|e|$, we use $\perp_e$ to replace $|s|$ and the new bound T_l is computed as below.

- Jaccard Similarity: $T_l = \lceil |e| * \delta \rceil$.
- Cosine Similarity: $T_l = \lceil |e| * \delta^2 \rceil$.
- Dice Similarity: $T_l = \lceil |e| * \frac{\delta}{2-\delta} \rceil$.
- Edit Distance: $T_l = |e| - \tau * q$.
- Edit Similarity: $T_l = \lceil |e| - \big((|e| + q - 1) * \frac{(1-\delta)}{\delta} * q\big) \rceil$.

Obviously $T_l \leq T$. If $|P_e| < T_l \leq T$, e cannot be similar to any substring, and thus we can prune e (Sect. 2.2). For instance, in Fig. 5, consider e_1. Suppose $\tau = 1$. $|e_1| = 9$. $T_l = |e_1| - \tau * q = 9 - 2 = 7$. As $|P_{e_1}| = 5 < T_l$, e_1 can be pruned. The correctness is stated in Lemma 3.

Lemma 3 *Given an entity e on the single heap, if its occurrence number in the heap ($|P_e|$) is smaller than T_l, e will not be similar to any valid substring.*

Proof Since the inverted lists of D must contain those of D's any substring, e's occurrence in any valid substring must appear in D. Thus its occurrence number in any valid substring must be smaller than $T_{\min}$. Thus e will not be similar to any valid substring. □

Bucket-count pruning We have an observation that we can split the position list P_e into several disjoint buckets and apply the lazy-count pruning technique on each of the buckets. As each bucket contains a sublist list of P_e, the bucket size is much smaller than $|P_e|$ and thus each bucket has larger probability to be pruned. Next, we discuss how to split P_e into disjoint buckets.

Consider an entity e. If a valid substring s is similar to e, s must have at most $\top_e$ tokens and shares at least T_l tokens with e. In other words, if s is similar to e, they must have no larger than $\top_e - T_l$ mismatched tokens. We can use this property for effective pruning.

Formally, given two neighbor elements in list P_e, p_i and p_{i+1}, any substring containing both the two tokens (TOKEN$[p_i]$ and TOKEN$[p_{i+1}]$) has at least $p_{i+1} - p_i - 1$ mismatched tokens. If $p_{i+1} - p_i - 1 > \top_e - T_l$, any substrings containing both the two tokens cannot be similar to e. Thus we do not need to count e's occurrence numbers for any substrings.

To use this feature, we partition the elements in P_e into different buckets. If the number of elements in a bucket is smaller than T_l, we can prune all the elements in the bucket (lazy-count pruning); otherwise, we use the elements in the bucket to count e's occurrence number for valid substrings with token numbers between $\perp_e$ and $\top_e$ (Sect. 3.3).

Next, we introduce how to do the partition. Initially, we create a bucket b_1 and put the first element p_1 into the bucket. Then, we consider the next element p_2. If $p_2 - p_1 - 1 > \top_e - T_l$, we create a new bucket b_2 and put p_2 into bucket b_2; otherwise, we put p_2 into bucket b_1. Iteratively, we can partition all elements into different buckets.

We give a tighter bound for different similarity functions. For example, consider edit distance. We can use $p_{i+1} - p_i - 1 > \tau * q$ for pruning, since there exists at least $\tau * q + 1$ mismatched tokens between p_i and p_{i+1}, which need at least $\tau + 1$ single-character edit operations to destroy these $\tau * q + 1$ mismatched tokens. Similarly, for edit similarity, we can use $p_{i+1} - p_i - 1 > \lfloor \frac{(|e|+q-1)}{\delta} * (1 - \delta) * q \rfloor$ for pruning.

For example, in Fig. 5, suppose we use edit distance and $\tau = 1$. Consider $P_{e_4} = [1, 2, 3, 4, 9, 14, 19]$. $T_l = |e_4| - \tau * q = 8 - 1 * 2 = 6$. Obviously, e_4 can pass the lazy-count pruning as $|P_{e_4}| \geq T_l$. Next, we check whether it can pass the bucket-count pruning. We first partition P_{e_4} into different buckets. Initially, we create a bucket b_1 and put p_1 into the bucket. Next, for $p_2 = 2$, as $p_2 - p_1 - 1 \leq \tau * q = 2$, we put p_2 into bucket b_1. Similarly, $p_3 = 3$ and $p_4 = 4$ are added into b_1. For $p_5 = 9$, as $p_5 - p_4 - 1 > \tau * q$, we create a new bucket b_2 and add p_5 into bucket b_2. We repeat theses steps and finally get $b_1 = [1, 2, 3, 4]$, $b_2 = [9]$, $b_3 = [14]$, and $b_4 = [19]$. As the size of each bucket is smaller than T_l, we

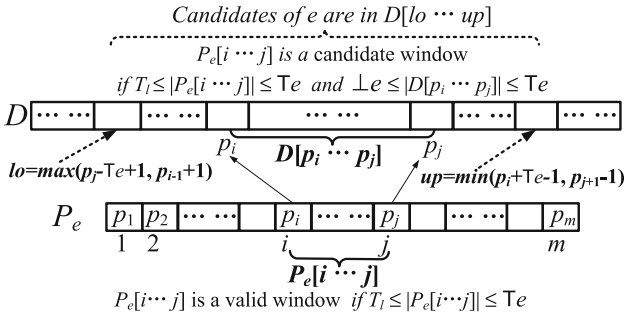


Fig. 6 Candidate window and Valid window

can directly prune the elements in each bucket. Thus, we do not need to count the occurrence number of e_4 in any valid substrings.

Moreover, we can generalize this idea: Given two elements p_i and $p_j (i < j)$, if $p_j - p_i - (j - i) > \top_e - T_l$, any substrings containing both the two tokens (TOKEN$[p_i]$ and TOKEN$[p_j]$) cannot be similar to entity e. For example, consider the document and entities in Table 1. $P_{e_4} = [10, 17, 33, 34, 43, 58, 59, 60, 61, 66, 71, 76, 81, 86]$. Suppose, we use edit distance and $\tau = 2$. $|e_4| = 8, T_l = |e_4| - \tau * q = 4$. $\perp_{e_4} = |e_4| - \tau = 6$ and $\top_{e_4} = |e_4| + \tau = 10$. If we apply the bucket-count pruning, we will get five buckets $b_1 = [10], b_2 = [17], b_3 = [33, 34], b_4 = [43]$, and $b_5 = [58, 59, 60, 61, 66, 71, 76, 81]$ and only b_5 can pass the lazy-count pruning. Nevertheless, we can further remove p_{11} from b_5 as $p_{11} - p_9 - (11 - 9) > \top_{e_4} - T_l$. Similarly, we can also remove p_{12} and p_{13} from b_5 and finally get $b_5 = [58, 59, 60, 61, 66]$. Next, we will introduce how to use this property to do further pruning.

Batch-count pruning Inspired from bucket-count pruning, we find that we do not need to enumerate each element in the position list P_e to count the occurrence numbers for every valid substring. Instead, we check sublists of P_e and check whether the sublists can produce candidate pairs. If so, we find candidate pairs in the sublists; otherwise, we prune the sublists.

Formally, if a valid substring s is similar to entity e, they must share *enough* common tokens ($|e \cap s| \geq T_l$). In other words, we only need to check the sublist with no smaller than T_l elements. Consider a sublist $P_e[i \cdots j]$ with $|P_e[i \cdots j]| = j - i + 1 \geq T_l$. Let $D[p_i \cdots p_j]$ denote the substring exactly containing tokens $token[p_i], token[p_{i+1}], \cdots, token[p_j]$ (Fig. 6). If $|D[p_i \cdots p_j]| = p_j - p_i + 1 > \top_e$, any valid substring containing all tokens in $D[p_i \cdots p_j]$ has larger than $\top_e$ tokens. Thus, we can prune $P_e[i \cdots j]$. On the contrary, $D[p_i \cdots p_j]$ may be similar to e if $\perp_e \leq |D[p_i \cdots p_j]| \leq \top_e$. This pruning technique is much power than the mismatch-based pruning, since if $p_j - p_i - (j - i) > \top_e - T_l$, then $p_j - p_i + 1 > \top_e$; on the contrary if $p_j - p_i + 1 > \top_e$, $p_j - p_i - (j - i) > \top_e - T_l$ may not hold. In addition, as $|P_e[i \cdots j]| \leq |D[p_i \cdots p_j]|$, $|P_e[i \cdots j]|$ should be not larger than $\top_e$, thus we have $T_l \leq |P_e[i \cdots j]| \leq \top_e$.

Based on this observation, we can first generate sublists of P_e with sizes (number of elements) between T_l and $\top_e$, i.e., $T_l \leq |P_e[i \cdots j]| \leq \top_e$. Then, for each such list $P_e[i \cdots j]$, if $|D[p_i \cdots p_j]| > \top_e$, we prune the sublist; otherwise, if $\perp_e \leq |D[p_i \cdots p_j]| \leq \top_e$, we find *candidates* of entity e (a substring s is a candidate of e if $|e \cap s| \geq T$ and $\perp_e \leq |s| \leq \top_e$). For each candidate s of entity e, $\langle s, e \rangle$ is a candidate pair.

Next, we discuss how to find candidates of e based on P_e. For ease of presentation, we first introduce two concepts.

Definition 3 (*Valid window and Candidate window*) Consider an entity e and its position list $P_e = [p_1 \cdots p_m]$. A sublist $P_e[i \cdots j]$ is called a window of P_e for $1 \leq i \leq j \leq m$. $P_e[i \cdots j]$ is called a valid window, if $T_l \leq |P_e[i \cdots j]| \leq \top_e$. $P_e[i \cdots j]$ is called a candidate window, if $P_e[i \cdots j]$ is a valid window and $\perp_e \leq |D[p_i \cdots p_j]| \leq \top_e$.

The valid window restricts the length of a window. The candidate window restricts the number of tokens of a valid substring. If a valid substring is a candidate of entity e, it must contain a candidate window (Fig. 6). Recall the above example. $P_{e_4} = [10, 17, 33, 34, 43, 58, 59, 60, 61, 66, 71, 76, 81, 86]$. The edit distance $\tau = 2$. $|e_4| = 8, T_l = |e_4| - \tau * q = 4$. $\perp_{e_4} = |e_4| - \tau = 6$ and $\top_{e_4} = |e_4| + \tau = 10$. $P_{e_4}[1 \cdots 4] = [10, 17, 33, 34]$, $P_{e_4}[1 \cdots 5] = [10, 17, 33, 34, 43]$, and $P_{e_4}[6 \cdots 9] = [58, 59, 60, 61]$ are three valid windows. $P_{e_4}[1 \cdots 4]$ is not a candidate window as $p_4 - p_1 + 1 = 34 - 10 + 1 > \top_{e_4}$. The reason is that $D[p_1 \cdots p_4]$ contains more than $\top_{e_4}$ tokens and any substring containing $P_{e_4}[1 \cdots 4]$ must have more than $\top_{e_4}$ tokens. Although $p_9 - p_6 + 1 \leq \top_{e_4}$, $P_{e_4}[6 \cdots 9]$ is not a candidate window as $p_9 - p_6 + 1 < \perp_{e_4}$.

Notice that we can optimize the pruning condition for jaccard similarity, cosine similarity, and dice similarity, since they depend on $|e \cap s|$. Given a valid window $P_e[i \cdots j]$, let $s = D[p_i \cdots p_j]$. $|P_e[i \cdots j]| \geq |e \cap s|$.[5] Take jaccard similarity as an example. If s and e are similar, $\frac{|e \cap s|}{|e \cup s|} \geq \delta$. $|D[p_i \cdots p_j]| = |s| \leq |e \cup s| \leq \frac{|e \cap s|}{\delta} \leq \frac{\min(|e|, |P_e[i \cdots j]|)}{\delta}$. Thus, we give a tighter bound of $|D[p_i \cdots p_j]|$. For jaccard similarity, $\perp_e \leq |D[p_i \cdots p_j]| \leq \frac{\min(|e|, |P_e[i \cdots j]|)}{\delta}$; for cosine similarity, $\perp_e \leq |D[p_i \cdots p_j]| \leq \frac{\min(|e|, |P_e[i \cdots j]|)}{\delta^2}$; for dice similarity, $\perp_e \leq |D[p_i \cdots p_j]| \leq \min(|e|, |P_e[i, j]|) * \frac{2-\delta}{\delta}$.

Now, we introduce how to find candidates of e from candidate windows $P_e[i \cdots j]$. The substrings that contain all tokens in $D[p_i \cdots p_j]$ may be candidates of e. We can find

[5] As $D[p_i \cdots p_j]$ may contain duplicate tokens, $|P_e[i \cdots j]| \geq |e \cap s|$ and $|P_e[i \cdots j]|$ may also be larger than $|e|$.

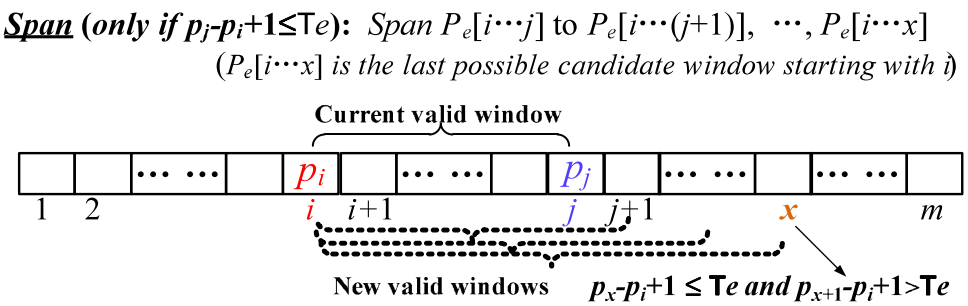


Fig. 7 span operation

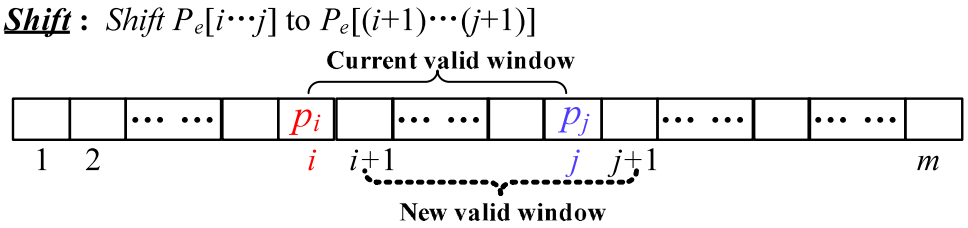


Fig. 8 shift operation

these substrings as follows. As these substrings must contain $token[p_i]$, the "maximum start position" of such substrings is p_i and the "maximum end position" is $\mathsf{up} = p_i + \top_e - 1$. Similarly, as these substrings must contain $token[p_j]$, the "minimum start position" is $\mathsf{lo} = p_j - \top_e + 1$ and the "minimum end position" is p_j. Thus, we only need to find candidates among substrings $D[p_{start} \cdots p_{end}]$ where $\mathsf{lo} \le p_{start} \le p_i, p_j \le p_{end} \le \mathsf{up}$. Substring $s = D[p_{start} \cdots p_{end}]$ is a candidate of e if $\perp_e \le |s| = p_{end} - p_{start} + 1 \le \top_e$ and $|e \cap s| \ge T$. (Here, we use threshold T as $s = D[p_{start} \cdots p_{end}]$ is known.)

However, this method may generate duplicate candidates. For example, suppose $p_j - \top_e + 1 < p_{i-1} + 1$. $D[p_{i-1}, \top_e] = D[p_{i-1} \cdots (p_{i-1} + \top_e - 1)]$ could be a candidate generated from $P_e[i \cdots j]$. In this case, as $\perp_e \le p_j - p_i + 1 \le p_j - p_{i-1} + 1 \le \top_e$ and $T_l \le |P_e[i \cdots j]| \le |P_e[i-1 \cdots j]| = p_j - p_{i-1} + 1 \le \top_e$, $P_e[(i-1) \cdots j]$ is also a candidate window. Thus, $P_e[(i-1) \cdots j]$ also generates the candidate $D[p_{i-1}, \top_e]$. For $P_e[i \cdots j]$, to avoid generating duplicates with $P_e[i-1 \cdots j]$ and $P_e[i \cdots j+1]$, we will not extend $P_e[i \cdots j]$ to positions smaller than $p_{i-1} + 1$ and larger than $p_{j+1} - 1$, and set $\mathsf{lo} = \max(p_j - \top_e + 1, p_{i-1} + 1)$, $\mathsf{up} = \min(p_i + \top_e - 1, p_{j+1} - 1)$. In this way, our method will not generate duplicate candidates.

In summary, to find all candidates for an entity, we first get the entity's position list, and then generate the valid windows and candidate windows. Next, we identify its candidates from candidate windows. Finally, the pair of each candidate and the entity is a candidate pair.

4.2 Finding candidate windows efficiently

Given an entity e, as there are larger numbers of valid windows ($\sum_{l=T_l}^{\top_e} |P_e| - l + 1$), it could be expensive to enumerate the valid windows for finding all candidate windows. To improve the performance, this section proposes efficient methods to find candidate windows.

Span and Shift based method: For ease of presentation, we first introduce a concept "possible candidate windows." A valid window $P_e[i \cdots j]$ is called a *possible candidate window* if $p_j - p_i + 1 \le \top_e$. Based on this concept, we introduce two operations: span and shift. Given a current valid window $P_e[i \cdots j]$, we use the two operations to generate new valid windows as follows.

- span: As shown in Fig. 7, if $p_j - p_i + 1 \le \top_e$, for $k \ge j$, $P_e[i \cdots k]$ may be a possible candidate window. We span it to generate all possible candidate windows starting with i : $P_e[i \cdots (j+1)], \ldots, P_e[i \cdots x]$, where x satisfies $p_x - p_i + 1 \le \top_e$ and $p_{x+1} - p_i + 1 > \top_e$. For $j \le k \le x$, if $p_k - p_i + 1 \ge \perp_e$, $P_e[i \cdots k]$ is a candidate window. On the contrary, if $p_j - p_i + 1 > \top_e$, for $k \ge j$, as $p_k - p_i + 1 \ge p_j - p_i + 1 > \top_e$, $P_e[i \cdots k]$ cannot be a candidate window. Thus, we do not need to span $P_e[i \cdots j]$.
- shift: As shown in Fig. 8, we shift to a new valid window $P_e[(i+1) \cdots (j+1)]$.

We use the two operations to find candidate windows as follows. Initially, we get the first valid window $P_e[1 \cdots T_l]$. We do span and shift operations on $P_e[1 \cdots T_l]$. For the new valid windows generated from the span operation, we check whether they are candidate windows; for the new valid window generated from the shift operation, we do span and shift operations on it. Iteratively, we can find all candidate windows from $P_e[1 \cdots T_l]$. We give an example to show how our method works. For e_4("venkatesh"), $P_{e_4} = [10, 17, 33, 34, 43, 58, 59, 60, 61, 66, 71, 76, 81, 86]$. Suppose $\tau = 2$. $|e_4| = 8, T_l = |e_4| - \tau * q = 4, \perp_{e_4} =$

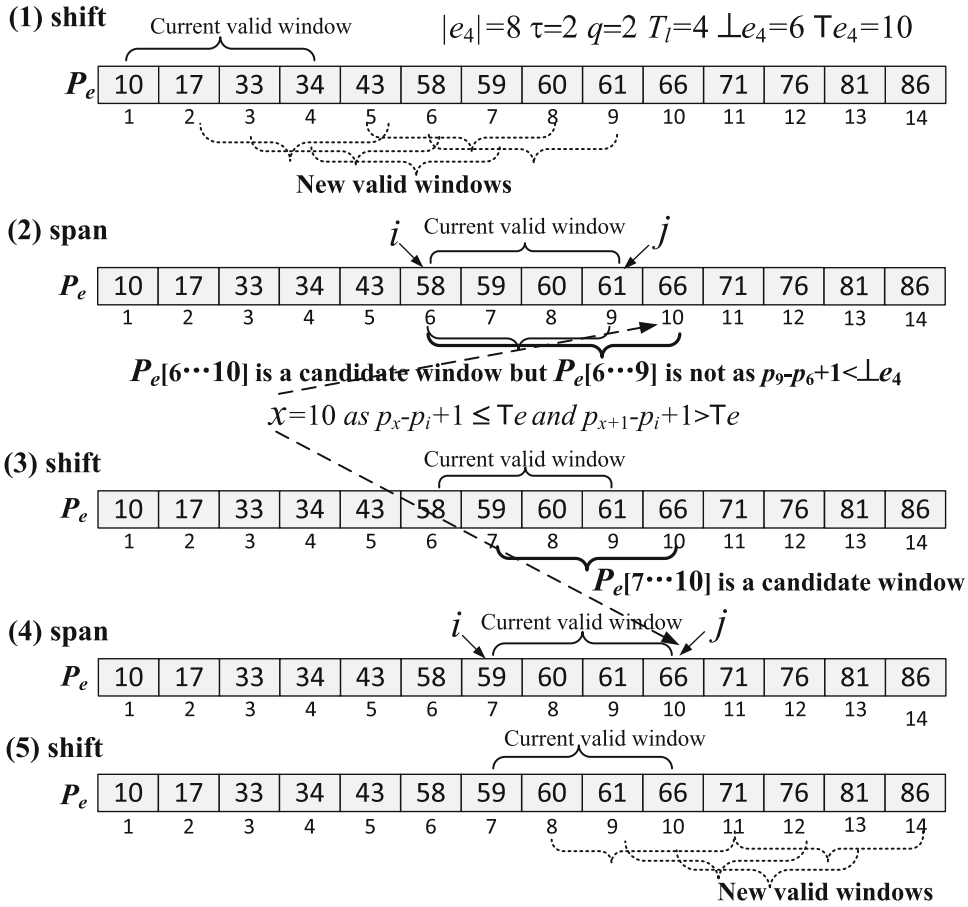


Fig. 9 An example for span and shift operations

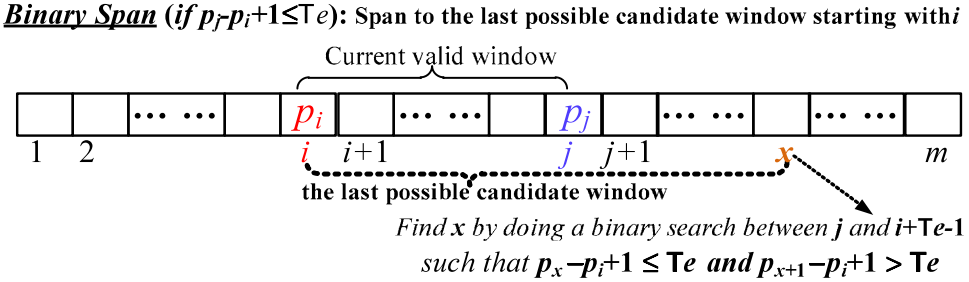


Fig. 10 span operation in a binary-search way

$|e_4| - \tau = 6, \top_{e_4} = |e_4| + \tau = 10$. The first valid window is $P_{e_4}[1 \cdots 4] = [10, 17, 33, 34]$. As $p_4 - p_1 + 1 = 34 - 10 + 1 > \top_{e_4}$, we do not need to do span operation. We do a shift operation and get the next window $P_{e_4}[2 \cdots 5]$. As $p_5 - p_2 + 1 = 43 - 17 + 1 > \top_{e_4}$, we do another shift operation. When we shift to valid window $P_{e_4}[6 \cdots 9]$, as $p_9 - p_6 + 1 = 61 - 58 + 1 < \perp_{e_4} \leq \top_{e_4}$, $P_{e_4}[6 \cdots 9]$ is not a candidate window. We do a span operation. As $p_{10} - p_6 + 1 = 9 \leq \top_{e_4}$ and $p_{11} - p_6 + 1 = 14 > \top_{e_4}, x = 10$. We get a valid window $P_{e_4}[6 \cdots 10]$, which is a candidate window. Next, we shift to $P_{e_4}[7 \cdots 10]$. Iteratively, we find all candidate windows: $P_{e_4}[6 \cdots 10]$ and $P_{e_4}[7 \cdots 10]$ (Fig. 9).

Given a valid window $P_e[i \cdots j]$, if $p_j - p_i + 1 > \top_e$, the shift operations can prune the valid windows starting with i, e.g., $P_e[i \cdots k]$ for $j < k \leq i + \top_e - 1$. However, this method still scans large numbers of candidate windows. To further improve performance, we propose a binary-search-based method which can skip many more valid windows.

Binary Span and Shift based method: The basic idea is as follows. Given a valid window $P_e[i \cdots j]$, if $p_j - p_i + 1 > \top_e$, we will not shift to $P_e[(i+1) \cdots (j+1)]$. Instead, we want to directly shift to the *first possible candidate window* after i, denoted by $P_e[mid \cdots (mid + j - i)]$, where mid satisfies $p_{mid+j-i} - p_{mid} + 1 \leq \top_e$ and for any $i \leq mid' < mid$, $p_{mid'+j-i} - p_{mid'} + 1 > \top_e$. Similarly, if $p_j - p_i + 1 \leq \top_e$, we will not iteratively span it to $P_e[i \cdots (j+1)]$, $P_e[i \cdots (j+2)]$, ..., $P_e[i \cdots x]$. Instead, we want to directly span to the *last possible candidate window* starting with i, denoted by $P_e[i \cdots x]$, where x satisfies $p_x - p_i + 1 \leq \top_e$ and for any $x' > x$, $p_{x'} - p_i + 1 > \top_e$.

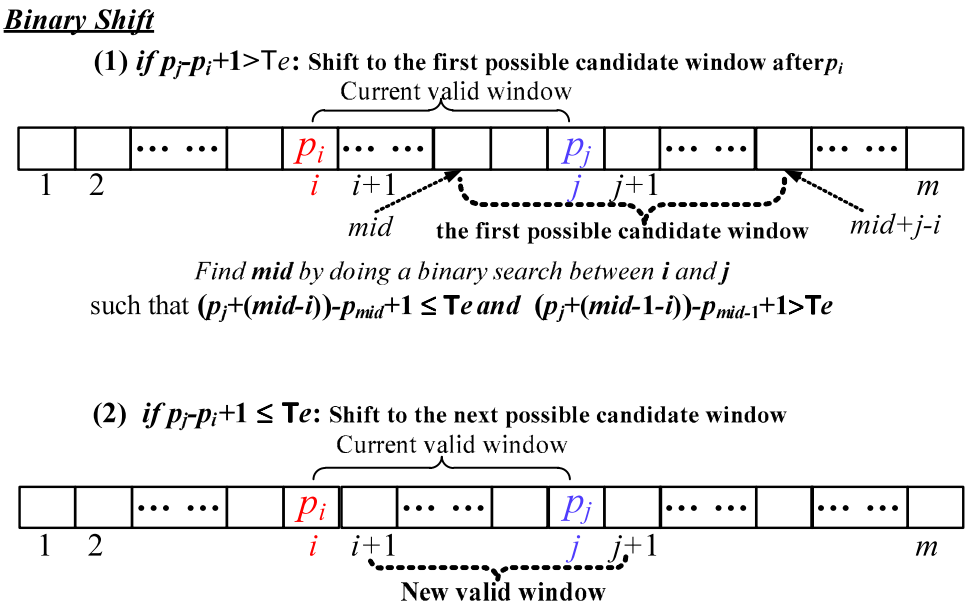


Fig. 11 shift operation in a binary-search way

If the function $F(x) = p_x - p_i + 1$ for span and $F'(mid) = p_{mid+j-i} - p_{mid} + 1$ for shift are monotonic, we can use a binary-search method to find x and mid efficiently.

For the span operation, obviously $F(x) = p_x - p_i + 1$ is monotonic as $F(x+1) - F(x) = p_{x+1} - p_x > 0$. Next, we give the lower bound and upper bound of the search range. Obviously, $x \ge j$. In addition, as $p_i + j \le p_{i+j}$, we have $p_x \le p_i + \top_e - 1 \le p_{i+\top_e-1}$ and $x \le i + \top_e - 1$. Thus, we find x by doing a binary search between j and $i + \top_e - 1$.

However, $F'(mid) = p_{mid+j-i} - p_{mid} + 1$ is not monotonic. We have an observation that $F''(mid) = \big(p_j + (mid - i)\big) - p_{mid} + 1$ is monotonic, since $F''(mid-1) - F''(mid) = p_{mid} - p_{mid-1} - 1 \ge 0$. More importantly, for $i \le mid \le j$, $F''(mid) < F'(mid)$ as $\big(p_j + (mid - i)\big) \le p_{mid+j-i}$. Thus if $F''(mid-1) > \top_e$, $F'(mid-1) > \top_e$ and $P_e[(mid-1)\cdots(mid-1+j-i)]$ cannot be a candidate window. If $F''(mid) \le \top_e$, $P_e[(mid)\cdots(mid+j-i)]$ could be a candidate window. In this way, we can find mid by doing a binary search between i and j such that $F''(mid-1) > \top_e$ and $F''(mid) \le \top_e$. If $F'(mid) \le \top_e$, we have found the last possible candidate window; otherwise, we continue to find mid' between $mid+1$ and $mid+1+j-i$. Iteratively, we can find the last possible candidate window.

Thus, given a valid window $P_e[i \cdots j]$, we use binary span and shift operations to find candidate windows.

- Binary span: As shown in Fig. 10, if $p_j - p_i + 1 \le \top_e$, we first find x by doing a binary search between j and $i + \top_e - 1$, where x satisfies $p_x - p_i + 1 \le \top_e$ and $p_{x+1} - p_i + 1 > \top_e$, and then directly span to $P_e[i \cdots x]$.
- Binary shift: As shown in Fig. 11, if $p_j - p_i + 1 > \top_e$, we find mid by doing a binary search between i and j where mid satisfies $\big(p_j + (mid - i)\big) - p_{mid} + 1 \le \top_e$ and $\big(p_j + (mid - 1 - i)\big) - p_{mid-1} + 1 > \top_e$. If $p_{mid+j-i} - p_{mid} + 1 > \top_e$, we iteratively do the binary shift operation between $mid + 1$ and $mid + 1 + j - i$. On the contrary, if $p_j - p_i + 1 \le \top_e$, we shift to a new valid window $P_e[(i+1)\cdots(j+1)]$.

Given a valid window $P_e[i \cdots j]$, the binary shift can skip unnecessary valid windows (non-candidate windows), such as $P_e[(i+1)\cdots(j+1)], \ldots, P_e[(mid-1)\cdots(mid-1+j-i)]$, as proved in Lemma 4. For example, consider the position list in Fig. 12. Suppose $\tau = 2$. $T_l = 4$, $|e_4| = 8$, $\top_{e_4} = 10$. Consider the first valid window $P_{e_4}[1 \cdots 4]$. The shift operation shifts it to $P_{e_4}[2 \cdots 5]$, $P_{e_4}[3 \cdots 6]$, $\cdots$, $P_{e_4}[6 \cdots 9]$, and checks whether they are candidate windows. The binary shift operation can directly shift it to $P_{e_4}[3 \cdots 6]$ and then to $P_{e_4}[6 \cdots 9]$. Thus it skips many valid windows.

Lemma 4 *Given a valid window $P_e[i \cdots j]$ with $p_j - p_i + 1 > \top_e$, if $\big(p_j + (mid - i)\big) - p_{mid} + 1 \le \top_e$ and $\big(p_j + (mid-1) - i\big) - p_{mid-1} + 1 > \top_e$, $P_e[(i+1)\cdots(j+1)], \ldots, P_e[(mid-1)\cdots\big((mid-1)+j-i\big)]$ are not candidate windows.*

Proof We prove it by contradiction. Suppose there exists a valid window $P_e[k \cdots (j+k-i)](i \le k \le mid - 1)$, $P_e[k \cdots (j+k-i)]$ is a candidate window. That is $p_{j+k-i} - p_k + 1 \le \top_e$. As $p_j + k - i \le p_{j+k-i}$, $p_j + k - i - p_k + 1 \le \top_e$. Thus $p_j + k - i + (mid - 1 - k) - (p_k + (mid - 1 - k)) + 1 \le \top_e$, and $p_j + (mid-1) - i - (p_{mid-1}) + 1 \le \top_e$. This contradicts with $p_j + (mid-1) - i - p_{mid-1} + 1 > \top_e$. Thus $P_e[(i+1)\cdots(j+1)], \ldots, P_e[(mid-1)\cdots(j+mid-1-i)]$ are not candidate windows. □

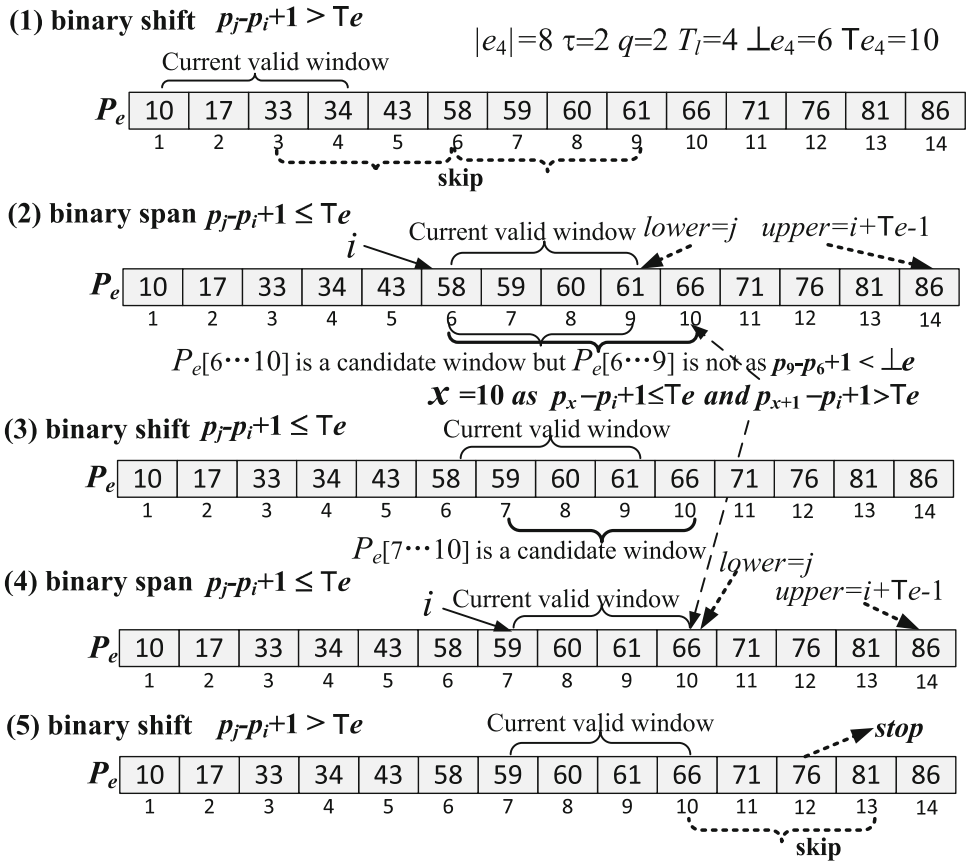


Fig. 12 An example for binary span and shift

The binary span operation directly spans to $P_e[i \cdots x]$ and has two advantages. Firstly, in many applications, users want to identify the best similar pairs (sharing common tokens as many as possible), and the binary span can efficiently find such substrings. Secondly, we do not need to find candidates of e for $P_e[i \cdots (j+1)], \ldots, P_e[i \cdots x]$ one by one. Instead, since there may be many candidates between $lo = p_j - \top_e + 1$ and $up = p_{i+x-j} + \top_e - 1$, we find them in a batch manner. We group the candidates based on their token numbers. Entities in the same group have the same number of tokens. Consider the group with g tokens, suppose T_g is the threshold computed using $|e|$ and g. If $|P_e[i \cdots x]| < T_g$, we prune all candidates in the group.

We use the two binary operations to replace the shift and span operations to skip valid windows. We devise an algorithm to find candidate windows using the two operations as illustrated in Fig. 13. It first initializes the first valid window (line 2–line 4). Then, it uses the two binary operations to extend the valid window until reaching the last valid window (line 3). If its token number is no larger than $\top_e$, we do a binary span operation by calling its subroutine BinarySpan (line 6) and do a shift operation (line 7); otherwise calling its subroutine BinaryShift (line 8). BinarySpan does a binary search to find the last possible candidate window starting with p_i (lines 3-6). Then, it retrieves the candidate windows (line 8). BinaryShift does a binary search to find the first possible candidate window after p_i. Iteratively, our method finds all candidate windows. Figure 12 illustrates an example to walk through our algorithm.

Complexity of the improved single-heap-based method Different from the basic single-heap-based method, the improved method does not count the occurrence number using the array. Instead, for each position list P_e, it finds candidate windows. For any list P_e, in the worst case, its sublists with sizes between $\perp_E$ and $\top_E$ are candidate windows. Thus, the time complexity for finding its candidate windows is $|P_e| * \top_E$. The method requires to find candidate windows for each position list, and the sum of lengths of position lists is the sum of lengths of inverted lists, and thus, the time complexity of finding candidate windows using the improved method is $\mathcal{O}(N * \top_E)$. Based on the candidate windows, it extends the candidate windows to find candidates and the complexity is the number of candidates.

4.3 The Single algorithm

In this section, we propose a single-heap-based algorithm, called Single, to efficiently find all answers.

Algorithm 1: Find Candidate Windows

```
Input: e: An entity; P_e: Position list of e on the heap;
       T_l: Threshold; ⊤_e: The upper bound of token numbers;
1  begin
2      i = 1;
3      while i ≤ |P_e| − T_l + 1 do
4          j = i + T_l − 1;
5          if p_j − p_i + 1 ≤ ⊤_e then
6              BinarySpan(i, j);
7              i = i + 1; /* shift to the next window */
8          else i = BinaryShift(i, j);
9  end
```

Procedure BinarySpan(i, j)

```
Input: i: the start point; j: the end point;
1  begin
2      lower = j; upper = i + ⊤_e − 1 ;
3      while lower ≤ upper do
4          mid = ⌈(upper + lower)/2⌉;
5          if p_mid − p_i + 1 > ⊤_e then upper = mid − 1;
6          else lower = mid + 1;
7      mid = upper ;
8      Find candidate windows in D[i ··· mid];
9  end
```

Procedure BinaryShift(i, j)

```
Input: i: the start point; j: the end point
Output: i: the new start point;
1  begin
2      lower = i; upper = j;
3      while lower ≤ upper do
4          mid = ⌈(lower + upper)/2⌉;
5          if (p_j + (mid − i)) − p_mid + 1 > ⊤_e then
6              lower = mid + 1;
7          else upper = mid − 1;
8      i = lower; j = i + T_l − 1;
9      if p_j − p_i + 1 > ⊤_e then i = BinaryShift (i, j);
10     else return i;
11 end
```

Fig. 13 Algorithm: Find candidate windows

We first construct an inverted index for all entities in the given dictionary E. Then, for the document D, we get its tokens and corresponding inverted lists. Next, we construct a single heap on top of inverted lists of tokens. We use the heap to generate entities in ascending order. For each entity e, we get its position list P_e. If $|P_e| < T_l$, we prune e based on lazy-count pruning; otherwise we use the two binary operations to find candidate windows. Then, based on the candidate windows, we generate candidate pairs. Finally, we verify the candidate pairs and get the final results. Figure 14 gives the pseudo-code of the Single algorithm.

The Single algorithm first constructs an inverted index for predefined entities (line 2), and then tokenizes the document, gets inverted lists (line 3), and constructs a heap (line 4). Single uses $\langle e_i, p_i \rangle$ to denote the top element of the heap, where e_i is the current minimal entity and p_i is the position of the inverted list from which e_i is selected. Single constructs a position list P_e to keep all the positions of inverted lists in the heap that contain e (line 6). Then, for each top element $\langle e_i, p_i \rangle$ on the heap, if $e_i = e$, we add p_i into P_e (line 8–line 9), where e is the last popped entity from the heap; otherwise Single checks the position list as follows. Single derives a threshold T_l for entity e based on the similarity function and threshold. If $|P_e| \geq T_l$, there may exist candidate pairs. Single generates candidate windows based on Algorithm 1 (line 13) and finds candidate pairs based on candidate windows (line 14). Single adjusts the heap to generate candidates for the next entity (line 16). Finally, Single verifies the candidates to get the final results (line 17).

Next, we give a running example to walk through our algorithm. Consider the entities and document in Table 1. We fist construct a single min-heap (Fig. 5). Then, we adjust the heap to generate the position list for each entity. Consider the position list for entity e_4 ("venkatesh") in Fig. 12. Suppose $\tau = 2$. $|e_4| = 8$, $\perp_E = 6$, $\top_E = 12$, $\perp_{e_4} = 6$, $\top_{e_4} = 10$, $T_l = 4$. We use the binary shift and span operations to get candidate windows ($P_e[6 \cdots 10]$ and $P_e[7 \cdots 10]$), and then generate candidate pairs based on the candidate windows (e.g., $\langle D[58, 9]$="*venkaee sh*", e_4="venkatesh"$\rangle$). Finally we verify the candidates to get the final answers.

4.4 Correctness and completeness

In this section, we prove the correctness and completeness of the Single algorithm as stated in Theorem 1.

Theorem 1 *The Single algorithm extracts all similar substrings from the document correctly and completely.*

Proof We first prove the Single algorithm finds answers completely. That is given a similar substring and entity pair $\langle s, e \rangle$, the Single algorithm can find this pair as a result. As s and e are similar, based on Lemma 1, $|e \cap s| \geq T_l$. For the position list P_e, we have $|P_e| \geq |e \cap s| \geq T_l$. Thus, the position list P_e can pass the lazy-count pruning and the Single algorithm will find its candidate windows. As s and e are similar, based on Lemma 2 we have $\perp_e \leq |s| \leq \top_e$. Without loss of generality, suppose $P_e[i]$ and $P_e[j]$ are the first and last common token positions of s and e. The Single algorithm must find $P_e[i \ldots j]$ as a candidate window because $T_l \leq |P_e[i \ldots j]| \leq |s| \leq \top_e$. In the verification stage, the Single algorithm must find $\langle s, e \rangle$ as a result. Thus, the Single algorithm satisfies the completeness property.

Next, we prove the correctness of the Single algorithm. That is all the substring and entity pairs found by the Single algorithm must be similar pairs. As the Single algorithm has a verification stage which only outputs those sim-

Algorithm 2: Single Algorithm

Input: A dictionary of entities $E = \{e_1, e_2, \ldots, e_n\}$;
A document D;
A similarity function and a threshold;
Output: $\{\langle s, e\rangle |\ s$ and e are similar for the function and threshold$\}$, where s is a substring of D and $e \in E$.

```
1  begin
2      Tokenize entities in E and construct an inverted index;
3      Tokenize D and get inverted lists of tokens in D;
4      Construct a heap H on top of inverted lists of D;
5      e is the top element of H; /* keep the current entity*/
6      Initialize a position list P_e = φ;
7      while (⟨e_i, p_i⟩ = H.top)! = φ do
8          if e_i == e then
9              P_e ∪ = {p_i}; /* e_i is the new top entity. */
10         else
11             Derive the threshold T_l for entity e;
12             if |P_e| ≥ T_l then
13                 Find candidate windows using Algorithm 1;
14                 Get candidates using candidate windows;
15             e = e_i; P_e = {p_i}; // update the current entity
16         Adjust the heap;
17     Verify candidate pairs;
18 end
```

Fig. 14 The Single algorithm

ilar pair, thus the Single algorithm satisfies the correctness property. □

5 The hybrid method

Both of the multi-heap-based method and the single-heap-based method require to construct heaps, access inverted lists of tokens, and adjust heaps to count the occurrence number. The multi-heap-based method builds many heaps and accesses some inverted lists multiple times (if the corresponding tokens appear in different heaps). The single-heap-based method constructs a single heap and accesses each inverted list once. It is worth noting that the single-heap-based method contains more tokens in the heap and the cost for each heap-adjustment operation is more expensive than the multi-heap-based method. Thus, there is a trade-off between the heap-adjustment cost and the inverted-list-access cost.

To address this issue, we propose a hybrid method by combining the two methods. The basic idea is to split the document into multiple fragments and utilize the single-heap based method on each fragment to generate candidate pairs. Obviously, the hybrid method can reduce the heap-adjustment cost compared with the single-heap-based

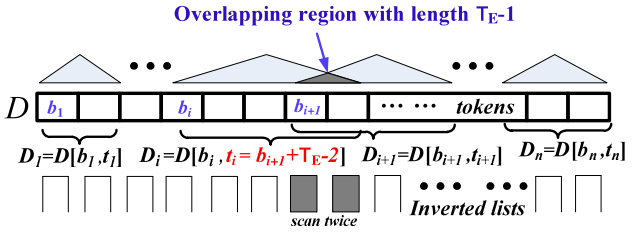


Fig. 15 The hybrid method

method. If the fragments do not share many common tokens, the hybrid method does not require to access many inverted lists multiple times, and thus, it can achieve much better performance than the single-heap-based and the multi-heap-based methods. To achieve this goal, in this section, we study how to judiciously partition the documents. We first discuss how to quantify a partition method (Sect. 5.1) and then devise algorithms to generate high-quality fragments (Sects. 5.2, 5.3). Next we discuss some optimization techniques (Sect. 5.4). Finally we introduce our hybrid algorithm (Sect. 5.5).

5.1 Quantifying document fragmentation strategies

Since there are many document fragmentation strategies, we discuss how to quantify them. For ease of presentations, we first introduce some notations. We first tokenize the document D and get a list of tokens. For each token, we retrieve the corresponding inverted list. We use $token[i]$ to denote the i-th token, and $\mathcal{IL}[i]$ to denote the inverted list of the i-th token. Then, we split the document D into n fragments $D_1, D_2, \ldots, D_n$ as shown Figure 15. Next, for each fragment D_i, we build a min-heap on top of its tokens with non-empty inverted lists. We utilize the single-heap based method on each heap to find candidate answers in D_i. It is worth noting that there are some valid substrings of D composed of tokens in two adjacent fragments D_i and D_{i+1}. Obviously, we cannot use either D_i or D_{i+1} to find these valid substrings as candidates. To avoid involving false negatives, two adjacent fragments must share an overlap region with length $\top_E - 1$, where $\top_E$ is the maximum length of valid substrings (defined in Sect. 2.3), because any substring with length larger than $\top_E$ cannot be a valid substring (see the definition of valid substring in Sect. 2.3) and any valid substring with length no longer than $\top_E$ can be found in D_i or D_{i+1}. Next, we analyze the time complexity of the hybrid method.

Without loss of generality, suppose the begin position of tokens in D_i is b_i where $b_1 = 1$. The end position of D_i is $t_i = b_{i+1} + \top_E - 2$. To be consistent, we manually set $b_{n+1} = |D| - \top_E + 2$. We also use k_j to denote the number of entities in $\mathcal{IL}[j]$, thus the number of entities in all inverted lists of D_i is $\sum_{j=b_i}^{t_i} k_j$. Based on the complexity analysis in Sects. 3.3 and 4.2, the time complexity for finding candidate

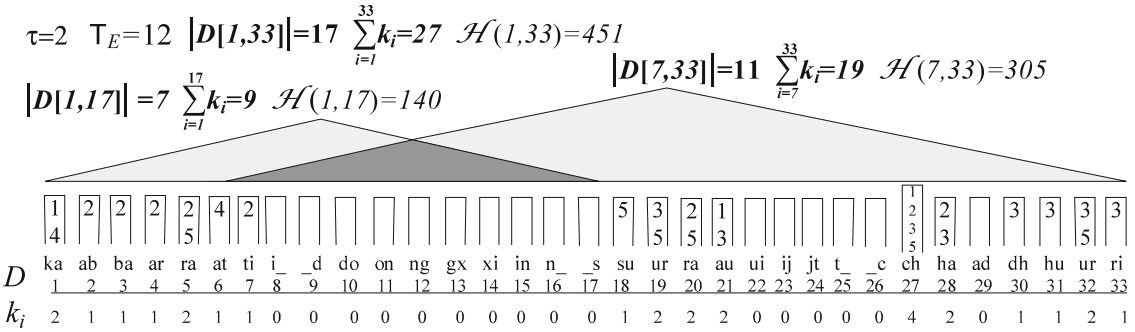


Fig. 16 An example of the hybrid method

pairs from D_i is

$$\mathcal{H}(b_i, t_i) = |D_i| + \log(|D_i|) * \sum_{j=b_i}^{t_i} k_j + \sum_{j=b_i}^{t_i} k_j * \top_E, \quad (1)$$

where the first one is the heap-construction cost, the second one is the heap-adjustment cost and the third one is finding the candidate windows cost.[6]

Thus, given a document fragment strategy with start positions $b_1, b_2, \ldots, b_n$, we can figure out the total cost $\mathcal{C}$ needed to find candidate pairs from the document D, i.e., $\mathcal{C} = \sum_{i=1}^{n} \mathcal{H}(b_i, b_{i+1} + \top_E - 2)$. We want to find a fragment strategy to minimize the cost. Next, we formally define the optimal document fragmentation problem.

Definition 4 (*Optimal Document Fragmentation*) Given a document D, a dictionary E, a similarity function and a similarity threshold, the optimal document fragmentation problem is to find a list of begin positions $F = \{b_1, b_2 \ldots b_n\}$ to fragment the document D in order to minimize $\mathcal{C} = \sum_{i=1}^{n} \mathcal{H}(b_i, b_{i+1} + \top_E - 2)$.

Next, we use an example to show our idea. Consider a document "`kabarati_dongxin_surauijt_chadhuri`" as shown in Fig. 16. If we use the single-heap based method and only build one heap over the document, as $|D| = 17$, $\sum_{i=1}^{33} k_i = 27$ and $\top_E = 12$, the cost is $\mathcal{H}(1, 33) = 17 + log(17) * 27 + 27 * 12 = 451$. If we split the document into two fragments with start positions $b_1 = 1$ and $b_2 = 7$, the total cost is $\mathcal{H}(1, 17) + \mathcal{H}(7, 33) = (7 + log(7) * 9 + 9 * 12) + (11 + log(11) * 19 + 19 * 12) = 445$. Thus, the hybrid method can improve the performance.

5.2 Optimal algorithm to document fragmentation

In this section, we propose a dynamic programing algorithm to solve the optimal document fragmentation problem. For ease of presentation, we first given some notations. We use $\mathcal{H}(p, q)$ to denote the cost needed to find candidate pairs from fragment $D[p \ldots q]$ by utilizing the single-heap based method. We use $\mathcal{C}(p)$ to denote the minimum cost needed to find all candidate pairs from $D[1 \ldots p]$. Thus, $\mathcal{C}(|D|)$ is the minimum cost needed to find candidate pairs from D. Next, we discuss how to calculate the minimum cost $\mathcal{C}(t)$ which is the minimum cost to generate optimal document fragments before the t-th token. Suppose the begin position of the last fragment in $D[1 \ldots t]$ is b. Then, the cost of finding candidate pairs from $D[1 \ldots t]$ is $\mathcal{C}(b + \top_E - 2) + \mathcal{H}(b, t)$. Among all possible value of b, we need to select the one with the minimum cost and use it as the last begin position. As we need to keep an overlap region with length $\top_E - 1$, the value of b cannot be larger than $t - \top_E + 1$. Thus, we have

$$\mathcal{C}(t) = \min_{1 \le b \le t - \top_E + 1} \big(\mathcal{C}(b + \top_E - 2) + \mathcal{H}(b, t)\big). \quad (2)$$

Based on the begin position b with the minimum cost and the positions in $F(b - \top_E - 2)$ that achieves the optimal fragmentation of the document $D[1 \ldots b - \top_E - 2]$, we can get $F(t)$. For $t \le \top_E - 1$, $\mathcal{C}(t) = 0$ as there is no valid b and we do not need to construct min-heap over $D[1 \ldots t]$. By initializing $\mathcal{C}(t \le \top_E - 1) = 0$ and $F(t \le \top_E - 1) = \{1\}$, we can recursively find the optimal document fragmentation strategy $F(|D|)$ and its minimum cost $\mathcal{C}(|D|)$.

The pseudo code of the optimal document fragmentation algorithm is shown in Fig. 17. It takes t as input which corresponds to fragment document $D[1 \ldots t]$ and outputs the list of begin positions $F(t)$ to optimally fragment $D[1 \ldots t]$ with its cost $\mathcal{C}(t)$. First, the algorithm checks if the list of begin positions $F(t)$ has been calculated. If yes, it directly returns $F(t)$ and the cost $\mathcal{C}(t)$ (Line 2). Otherwise, if $t \le \top_E - 1$, it returns the initial value $F(t) = \{1\}$ and $\mathcal{C}(t) = 0$ (Lines 3). Next, the algorithm finds the begin position b with minimum cost among all possible values (Lines 4 to 6). Finally, the algorithm merges $\{b\}$ with $F(b + \top_E - 2)$ to get $F(t)$ and returns it with the cost $\mathcal{C}(t)$ (Lines 7 to 9).

We continue the last example. $\top_E = 12$. To get the optimal document fragment strategy, we initialize $\mathcal{C}(\top_E - 1 =$

[6] Notice that we do not consider finding candidates from candidate windows as such cost is same for any strategy.

Algorithm 3: `OptFragment`

Input: t: A position to fragment document $D[1 \dots t]$
Output: $F(t)$: begin position list to fragment $D[1 \dots t]$;
$\mathcal{C}(t)$: minimum cost to find candidates in $D[1 \dots t]$;

```
1  begin
2      if F(t) is not empty then return ⟨F(t), C(t)⟩;
3      if t ≤ ⊤_E − 1 then return ⟨F(t) = {1}, C(t) = 0⟩;
4      for b = 1 to t − ⊤_E + 1 do
5          ⟨F(b+⊤_E−2), C(b+⊤_E−2)⟩ = OptFragment(b+⊤_E−2);
6          if C(b + ⊤_E − 2) + H(b, t) is minimum then
7              F(t) = {b} ∪ F(b + ⊤_E − 2);
8              C(t) = C(b + ⊤_E − 2) + H(b, t);
9              return ⟨F(t), C(t)⟩;
10 end
```

Fig. 17 Algorithm: optimal document fragmentation

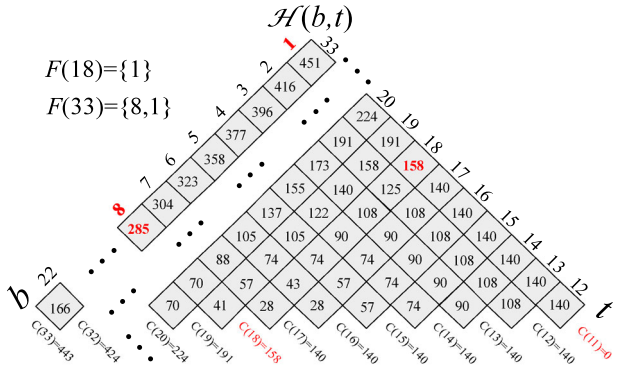


Fig. 18 An example of the `OptFragment` algorithm

$11) = 0$ and call `OptFragment(t=33)` to find b from the range $[1, t-\top_E+1 = 22]$ to minimize $\mathcal{C}(b+10)+\mathcal{H}(b, 33)$. We recursively call Algorithm 3 to figure out $\mathcal{C}(b + 10)$. The results are shown in Fig. 18. The numbers in the matrix are the value of $\mathcal{H}(b, t)$, and the formulas in the bottom are gotten by Algorithm 3. We find that when $b = 8, \mathcal{C}(33) = \mathcal{C}(b+10)+\mathcal{H}(b, 33)$ achieves minimum value. We add $b = 8$ into the begin position list $F(33)$ and further find the optimal strategy to fragment document $D[1, 18]$. Finally, we got $F(33) = \{1, 8\}$ which means we fragment the document into two fragments $D[1, 18]$ and $D[8, 33]$ with minimum cost $\mathcal{C}(33) = \mathcal{H}(1, 18) + \mathcal{H}(8, 33) = 158 + 285 = 443$.

Complexity From the procedure `OptFragment`, we can see that to get $F(|D|)$ and $\mathcal{C}(|D|)$ we need to calculate all $F(t)$ and $\mathcal{C}(t)$ where $\top_E - 1 \le t \le |D| - 1$. To calculate $F(t)$ and $\mathcal{C}(t)$, we need to compare at least $t - \top_E + 1$ times to get the minimum cost. Thus, the time complexity is $\mathcal{O}(|D|^2)$. Note that we can index all $\sum_{i=1}^{t} k_i$ in $\mathcal{O}(|D|)$ time, and it takes $\mathcal{O}(1)$ time to get any $\sum_{i=b}^{t} k_i$. In the algorithm, we need to store $\mathcal{C}(t)$, $F(t)$ and the value of $\sum_{i=1}^{t} k_i$ where $\top_E - 1 \le t \le |D|$. The space complexity is $\mathcal{O}(|D|^2)$ as $F(t)$ is a list of begin positions and in worst case each position b is a begin position which is stored in all $F(t)$ where $b \le t$.

Algorithm 4: `GreedyFragment`

Input: D: the document to fragment;
k_i: the number of elements in i-th inverted list;
Output: F: begin position list to fragment D;

```
1  begin
2      F = {1}, b = 1, H(b, |D|) = FragmentCost(1, |D|);
3      for b' = 2 to |D| − ⊤_E + 1 do
4          if H(b, b'+⊤_E−2) + H(b', |D|) < H(b, |D|) then
5              F = F ∪ b', b = b',
               H(b, |D|) = FragmentCost(b, |D|);
6      return F;
7  end
```

Fig. 19 Algorithm: Greedy document fragmentation

5.3 Greedy algorithm to document fragmentation

For a document D with $|D|$ tokens, the time and space complexities of the optimal document fragmentation algorithm are both $\mathcal{O}(|D|^2)$. Typically, the length $|D|$ of a document may be rather large and the optimal document fragmentation algorithm is very expensive. To address this issue, we propose an efficient greedy algorithm with $\mathcal{O}(|D|)$ time and space complexity to fragment the document.

The greedy algorithm scans the document token by token and tries to split the document into two fragments based on the current token. For each token, if the total cost of finding candidates from the two fragments is smaller than the cost of finding candidates from the entire fragment, the greedy algorithm adds the current token into begin position list, splits the document into two fragments and next greedily splits the next fragment. Formally, suppose the current token position is b' and the last begin position in F is b. Note that the begin position of the first fragment must be 1 and we initialize the begin position list F as $\{1\}$ and $b = 1$. Next, we calculate the time cost $\mathcal{H}(b, |D|)$ of finding candidates from $D[b \dots |D|]$ and the total time cost $\mathcal{H}(b, b' + \top_E - 2) + \mathcal{H}(b', |D|)$ of finding candidates from the two fragments $D[b \dots b' + \top_E - 2]$ and $D[b' \dots |D|]$ by utilizing the single-heap based method. We add b' into F if and only if $\mathcal{H}(b, b' + \top_E - 2) + \mathcal{H}(b', |D|) < \mathcal{H}(b, |D|)$. Finally, we get the list of begin positions F to fragment the document D.

The pseudo code of the greedy document fragmentation algorithm is shown in Fig. 19. The greedy algorithm first initializes the begin position list as $\{1\}$, sets $b = 1$ and calculates the cost of $\mathcal{H}(1, |D|)$(Line 2). Next, it scans the document D (Lines 3 to 5). For each position b', it calculates the values $\mathcal{H}(b, b' + \top_E - 2)$ and $\mathcal{H}(b', |D|)$. If position b' can decrease the cost of finding candidate pairs from $D[b \dots |D|]$, it adds b' into F, replaces b with b' and calculates value of $\mathcal{H}(b, |D|)$(Line 4). When the algorithm reaches the end of the document, it returns the list of begin positions F(Line 6).

We continue the last example. We first add 1 into F, set $b = 1$ and calculate $\mathcal{H}(1, |D| = 33) = 451$. Next, we traverse b' from 2 to $|D| - \top_E + 1 = 22$. For $b' = 2$, we have $\mathcal{H}(1, 12) + \mathcal{H}(2, 33) = 140 + 416 = 556 > 451$. Next, we set $b' = 3$. We continue this procedure and for $b' = 7$ we have $\mathcal{H}(1, 17) + \mathcal{H}(7, 33) = 140 + 304 = 444 < \mathcal{H}(1, 33) = 451$. We add $b' = 7$ into F, split the document into two fragments, set $b = 7$, calculate $\mathcal{H}(b = 7, 33) = 304$ and continue to fragment $D[7, 33]$. Finally, we have $F = \{1, 7\}$, and thus we fragment the document into two fragments $D[1, 17]$ and $D[7, 33]$ with cost 444.

Complexity We can easily deduce that the time complexity and space complexity of the greedy algorithm to fragment documents are both $\mathcal{O}(|D|)$.

5.4 Optimization techniques

The above two document fragmentation algorithms use $\top_E$ as an upper bound of lengths of valid substrings. We find that the size of an overlap region can be reduced to $\top_e$ and thus we can further decrease the cost of the heap-adjustment operations. In addition, they have to scan all the inverted lists in the overlap region twice to avoid false negatives. Using a tighter bound, we only need to scan some of the inverted lists in the overlap region twice.To achieve this goal, for each inverted list $\mathcal{IL}[i]$ with position i, we compute the maximum length $\top_e$ for each entity e in the inverted list and store the maximum one $\mathcal{IL}_i^{\top} = max\{\top_e | e \in \mathcal{IL}[i]\}$ on the list.

Consider a begin position b of a document fragment. Recall Sect. 5.1, we deduce its overlap region as $D[b, b + \top_E - 2]$, and each inverted list in the overlap region must be scanned twice by two adjacent heaps, to avoid false negatives. However, we find that for any token $token[i]$ in the overlap region with the position i, if $i - \mathcal{IL}_i^{\top} + 1 \geq b$, we do not need to add it into the former heap. This is because for any similar string pair $\langle s, e\rangle$, where s is a substring of D containing the token $token[i]$ and e is an entity in the inverted list $\mathcal{IL}[i]$ which also contains $token[i]$, all their common tokens can be found in the latter heap, as $|s| \leq \top_e \leq \mathcal{IL}_i^{\top} \leq i - b + 1$ (i.e. the begin position of $|s|$ is no smaller than b, which is the start position of the latter heap). Thus, we only need to scan the inverted list of $token[i]$ once in the latter heap and any similar pair $\langle s, e\rangle$ that shares the common token $token[i]$ can be found in the latter heap. We also have that for any inverted list $\mathcal{IL}[i]$ in the overlap region $D[b, b + \top_E - 2]$, if $i + \mathcal{IL}_i^{\top} - 1 \leq b + \top_E - 2$, we do not need to add it into the latter heap as for any similar pair $\langle s, e\rangle$ sharing the token $token[i]$, $i + |s| - 1 \leq i + \top_e - 1 \leq i + \top_p - 1 \leq b + \top_E - 2$ (i.e., the end position of s cannot be larger than $i + \top_E - 2$, which is the end position of the former heap).

Given start and end positions b, t, we only need to select some tokens to construct the heap based on the tighter bound. The algorithm is shown in Fig. 20. The algorithm uses L to store all the inverted lists in the fragment $D[b \ldots t]$ to construct the heap. First, it initializes L as $\{b + \top_E - 1, b + \top_E, \ldots, t - \top_E + 1\}$(Line 2). Next, it checks the inverted lists in the overlap region and only adds those inverted list $\mathcal{IL}[i]$ which satisfies $i + \mathcal{IL}_i^{\top} - 1 < b + \top_E - 2$ or $i - \mathcal{IL}_i^{\top} + 1 < t - \top_E + 2$ into L(Lines 3 to 5 and Lines 6 to 8). Note that here $b + \top_E - 2$ is the end position of previous fragment and $t - \top_E + 2$ is the begin position of next fragment. Finally, it returns the list L of inverted lists to construct the heap(Line 9).

Algorithm 5: `ListSelection`

Input: b, t: the begin and end positions of the fragment;
Output: L: the list of inverted lists selected to construct the heap;

```
1  begin
2      L = {b + ⊤_E − 1, b + ⊤_E, ..., t − ⊤_E + 1};
3      for i = b to b + ⊤_E − 2 do
4          if i + IL_i^⊤ − 1 > b + ⊤_E − 2 then
5              L = L ∪ i ;
6      for i = t − ⊤_E + 2 to t do
7          if i − IL_i^⊤ + 1 < t − ⊤_E + 2 then
8              L = L ∪ i ;
9      return L;
10 end
```

Fig. 20 Algorithm: select inverted lists to construct a heap

We still use the example in Sect. 5.1. Suppose, we split the document into two fragments $D[1, 18]$ and $D[8, 33]$. We need to build two heaps over the two fragments. For fragment $D[1, 18]$, we invoke the procedure `ListSelection(1, 18)` to select the inverted lists to construct the heap. We first initialize L as $\{1, 2, \ldots, 7\}$. Next, we traverse p from 8 to 18. For $i = 8$, we have $\mathcal{IL}_i^{\top} = 0$ and $i - \mathcal{IL}_i^{\top} = 8 \geq t - \top_E + 2 = 8$. Thus, we do not add the eighth inverted list into L. For $i = 18$, we have $\mathcal{IL}_i^{\top} = 11$ and $i - \mathcal{IL}_i^{\top} + 1 = 8 \geq t - \top_E + 2 = 8$. Thus, we also do not add this inverted list into L. Finally, we return $L = \{1, 2, \ldots, 7\}$ for fragment $D[1, 18]$.

Removing duplicate results There may be some duplicate result pairs between two adjacent heaps. To remove these duplicate results, we only need to use the begin position b of the latter heap as a bound of the start positions of substrings found by the former heap. That is for the former heap we only return those candidate substrings with start position smaller than b.

5.5 The hybrid algorithm

In this section, we propose a hybrid algorithm to efficiently find all answers. Similar to the single-heap-based algorithm, we first build an inverted index for the given dictionary E, and get the tokens and corresponding inverted lists of the doc-

Algorithm 6: The Hybrid Algorithm

```
   Input: Same as Algorithm 2
   Output: Same as Algorithm 2
1  begin
       /* Same as line 2 to 3 in Algorithm 2        */
2      Fragment the document D using GreedyFragment
       algorithm and get the list F of begin positions;
3      for two adjacent begin positions b and b' in F do
4          L = ListSelection(b, b' + ⊤_E − 2);
5          Construct a heap H on top of the lists in L;
           /* Same as line 5 to 17 in Algorithm 2   */
6  end
```

Fig. 21 The hybrid algorithm

ument D. Next, we split the document D into multiple fragments. For each fragment, we select a list of inverted lists and build a heap on top of them. Finally, we use the single-heap-based algorithm to find results in the fragment. Figure 21 gives the pseudo-code of the hybrid algorithm (Hybrid).

The Hybrid algorithm is similar to the Single algorithm. Instead of constructing a single heap on top of the entire document, the Hybrid algorithm invokes the procedure `GreedyFragment` to fragment the document D and gets a list F of begin positions of the fragments (Line 16). Then, for each fragment $D[b, b'+\top_E-2]$, where b and b' are two adjacent begin positions in F, the Hybrid algorithm selects a list L of inverted lists using procedure `ListSelection`(Line 3 to 4). Finally, the Hybrid method builds a heap over the inverted lists in L and utilizes the Single algorithm to find results in this fragment (Line 5).

Next, we give a running example to walk through the Hybrid algorithm. Consider the entities and document in Table 1. We first build an inverted index for the given dictionary as shown in Fig. 1. Then, we fragment the document using the `GreedyFragment` algorithm and get a list of begin positions $F = \{1, 18, 85, 87\}$. Next, for each of the three fragments $D[1, 28]$, $D[18, 95]$, $D[85, 97]$ and $D[87, 119]$, we invoke procedure `ListSelection` to select inverted lists and build a heap on top of the selected inverted lists. Finally, we utilize the Single algorithm to get the results.

6 Experiments

We have implemented our proposed techniques and conducted experiments to evaluate our methods. The objective of the experiments is to measure the performance, and we report results in this section.

Experimental setting We compared our algorithms with state-of-the-art methods NGPP [31] (the best method for edit distance) and ISH [4] (the best method for jaccard similarity and edit similarity). We downloaded the binary codes of NGPP [31] from "Similarity Joins" project website[7] and implemented ISH by ourselves. The algorithms were implemented in C++ and compiled using GCC 4.2.4 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core 2 Quad X5450 3.0 GHz processor and 4 GB memory.

Table 4 Datasets

Datasets & details	Cardinality	len	tokens
DBLP Dict, Author	100,000	21.1	2.77
DBLP Docs, Bibliography	10,000	123.3	16.99
PubMed Dict, Title	100,000	52.96	6.98
PubMed Docs, Record	10,000	235.8	33.6
WebPage Dict, Title	100,000	66.89	8.5
WebPage Docs, Page	1,000	8949	1268

Datasets We used three real datasets, DBLP,[8] PubMed,[9] and ACM WebPage.[10] DBLP is a computer science bibliography dataset. We selected 100,000 author names as entities and 10,000 bibliographies as documents. PubMed is a biomedical literature citation dataset. We selected 100,000 paper titles as entities and 10,000 publication records as documents. WebPage is a set of ACM web pages. We crawled 100,000 paper titles as a dictionary, and 1,000 web pages as documents (thousands of tokens). Table 4 illustrates the dataset statistics (where len denotes the average length and token denotes the average token number). We did not consider different attributes in the entities and documents, and each entity in the dictionary is just a string.

6.1 Multi-heap versus single heap

In this section, we compared the multi-heap-based method with the single-heap-based method (without using pruning techniques in Sect. 4). We tested the performance of the two methods for different similarity functions on the three datasets. Fig. 22 shows the experimental results.

We see that the single-heap-based method outperforms the multi-heap-based method by 1–2 orders of magnitude, and even 3 orders of magnitude in some cases. For example, on the DBLP dataset with edit-distance threshold $\tau = 3$, the multi-heap-based method took more than 10,000 s and the single-heap-based method took about 180 s. On the PubMed dataset with EDS similarity threshold $\delta = 0.9$, the multi-heap-based method took more than 14,000 s and the single-

[7] http://www.cse.unsw.edu.au/~weiw/project/simjoin.html.

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

[9] http://www.ncbi.nlm.nih.gov/pubmed.

[10] http://portal.acm.org.

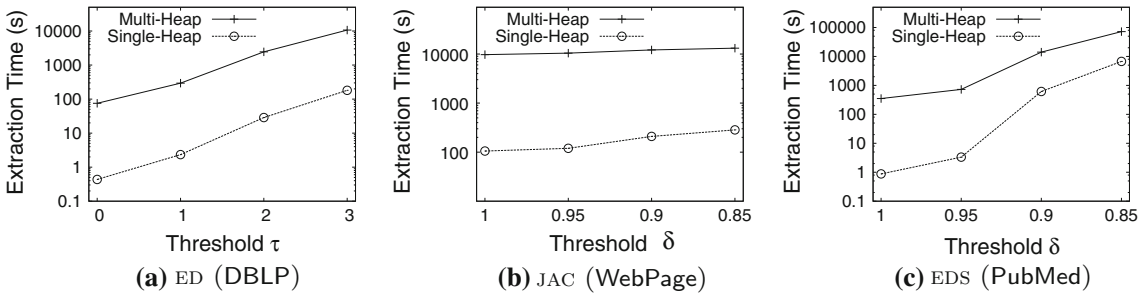


Fig. 22 Performance comparison of multi-heap-based methods and single-heap-based methods

heap-based method took only 600 s. There are two reasons that the single-heap-based method is better than the multi-heap-based method. Firstly, the multi-heap-based method scans each inverted list of the document many times and the single-heap-based method only scans them once. Secondly, the multi-heap-based method constructs larger numbers of heaps and does larger numbers of heap adjustment than the single-heap-based method. It is worth noting that the y-axis of Fig. 22 is logarithmic which leads to the single-heap-based method seems not asymptotically more efficient than the multi-heap-based method. In addition, the overall performance also depends on the number of candidates that require to be verified (especially when there are large numbers of candidates). With the decrease of similarity thresholds, the number of candidates increases and the verification cost also increases, and thus, the performance gap between two methods becomes smaller. As the single-heap-based method outperforms the multi-heap-based method, we focus on the single-heap-based method in the remainder of the experimental comparison.

6.2 Evaluating pruning techniques for the single-heap-based method

In this section, we tested the effectiveness of our pruning techniques. We first evaluated the number of candidates by applying different pruning techniques to our algorithm (lazy-count pruning, bucket-count pruning, and binary span and shift pruning). As batch-count pruning is a special case of binary span and shift pruning, we only show the results for binary span and shift pruning. In the paper, the number of candidates refers to the number of non-zero values in the occurrence arrays, which need to be verified. Figure 23 gives the results. In the figure, we tested edit distance on the DBLP dataset, jaccard similarity on WebPage dataset, and edit similarity on PubMed dataset. Note that in the figures, the results are in 10^x formate. For example, if there are 100 million candidates, the number in the figure is 8 ($10^8 = 100M$). In the paper, we tuned the parameters for different our method used parameter q for the gram-based method and reported the best performance, where $q = 16, 8, 5, 4, 3$ for $\tau = 0, 1, 2, 3, 4$ respectively for edit distance on DBLP and $q = 26, 11, 7, 5, 4$ for $\delta = 1, 0.95, 0.9, 0.85, 0.8$ edit similarity on PubMed.

We observe that our proposed pruning techniques can prune large numbers of candidates. For example, on the DBLP dataset, for $\tau = 3$, the method without any pruning techniques involved 11 billion candidates, and the lazy-count pruning reduced the number to 860 million. The bucket-count pruning further reduced the number to 600 million. The binary span and shift pruning had only 200 million candidates. On the WebPage dataset, for $\delta = 0.9$, the binary span and shift pruning reduced the number of candidates from 10 billion to 35. On the PubMed dataset, for $\delta = 0.85$, the binary span and shift pruning reduced the number of candidates from 180 billion to 120 million. The main reason is that we compute an upper bound of the overlap of an entity and a

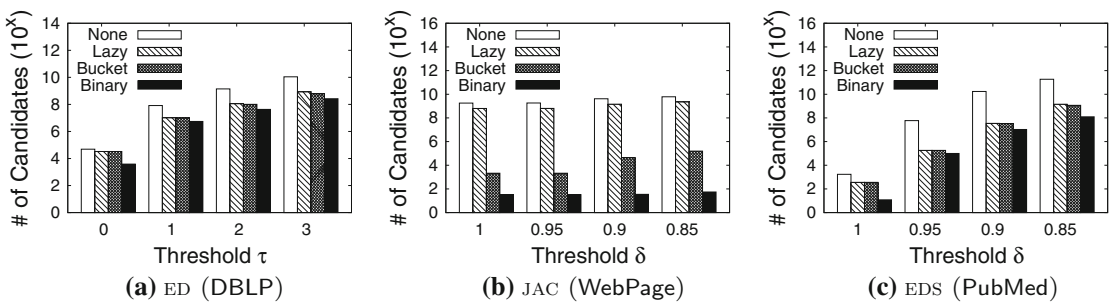


Fig. 23 Number of candidates with different pruning techniques

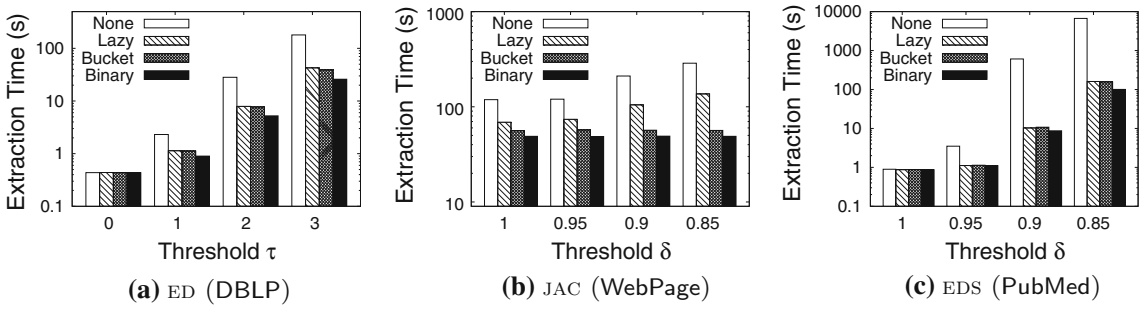


Fig. 24 Performance comparison with different pruning techniques

substring, and if the bound is smaller than the overlap threshold, we prune the substring. If for any substring, the bound of an entity is smaller than the threshold, we prune the entity. This confirms the superiority of our pruning techniques.

Next, we evaluated the performance benefit of the pruning techniques. Figure 24 shows the results. We observe that the pruning techniques can improve the performance. For instance, on the DBLP dataset, for $\tau = 3$, the elapsed time of the method without any pruning technique was 180 s, and the lazy-count pruning decreased the time to 43 s. The binary span and shift pruning reduced the time to 25 s. On the PubMed dataset, for $\delta = 0.9$, the pruning techniques can improve the time from 600 to 8 s. This shows that our pruning techniques can improve the performance.

6.3 Hybrid versus single heap

In this section, we evaluate the Hybrid method and compare it with the Single algorithm. We implemented three Hybrid methods, the `Hybrid-Optimal` method, the `Hybrid-Greedy` method, and the `Hybrid-Greedy+` method. The `Hybrid-Optimal` method uses the optimal fragment algorithm `OptFragment` to fragment documents and the other two methods utilizes the greedy algorithm `GreedyFragment` to fragment documents. In addition, the `Hybrid-Greedy+` method incorporates the `ListSelection` algorithms to avoid scanning the inverted lists in overlap regions twice while the other two methods do not use this feature.

As the hybrid method is only effective for long documents, for each dataset we concatenated 100 documents into a new document and conducted approximate entity extraction on the new documents to show the effectiveness of the Hybrid method. Figure 25 shows the filtering times of each method. We have the following observations. First, the three hybrid methods outperformed the single-heap-based method. For example, on the DBLP dataset with edit-distance threshold $\tau = 4$, the Single algorithm took 800 s, while the `Hybrid-Optimal` method, the `Hybrid-Greedy` method, and the `Hybrid-Greedy+` method took 585, 660 and 280 s respectively. This is because the hybrid methods can decrease the heap sizes and thus reduce the heap-adjustment cost. Second, the `Hybrid-Greedy+` method outperformed the `Hybrid-Optimal` method and the `Hybrid-Greedy` method, because the `ListSelection` algorithm can avoid scanning the inverted lists in the overlap region twice. Third, the `Hybrid-Greedy` method had good approximation quality and achieved comparable performance with the `Hybrid-Optimal` method.

We next compared the partition time of the four methods. Figure 26 shows the results. As the Single method has no partition stage, its partition time was 0. For the Hybrid methods, the `Hybrid-Optimal` method is much worse than the other two methods. This is because the `OptFragment` algorithm requires to find the optimal fragments, and the time complexity of the `OptFragment` algorithm is very high. The partition time of the `Hybrid-Greedy+` method is a little longer than the `Hybrid-Greedy` method, as the `ListSelection` algorithm requires to select the inverted lists in the overlap regions. For example, on the WebPage dataset with Jaccard-similarity threshold $\delta = 0.9$, the `Hybrid-Optimal` method took about 6,000 s while the `Hybrid-Greedy` method and the `Hybrid-Greedy+` method only took about 0.3 and 1 s.

Finally, we tested the overall extraction time, including the partition time, the filtering and verification time. Figure 27 shows the overall extraction time. Since the `Hybrid-Optimal` method is rather slow as the partition time is too long, we do not show its overall extraction time here. We can see from the figure that the `Hybrid-Greedy+` method has the best performance and the `Hybrid-Greedy` method beats the Single method. For example, on the PubMed dataset with edit-similarity threshold $\delta = 0.8$, the Single algorithm and the`Hybrid-Greedy` algorithm took 2,900 and 2,200 s to extraction entities respectively while the `Hybrid-Greedy+` algorithm only took 1,100 s. This is because the `Hybrid-Greedy+` method avoids scanning some inverted lists in the overlap region twice and the `Hybrid-Greedy` algorithm avoids some heap-adjustment operations.

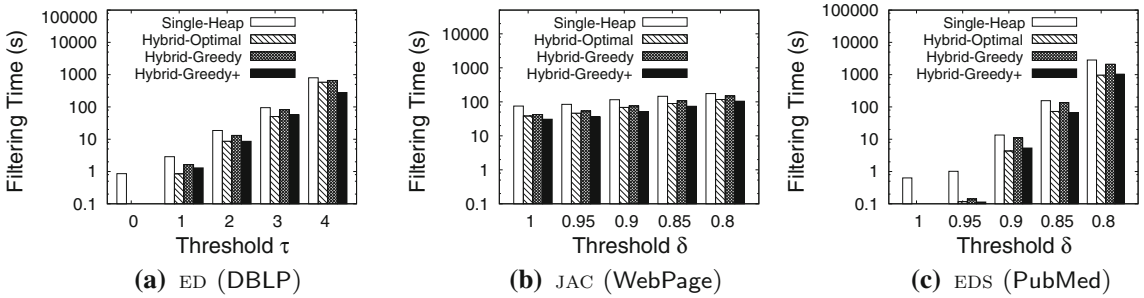


Fig. 25 Filtering time of the Single method and the Hybrid methods

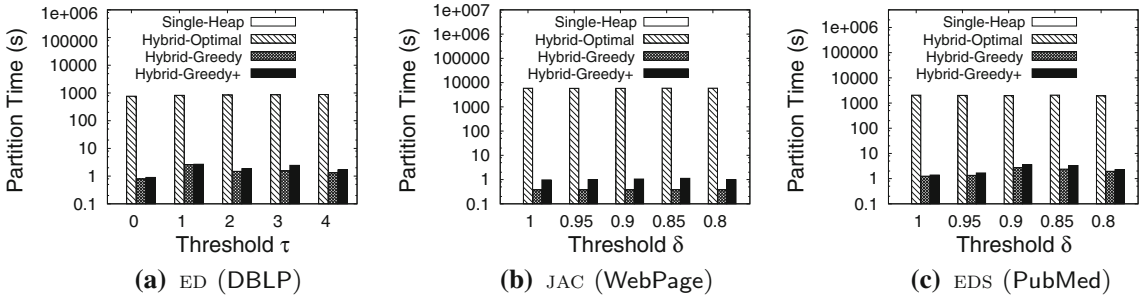


Fig. 26 Partition time of the Single method and the Hybrid methods

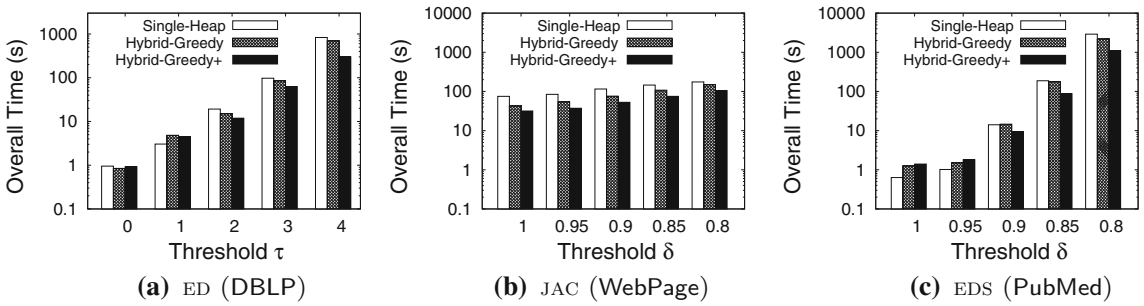


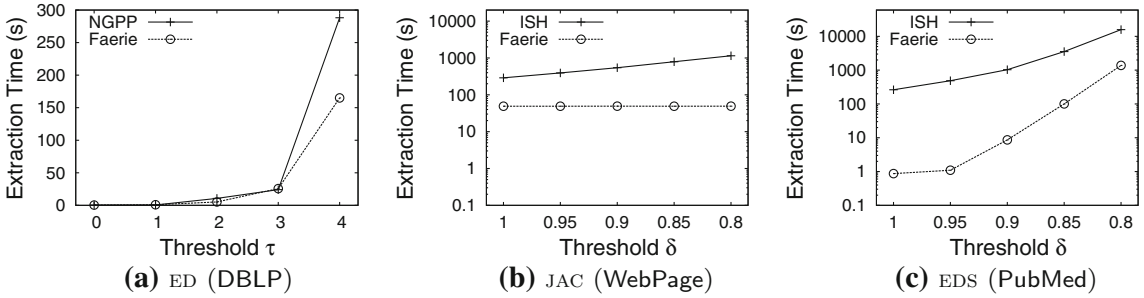
Fig. 27 Overall time of the Single method and the Hybrid methods

6.4 Comparison with state-of-the-art methods

In this section, we compared our algorithm with state-of-the-art methods NGPP [31] (which only supports edit distance) and ISH [4] (which supports edit similarity and jaccard similarity). As state-of-the-art methods are rather slow for long documents, we only show the performance of the Single method. In the figures, we used Faerie to denote our single-heap-based method.

We tuned the parameters of NGPP and ISH (e.g., prefix length of NGPP) to make them achieve the best performance. Figure 28 shows the results. We see that Faerie achieved the highest performance. Especially Faerie outperformed ISH by 1-2 orders of magnitude for edit similarity and jaccard similarity. For example, on the PubMed with edit-similarity threshold $\delta = 0.9$, the elapsed time of ISH was 1,000 s. Faerie reduced the time to 8 s. This is because Faerie used the shared computation across overlapped tokens. In addition, our pruning techniques can prune large numbers of unnecessary valid substrings and reduce the number of candidates. Although NGPP achieved high performance for smaller edit-distance thresholds, it is inefficient for larger edit-distance thresholds. The reason is that it needs to enumerate neighbors of entities and an entity has larger numbers of neighbors for larger thresholds. On jaccard similarity, as each entity has a smaller number of tokens (the average number is 8) and the thresholds T_l and $\top_e$ for different thresholds are nearly the same ($T_l = 8$ for $\delta = 1$ and $T_l = 10$ for $\delta = 0.8$), Faerie varied a little for different Jaccard-similarity thresholds.

In addition, we compared index sizes of difference algorithms. Note that NGPP had different index sizes for different edit-distance threshold τ, as NGPP uses τ to generate neighborhoods. The larger the edit-distance threshold, the larger indexes are involved for the neighborhoods of an entity, since an entity has larger numbers of neighbors for



(a) ED (DBLP) **(b)** JAC (WebPage) **(c)** EDS (PubMed)

Fig. 28 Comparison with existing methods

larger thresholds. On the DBLP dataset, for $\tau = 3$, NGPP consumed about 43 MB index size. The index size of Faerie was only 7 MB ($q = 4$). This result consists with that in [31]: the q-gram-based method has smaller index sizes than the neighborhood-based method (NGPP). On WebPage, ISH involved about 18 MB index size for jaccard-similarity threshold $\delta = 0.9$ (its parameter $k = 3$) and Faerie only used 4 MB.

6.5 Scalability with dictionary sizes

This section evaluates the scalability of our proposed method Faerie (we use the single-heap-based method) on various similarity functions. We varied the number of entities in the dictionary and identified similar pairs from the document collection in Table 4. Figure 29 shows the results for the five similarity functions. We observe that our method scaled well as the dictionary size increased. For example, on DBLP, for $\tau = 3$, Faerie took 6 s for 20,000 entities and 25 s for 100,000 entities. On WebPage, as each entity has a smaller number of tokens, Faerie varied a little for different thresholds. On PubMed, we evaluated edit similarity, dice similarity, cosine similarity, using q-grams. For edit similarity, when $\delta = 0.85$, Faerie took 9 s for 20,000 entities and 48 s for 100,000 entities.

In addition, we also evaluated the index sizes as the dictionary size increased. Table 5 shows the results. We see that the index sizes of our method were very small and scaled well as the number of entities increased.

6.6 Scalability with document length

In this section, we evaluated the scalability of our proposed method Faerie by varying the document length (we used our best algorithm `Hybrid-Greedy+`). We downloaded a DNA dataset from NCBI [11] to conduct this experiment. The dictionary contains 100,000 DNA segments with average length 108. We use 100 DNA sequences as documents. To evaluate the scalability of Faerie with document length, we fixed the size of dictionary and varied the average length of documents from 200,000 to 1,000,000. The results are shown in Fig. 30. We can see that Faerie scaled very well as the average length of document increases. For example, for edit distance threshold $\tau = 2$, the extraction time for the documents with average length 200,000, 400,000, 600,000, 800,000, and 1,000,000 were respectively 49, 98, 145, 197 and 246 s.

7 Related works

There have been some recent studies on approximate entity extraction [1,4,5,9,10,23,26,31]. Deng et al. [10] proposed a trie-based method for approximate entity extraction with edit distance constraint. Wang et al. [31] proposed neighborhood-generation-based methods for approximate entity extraction with edit-distance thresholds. They first partition strings into different partitions and guarantee that two strings are similar only if there exist two partitions of the two strings, which have an edit distance no larger than 1. Then, they generate neighborhoods of each partition by deleting one character from the partitions, and the edit distance between two partitions is not larger than 1 only if they have a common neighbor. How-

Table 5 Scalability of index sizes

# of Entities	20 k	40 k	60 k	80 k	100 k
(*a*) DBLP ($q = 5$)					
Inverted Index (MB)	1.6	3.22	4.9	6.5	8.2
Heap+Array (KB)	4.5	4.5	4.5	4.5	4.5
(*b*) WebPage					
Inverted Index (MB)	0.8	1.63	2.45	3.3	4.2
Heap+Array (KB)	38	38	38	38	38
(*c*) PubMed ($q = 7$)					
Inverted Index (MB)	4.5	9.2	14.1	18.3	22.8
Heap+Array (KB)	7.2	7.2	7.2	7.2	7.2

[11] http://www.ncbi.nlm.nih.gov/genome.

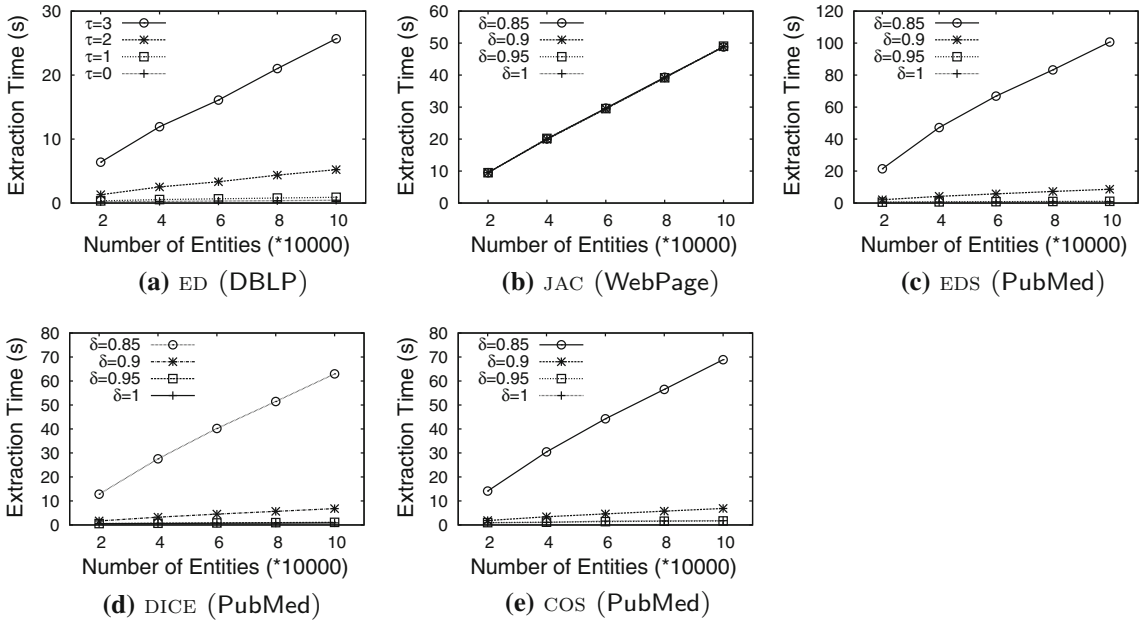


Fig. 29 Scalability of performance for various similarity functions on different datasets

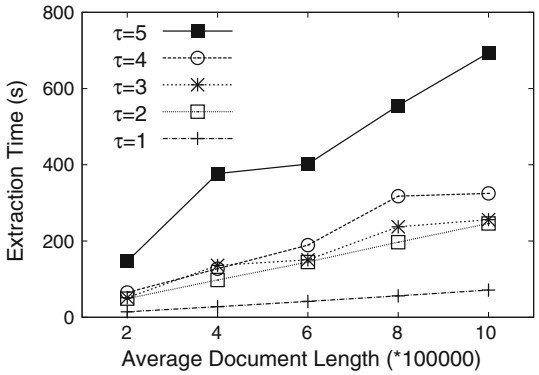


Fig. 30 Scalability with document length

ever, this method cannot support the token-based similarity. Chakrabarti et al. [4] proposed an inverted signature-based hash-table for membership checking. They first selected top-weighted tokens as signatures and encoded the dictionary as a 0-1 matrix. Then, they built a matrix for the document and used the matrix to find candidates. Lu et al. [26] proposed signature-based inverted lists to improve [4] by using a tighter threshold. However, this method cannot support edit distance. In addition, Agrawal et al. [1] proposed to use inverted lists for ad hoc entity extraction. Chandel et al. [5] studied the problem of batch top-k search for dictionary-based exact entity recognition. Chaudhuri et al. [9] proposed to expand a reference dictionary of entities by mining large document collections.

Different from existing works, we proposed a unified framework to support various similarity functions [23]. Compared with our previous work [23], the significant additions in this extended manuscript are summarized as follows.

– We proposed a new hybrid algorithm by combining the multi-heap-based method and the single-heap-based method, which can reduce the filtering cost while keeping the same pruning power. Sect. 5 was newly added.
– We conducted new experiments to evaluate the effectiveness of our hybrid method and showed its performance gain. Sect. 6.3 was newly added.
– We formally proved all the lemmas and theorems in the paper.

Similarity search and join Many studies have been proposed to address the approximate-string-search problem [2,4,6,8,11,14,15,17,18,21,22,34] which finds similar strings of a query string from a string collection, and the similarity-join problem [2,3,7,12,13,24,25,27–30,32,33] which finds similar string pairs from two string collections. Although we can extend them to solve our problem, they are very inefficient, since they need to enumerate all valid substrings in the document and cannot use the shared computation across overlaps of substrings. Existing works (NGPP [31] and ISH [4]) have proved that the extraction-based methods outperform similarity-join-based methods for the approximate entity-extraction problem, and thus, we only compare with state-of-the-art methods NGPP [31] and ISH [4]. In addition, there have been many studies on estimating selectivity for approximate string queries [16,19,20].

8 Conclusion

In this paper, we have studied the problem of approximate dictionary-based entity extraction. We proposed a unified framework to support various similarity functions. We devised heap-based filtering algorithms to efficiently extract similar entities from a document. We developed a single-heap-based algorithm which can utilize the shared computation across overlaps of substrings by constructing a single heap on top of inverted lists of tokens in the document and scanning every inverted list only once. We proposed several pruning techniques to prune large numbers of unnecessary candidate pairs. We devised binary-search-based techniques to improve the performance. We proposed a hybrid-based algorithm to combine the single-heap-based algorithm and the multi-heap-based algorithm to further enhance the performance. We have implemented our proposed techniques and tested our method on several real datasets. The experimental results show that our method achieves high performance and outperforms state-of-the-art studies significantly.

Acknowledgments This work was partly supported by the National Natural Science Foundation of China under Grant No. 61272090 and 61373024, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, Beijing Higher Education Young Elite Teacher Project under Grant No. YETP0105, a project of Tsinghua University under Grant No. 20111081073, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, the "NExT Research Center" funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490, and the FDCT/106/2012/A3.

References

1. Agrawal, S., Chakrabarti, K., Chaudhuri, S., Ganti, V.: Scalable ad-hoc entity extraction from text collections. PVLDB **1**(1), 945–957 (2008)
2. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact setsimilarity joins. In: VLDB, pp. 918–929 (2006)
3. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In WWW, pp. 131–140 (2007)
4. Chakrabarti, K., Chaudhuri, S., Ganti, V., Xin, D.: An efficient filter for approximate membership checking. In: SIGMOD Conference, pp. 805–818 (2008)
5. Chandel, A., Nagesh, P. C., Sarawagi, S.: Efficient batch top-k search for dictionary-based entity recognition. In: ICDE, pp. 28 (2006)
6. Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R.: Robust and efficient fuzzy match for online data cleaning. In: SIGMOD Conference, pp. 313–324 (2003)
7. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In ICDE, pp. 5–16 (2006)
8. Chaudhuri, S., Ganti, V., Motwani, R.: Robust identification of fuzzy duplicates. In: ICDE, pp. 865–876 (2005)
9. Chaudhuri, S., Ganti, V., Xin, D.: Mining document collections to facilitate accurate approximate entity matching. PVLDB **2**(1), 395–406 (2009)
10. Deng, D., Li, G., Feng, J.: An efficient trie-based method for approximate entity extraction with editdistance constraints. In: ICDE, pp. 762–773 (2012)
11. Deng, D., Li, G., Feng, J., Li, W.-S.: Top-k string similarity search with edit-distance constraints. In: ICDE, pp. 925–936 (2013)
12. Feng, J., Wang, J., Li, G.: Trie-join: a trie-based method for efficient string similarity joins. VLDB J. **21**(4), 437–461 (2012)
13. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
14. Hadjieleftheriou, M., Chandel, A., Koudas, N., Srivastava, D.: Fast indexes and algorithms for set similarity selection queries. In: ICDE, pp. 267–276 (2008)
15. Hadjieleftheriou, M., Koudas, N., Srivastava, D.: Incremental maintenance of length normalized indexes for approximate string matching. In: SIGMOD Conference, pp. 429–440 (2009)
16. Hadjieleftheriou, M., Yu, X., Koudas, N., Srivastava, D.: Hashed samples: selectivity estimators for set similarity selection queries. PVLDB **1**(1), 201–212 (2008)
17. Kim, M.-S., Whang, K.-Y., Lee, J.-G., Lee, M.-J.: ngram/ 2l: a space and time efficient two-level n-gram inverted index structure. In: VLDB, pp. 325–336 (2005)
18. Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing join and selection queries. In: VLDB, pp. 199–210 (2006)
19. Lee, H., Ng, R.T., Shim, K.: Extending q-grams to estimate selectivity of string matching with low edit distance. In: VLDB, pp. 195–206 (2007)
20. Lee, H., Ng, R.T., Shim, K.: Power-law based estimation of set similarity join size. PVLDB **2**(1), 658–669 (2009)
21. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
22. Li, C., Wang, B., Yang, X.: Vgram: Improving performance of approximate queries on string collections using variable-length grams. In: VLDB, pp. 303–314 (2007)
23. Li, G., Deng, D., Feng, J.: Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In: SIGMOD Conference, pp. 529–540 (2011)
24. Li, G., Deng, D., Feng, J.: A partition-based method for string similarity joins with edit-distance constraints. ACM Trans. Database Syst. **38**(2), 9 (2013)
25. Li, G., Deng, D., Wang, J., Feng, J.: Pass-join: a partition-based method for similarity joins. PVLDB **5**(3), 253–264 (2011)
26. Lu, J., Han, J., Meng, X.: Efficient algorithms for approximate member extraction using signature-based inverted lists. In: CIKM, pp. 315–324 (2009)
27. Sarawagi, S., Kirpal, A.: Efficient set joins on similarity predicates. In: SIGMOD Conference, pp. 743–754 (2004)
28. Wang, J., Li, G., Feng, J.: Trie-join: efficient trie-based string similarity joins with edit-distance constraints. PVLDB **3**(1), 1219–1230 (2010)
29. Wang, J., Li, G., Feng, J.: Fast-join: an efficient method for fuzzy token matching based string similarity join. In: ICDE, pp. 458–469 (2011)
30. Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: SIGMOD conference, pp. 85–96 (2012)
31. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: SIGMOD Conference (2009)
32. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
33. Xiao, C., Wang, W., Lin, X., Shang, H.: Top-k set similarity joins. In: ICDE, pp. 916–927 (2009)
34. Xiao, C., Wang, W., Lin, X. and Yu, J.X.: Efficient similarity joins for near duplicate detection. In: WWW (2008)