Faerie: Efficient Filtering Algorithms for Approximate Dictionary-based Entity Extraction

Guoliang Li Dong Deng Jianhua Feng
Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.
liguoliang@tsinghua.edu.cn; buaasoftdavid@gmail.com; fengjh@tsinghua.edu.cn

ABSTRACT

Dictionary-based entity extraction identifies predefined entities (e.g., person names or locations) from a document. A recent trend for improving extraction recall is to support *approximate* entity extraction, which finds all substrings in the document 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 programming efforts, hardware requirements, and the manpower. In addition, many substrings in the document have overlaps, and we have an opportunity to utilize the shared computation across the overlaps to avoid unnecessary redundant computation. In this paper, 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 efficient filtering algorithms to utilize the shared computation and develop effective pruning techniques to improve the performance. The experimental results show that our method achieves high performance and outperforms state-of-the-art studies.

Categories and Subject Descriptors:H.2 [Database Management]: Database applications; H.3.3 [Information Search and Retrieval]

General Terms: Algorithms, Experimentation, Performance

Keywords: Approximate Entity Extraction; Unified Framework; Filtering Algorithms; Pruning Techniques

1. INTRODUCTION

Dictionary-based entity extraction identifies all the substrings from a document that match the predefined entities in a given dictionary. For example, consider a document "*An Efficient Filter for Approximate Membership Checking. Venkaee shga Kamunshik kabarati, Dong Xin, Surauijt Chadhuri SIGMOD*", and a dictionary with two entities "`Surajit Chaudhuri`" and "`Dong Xin`". Dictionary-based entity extraction finds the predefined entity "`Dong Xin`" from the document. This problem has many real applications in the fields of information retrieval, molecular biology, bioinformatics, and natural language processing.

However, the document may contain typographical or orthographical errors and the same entity may have different representations [22]. For example, the substring "*Surauijt Chadhuri*" in the above document has typographical errors. The traditional (exact) entity extraction cannot find this substring from the document, since the substring does not *exactly* match the predefined entity "`Surajit Chaudhuri`". 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, we study the problem of approximate dictionary-based entity extraction, which, given a dictionary of entities and a document, finds 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 entity extraction can find the substring "*Surauijt Chadhuri*" which is similar to entity "`Surajit Chaudhuri`".

Although there have been some recent studies on approximate entity extraction [22, 4], 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 different similarity/dissimilarity functions, since the unified method can reduce not only the programming efforts, but also the hardware requirements and the manpower needed to maintain the codes for different functions.

In addition, we have an observation that many substrings in the document have overlaps. For example, consider the above document, many substrings (e.g., "*Surauijt Chadhuri*", "*urauijt ChadhuriSIG*", "*rauijt ChadhuriSIGMOD*") have overlaps (e.g., "*Chadhuri*"). We have an opportunity to utilize this feature to avoid the redundant computation across overlaps of different substrings. For example, we can share the computation on "*Chadhuri*" for different substrings.

To remedy the above problems, we propose a unified framework to support various similarity/dissimilarity functions.

[*]This work is partly supported by the National Natural Science Foundation of China under Grant No. 61003004, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, and National S&T Major Project of China under Grant No. 2011ZX01042-001-002.

SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

To avoid redundant computation across overlaps, we develop efficient filtering algorithms for approximate dictionary-based entity extraction, called “Faerie”. To summarize, we 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 efficient filtering algorithms, which can utilize the shared computation across the overlaps of multiple substrings of the document.
- We develop effective pruning techniques and devise efficient algorithms to improve the performance.
- We have implemented our method, and the experimental results show that our method achieves high performance and outperforms state-of-the-art approaches.

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

2. A UNIFIED FRAMEWORK

We first formulate the problem of approximate (dictionary-based) entity extraction (Section 2.1), and then introduce a unified method to support various similarity/dissimilarity functions (Section 2.2). Finally we introduce a concept of “valid substrings” to prune unnecessary substrings (Section 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 $e_i \in E$.*

This paper focuses on token-based similarity, character-based similarity, and character-based dissimilarity.

Token-based Similarity: It includes Jaccard Similarity, Cosine Similarity, and Dice Similarity. The token-based similarity takes a string as a set of tokens. Let JAC, COS, and DICE respectively denote the jaccard similarity, cosine similarity, and dice similarity. 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, JAC(“`sigmod 2011 conference`”, “`sigmod 2011`”) $= \frac{2}{3}$, COS(“`sigmod 2011 conference`”, “`sigmod 2011`”) $= \frac{2}{\sqrt{6}}$, and DICE(“`sigmod 2011 conference`”, “`sigmod 2011`”) $= \frac{4}{5}$.

Charater-based Dissimilarity: It includes Edit Distance. The charater-based dissimilarity takes a string as a sequence of characters. The edit distance between two 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 string r to string s. For example, ED(“`surajit`”, “`surauijt`”) $= 2$.

Table 1: A dictionary of entities and a document.

(a) Dictionary E

ID	Entity e	len(e)	$\lvert e \rvert$ (# of q-grams with $q = 2$)
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.

Charater-based Similarity: It includes Edit Similarity. The edit similarity between two strings r and s is defined as $\text{EDS}(r, s) = 1 - \frac{\text{ED}(r,s)}{\max\big(\text{len}(r), \text{len}(s)\big)}$, where len($s$) denotes the length of s. For instance, 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*” in the document misses a space between “*chadhuri*” and “*sigmod*” (a typographical error), our method still can find “*chadhuri*” (similar to entity “`chaudhuri`”).

It has been shown that approximate entity extraction can improve recall [22]. 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 take the selection of such thresholds as a future work.

2.2 A Unified Method

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 $|e \cap s|$ must be not 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* $\text{EDS}(e,s) \geq \delta$,
  $|e \cap s| \geq \lceil \max(|e|,|s|) - \big(\max(|e|,|s|) + q - 1\big) * (1-\delta) * q \rceil$.

In this way, we can transform various similarities/dissimilarities to the overlap similarity, and develop a unified filtering condition: if $|e \cap s| < T$, we prune the pair $\langle s, e \rangle$.

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 [22, 4].

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

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

We employ a filter-and-verify framework to address the problem of approximate entity extraction. 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/disimilarity. In this paper, we focus on the filter step.

3. HEAP-BASED FILTERING ALGORITHMS

In this section, we propose heap-based filtering algorithms to utilize the shared computation across overlaps. We first introduce an inverted index structure (Section 3.1), and then devise a multi-heap-based algorithm (Section 3.2) and a single-heap-based algorithm (Section 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 use the inverted index structure. 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, Figure 1 gives the inverted list for entities in Table 1 using q-grams with $q = 2$.

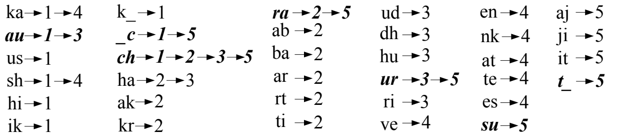


Figure 1: Inverted indexes for entities in Table 1.

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.

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 Figure 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.

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

[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.

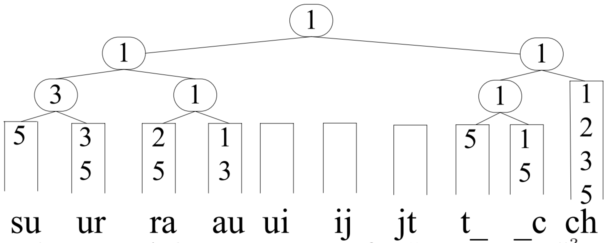


Figure 2: A heap structure for "surauijt ch"[3].

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 this section, we propose a multi-heap based method to count the occurrence numbers.

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 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 (Figure 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.

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 2(a)).

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 2(b)). 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 (Figure 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 2(b)).

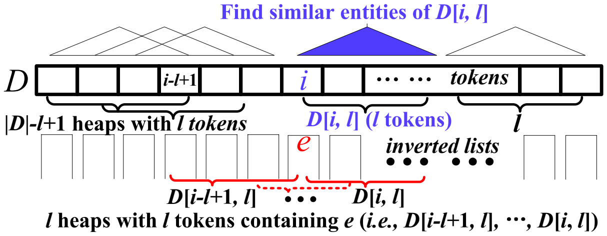

Find similar entities of D[i, l]
D
i-l+1
i
··· ···
tokens
|D|-l+1 heaps with l tokens
D[i, l] (l tokens)
l
e
inverted lists
D[i-l+1, l]
D[i, l]
l heaps with l tokens containing e (i.e., D[i-l+1, l], ···, D[i, l])

Figure 3: A multi-heap structure.

Table 2: Complexity of multi-heap based methods.

(a) Space Complexity

Maximum Heap	$\mathcal{O}(\top_E)$

(b) Time Complexity

Heap Construction	$\mathcal{O}\Big(\sum_{l=\perp_E}^{\top_E}(\lvert D\rvert - l + 1) * l\Big)$
Heap Adjustment	$\mathcal{O}\Big(\sum_{l=\perp_E}^{\top_E} log(l) * l * N\Big)$

The multi-heap based method needs to access inverted lists multiple times and does large numbers of heap-adjustment operations. To address this issue, we propose a new method which accesses every inverted list only once in Section 3.3.

3.3 Single-Heap based Method

In this section, we propose a single-heap-based method. 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. $V[i][l]$ 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 (Figure 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 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 array $V[1\cdots|D|][\perp_e\cdots\top_e]$.

Next, we give 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 Figure 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

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

[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.

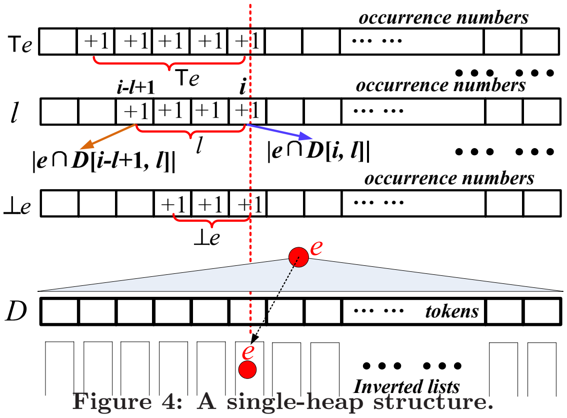


Figure 4: A single-heap structure.

Table 3: Complexity of single-heap based methods.

(a) Space Complexity

Single Heap	$\mathcal{O}(\lvert D\rvert)$
Counting Occurrence Numbers	$\mathcal{O}(\lvert D\rvert - \perp_E + 1)$

(b) Time Complexity

Heap Construction	$\mathcal{O}(\lvert D\rvert)$
Heap Adjustment	$\mathcal{O}\big(log(\lvert D\rvert) * N\big)$
Counting Occurrence Numbers	$\mathcal{O}\big(N * \max\{\sum_{l=\perp_e}^{\top_e} l \mid e \in E\}\big)$

substrings $D[1,l]$ for $\perp_E \le l \le \top_E$, i.e., $D[1,6],\dots,D[1,12]$. We increase the values of $V[1][6],\dots,V[1][12]$ 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 $\perp_E \le l \le \top_E$. 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 3(a)). 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 3(a)).

The time complexity of heap construction is $\mathcal{O}(|D|)$ (Table 3(b)). 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 entities 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 (Figure 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 3(b)).

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 (Section 6).

4. IMPROVING THE SINGLE-HEAP-BASED METHOD

In this section, we propose effective techniques to improve

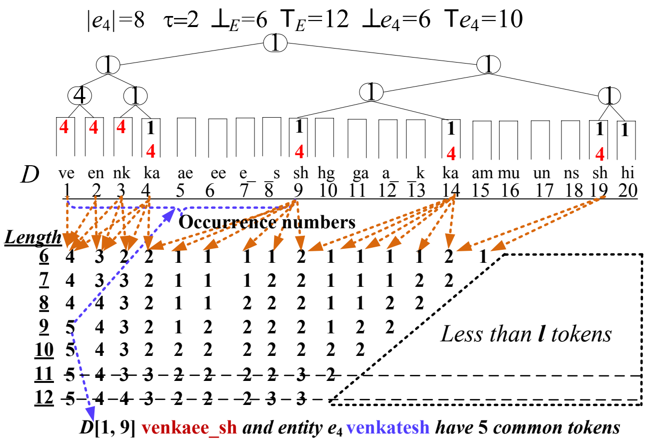


Figure 5: An example for the single-heap-based method on a document "`venkaee shga kamunshi`".

the performance of the single-heap-based method. Li et al. [18] have studied efficient heap-merge algorithms to find similar entities for a *single* substring. 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.

4.1 Pruning Techniques

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

Lazy-Count Pruning: For each top entity on the heap, we will not count its occurrence numbers for every valid string immediately. Instead, we count the numbers in a lazy manner. We first count its occurrence number in the heap, and if the number is small enough, we can prune the entity.

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 $\top_e$ (Section 3.3). For example, in Figure 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 (Section 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 $T_l \le T$. If $|P_e| < T_l \le T$, e cannot be similar to any substring, and thus we can prune e (Section 2.2). For instance, in Figure 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.*

Bucket-Count Pruning: 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 P_e, p_i and p_{i+1}, any substring containing both the two tokens ($\textsc{token}[p_i]$ and $\textsc{token}[p_{i+1}]$) 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 $\bot_e$ and $\top_e$ (Section 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 Figure 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 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 ($\textsc{token}[p_i]$ and $\textsc{token}[p_j]$) cannot be similar to entity e. Next we will introduce how to use this property to do further pruning.

Batch-Count Pruning: We have an observation 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 test 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 $token[p_i], token[p_{i+1}], \cdots, token[p_j]$ (Figure 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 $\bot_e \leq |D[p_i \cdots p_j]| \leq \top_e$. 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 $T_l \leq |P_e[i \cdots j]| \leq \top_e$.

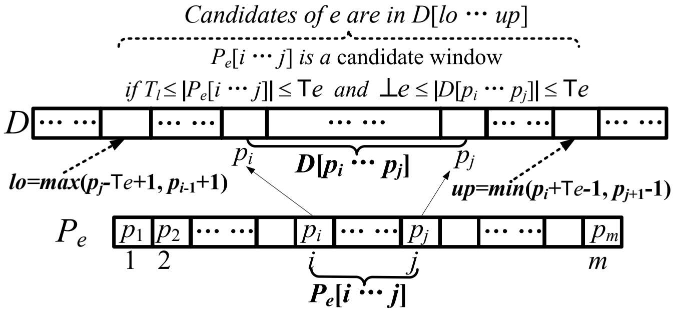

Candidates of e are in D[lo ⋯ up]
$P_e[i \cdots j]$ is a candidate window
if $T_l \leq |P_e[i \cdots j]| \leq \top e$ and $\bot e \leq |D[p_i \cdots p_j]| \leq \top e$
D
p_i
$D[p_i \cdots p_j]$
p_j
$lo=max(p_j-\top e+1, p_{i-1}+1)$
$up=min(p_i+\top e-1, p_{j+1}-1)$
P_e
p_1
p_2
p_i
p_j
p_m
1
2
i
j
m
$P_e[i \cdots j]$
$P_e[i \cdots j]$ is a valid window if $T_l \leq |P_e[i \cdots j]| \leq \top e$

Figure 6: Candidate window and Valid window.

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 $\bot_e \leq |D[p_i \cdots p_j]| \leq \top_e$, we find *candidates* of entity e (a substring s is a candidate of e if $|e \cap s| \geq T$ and $\bot_e \leq |s| \leq \top_e$). 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 $\bot_e \leq |D[p_i \cdots p_j]| \leq \top_e$.*

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 (Figure 6). 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 $\tau = 2$. $|e_4| = 8$, $T_l = |e_4| - \tau * q = 4$. $\bot_{e_4} = |e_4| - \tau = 6$ 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 $p_9 - p_6 + 1 < \bot_{e_4}$.

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, $\bot_e \leq |D[p_i \cdots p_j]| \leq \frac{\min(|e|, |P_e[i \cdots j]|)}{\delta}$; for dice similarity,

[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 $|e|$.

$\bot_e \le |D[p_i \cdots p_j]| \le \min(|e|, |P_e[i, j]|) * \frac{2-\delta}{\delta}$; for cosine similarity, $\bot_e \le |D[p_i \cdots p_j]| \le \frac{\min(|e|, |P_e[i \cdots j]|)}{\delta^2}$.

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 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 $\bot_e \le |s| = p_{end} - p_{start} + 1 \le \top_e$ 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 $\bot_e \le p_j - p_i + 1 \le p_j - p_{i-1} + 1 \le \top_e$ 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. The correctness and completeness of our method is formalized as below.

THEOREM 1 (CORRECTNESS AND COMPLEXNESS). *Our method finds the candidate pairs completely and correctly.*

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 (Figure 7).

- span: 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 $p_k - p_i + 1 \ge \bot_e$, $P_e[i \cdots k]$ 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: 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 $\tau{=}2$. $|e_4| = 8$, $T_l = |e_4| - \tau * q = 4$, $\bot_{e_4} = |e_4| - \tau = 6$, $\top_{e_4} = |e_4| + \tau = 10$. 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 $p_9 {-} p_6 {+} 1 {=} 61 {-} 58 {+} 1 {<} \bot_{e_4}$, $P_{e_4}[6 \cdots 9]$ is not a candidate window. We do a span operation. As $p_{10} {-} p_6 {+} 1 {=} 9 {\le} \top_{e_4}$ 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 $P_{e_4}[7 \cdots 10]$ (Figure 8).

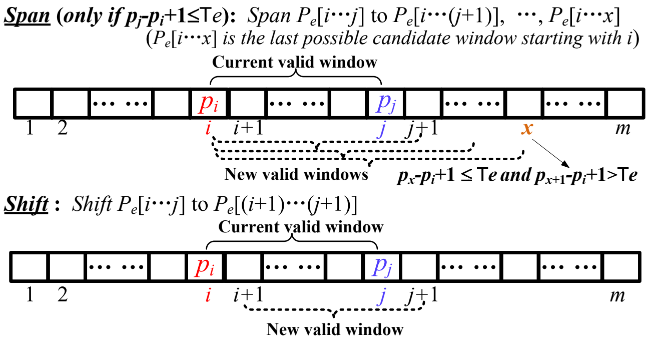
Given a valid window $P_e[i \cdots j]$, if $p_j - p_i + 1 > \top_e$, the shift operations can prune the valid windows starting with i, e.g., $P_e[i \cdots k]$ for $j < k \le i + \top_e - 1$. However this method still scans large numbers of candidate windows. To further improve performance, we propose 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 $p_{mid+j-i} - p_{mid} + 1 \le \top_e$ and for any $i \le mid' < mid$, $p_{mid'+j-i} - p_{mid'} + 1 > \top_e$. Similarly, if $p_j - p_i + 1 \le \top_e$, we will not iteratively span it to $P_e[i \cdots (j+1)], P_e[i \cdots (j+2)], \ldots, P_e[i \cdots x]$. 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 $p_x - p_i + 1 \le \top_e$ and for any $x' > x$, $p_{x'} - p_i + 1 > \top_e$.

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.

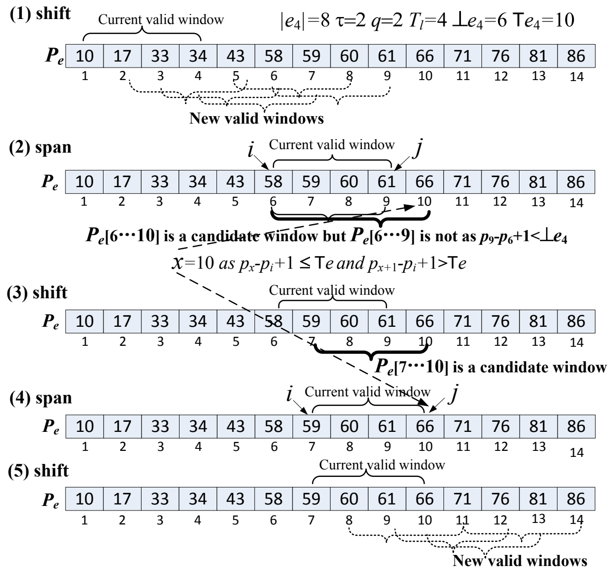
For the span operation, obviously $F(x) = p_x - p_i + 1$ is monotonic as $F(x+1) - F(x) = p_{x+1} - p_x > 0$. Next we give the lower bound and upper bound of the search range. Obviously $x \ge j$. In addition, as $p_i + j \le p_{i+j}$, we have $p_x \le p_i + \top_e - 1 \le p_{i+\top_e-1}$ and $x \le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



Span (only if p_j-p_i+1≤⊤e): Span $P_e[i\cdots j]$ to $P_e[i\cdots(j+1)]$, ⋯, $P_e[i\cdots x]$
($P_e[i\cdots x]$ is the last possible candidate window starting with i)
Current valid window
p_i
p_j
1
2
i
i+1
j
j+1
x
m
New valid windows
p_x-p_i+1 ≤ ⊤e and p_{x+1}-p_i+1>⊤e
Shift : Shift $P_e[i\cdots j]$ to $P_e[(i+1)\cdots(j+1)]$
Current valid window
New valid window

Figure 7: span and shift operations.



(1) shift
Current valid window
$|e_4|$=8 τ=2 q=2 T_l=4 ⊥e_4=6 ⊤e_4=10
P_e
10 17 33 34 43 58 59 60 61 66 71 76 81 86
1 2 3 4 5 6 7 8 9 10 11 12 13 14
New valid windows
(2) span
i
Current valid window
j
P_e[6⋯10] is a candidate window but P_e[6⋯9] is not as p_9-p_6+1<⊥e_4
x=10 as p_x-p_i+1 ≤ ⊤e and p_{x+1}-p_i+1>⊤e
(3) shift
P_e[7⋯10] is a candidate window
(4) span
(5) shift
New valid windows

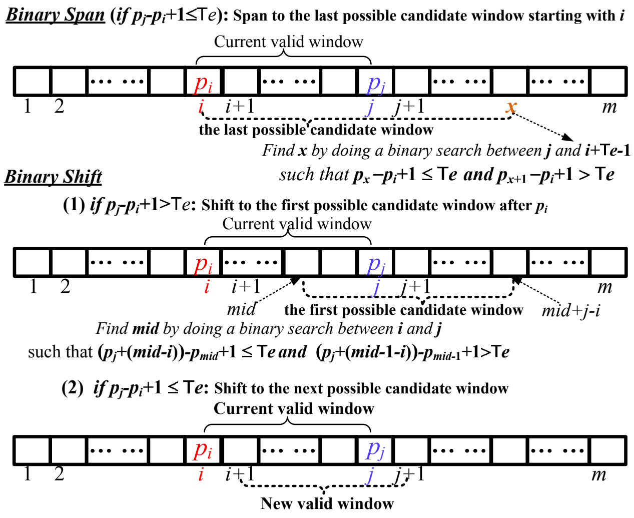
Figure 8: An example for span and shift operations.

Thus given a valid window $P_e[i\cdots j]$, we use binary span and shift operations to find candidate windows (Figure 9).

- Binary span: 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: 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 $P_e[(i+1)\cdots(j+1)]$.

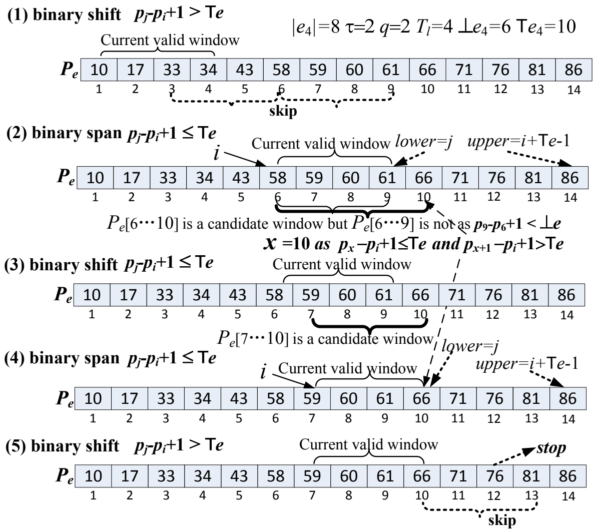
Given a valid window $P_e[i\cdots j]$, the binary shift can skip unnecessary valid windows (non-candidate windows), such as $P_e[(i+1)\cdots(j+1)]$, …, $P_e[(mid-1)\cdots(mid-1+j-i)]$, as proved in Lemma 4. For example, consider the position list in Figure 10. Suppose $\tau = 2$. $T_l = 4, |e_4| = 8, \top_{e_4} = 10$. Consider the first valid window $P_{e_4}[1\cdots4]$. The shift operation shifts it to $P_{e_4}[2\cdots5]$, $P_{e_4}[3\cdots6]$, $\cdots$, $P_{e_4}[6\cdots9]$, and checks whether they are candidate windows. The binary shift operation can directly shift it to $P_{e_4}[3\cdots6]$ and then to $P_{e_4}[6\cdots9]$. 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 $\big(p_j+(mid-1)-i\big)-p_{mid-1}+1 > \top_e$, $P_e[(i+1)\cdots(j+1)]$, …, $P_e[(mid-1)\cdots\big((mid-1)+j-i\big)]$ are not candidate windows.*



Binary Span (if p_j-p_i+1≤⊤e): Span to the last possible candidate window starting with i
Current valid window
p_i
p_j
1
2
i
i+1
j
j+1
x
m
the last possible candidate window
Find x by doing a binary search between j and i+⊤e-1
such that p_x−p_i+1 ≤ ⊤e and p_{x+1}−p_i+1 > ⊤e
Binary Shift
(1) if p_j-p_i+1>⊤e: Shift to the first possible candidate window after p_i
mid
the first possible candidate window
mid+j-i
Find mid by doing a binary search between i and j
such that (p_j+(mid-i))-p_{mid}+1 ≤ ⊤e and (p_j+(mid-1-i))-p_{mid-1}+1>⊤e
(2) if p_j-p_i+1 ≤ ⊤e: Shift to the next possible candidate window
New valid window

Figure 9: span and shift in a binary-search way.



(1) binary shift p_j-p_i+1 > ⊤e
$|e_4|$=8 τ=2 q=2 T_l=4 ⊥e_4=6 ⊤e_4=10
Current valid window
P_e
10 17 33 34 43 58 59 60 61 66 71 76 81 86
1 2 3 4 5 6 7 8 9 10 11 12 13 14
skip
(2) binary span p_j-p_i+1 ≤ ⊤e
lower=j
upper=i+⊤e-1
P_e[6⋯10] is a candidate window but P_e[6⋯9] is not as p_9-p_6+1 < ⊥e
x =10 as p_x−p_i+1≤⊤e and p_{x+1}−p_i+1>⊤e
(3) binary shift p_j-p_i+1 ≤ ⊤e
P_e[7⋯10] is a candidate window
(4) binary span p_j-p_i+1 ≤ ⊤e
(5) binary shift p_j-p_i+1 > ⊤e
stop
skip

Figure 10: An example for binary span and shift.

The binary span operation can directly span 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 can use the two binary operations to replace the shift and span operations in order to skip valid windows. We give an algorithm to find candidate windows using the two operations as illustrated in Figure 11. 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

```
Algorithm 1: Find Candidate Windows
  Input: $e$: An entity; $P_e$: Position list of $e$ on the heap;
         $T_l$: Threshold; $\top_e$: The upper bound of token numbers;
1 begin
2     $i = 1$;
3     while $i \le |P_e| - T_l + 1$ do
4         $j = i + T_l - 1$;
5         if $p_j - p_i + 1 \le \top_e$ 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     $lower = j; upper = i + \top_e - 1$ ;
3     while $lower \le upper$ do
4         $mid = \lceil (upper + lower)/2 \rceil$;
5         if $p_{mid} - p_i + 1 > \top_e$ then $upper = mid - 1$;
6         else $lower = mid + 1$;
7     $mid = upper$ ;
8     Find candidate windows in $D[i \cdots mid]$;
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 $lower \le upper$ do
4         $mid = \lceil (lower + upper)/2 \rceil$;
5         if $\big(p_j + (mid - i)\big) - p_{mid} + 1 > \top_e$ then
6             $lower = mid + 1$;
7         else $upper = mid - 1$;
8     $i = lower; j = i + T_l - 1$;
9     if $p_j - p_i + 1 > \top_e$ then $i$ = BinaryShift $(i, j)$;
10    else return $i$;
11 end
```

Figure 11: Algorithm: Find candidate windows

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 10 illustrates an example to walk through our algorithm.

5. THE Faerie ALGORITHM

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

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 12 gives the pseudo-code of the Faerie algorithm.

The Faerie 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). Faerie 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. Faerie 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 Faerie checks the position list as follows. Faerie derives a threshold T_l for entity e based on the similarity function and threshold. If $|P_e| \ge T_l$, there may exist candidate pairs. Faerie generates candidate windows based on Algorithm 1 (line 13), and finds candidate pairs based on candidate windows (line 14). Faerie adjusts the heap to generate candidates for the next entity (line 16). Finally Faerie verifies the candidates to get final results (line 17).

```
Algorithm 2: Faerie 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 $P_e = \phi$;
7     while $\big(\langle e_i, p_i \rangle = H.top\big)! = \phi$ do
8         if $e_i == e$ then
9             $P_e \cup = \{p_i\}$; /* $e_i$ is the new top entity. */
10        else
11            Derive the threshold $T_l$ for entity $e$;
12            if $|P_e| \ge T_l$ 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
```

Figure 12: The Faerie Algorithm.

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 (Figure 5). Then we adjust the heap to generate the position list for each entity. Consider the position list for entity e_4 ("venkatesh") in Figure 10. 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.

6. EXPERIMENTS

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

Experimental Setting: We compared our algorithms with state-of-the-art methods NGPP [22] (the best for edit distance) and ISH [4] (the best for jaccard similarity and edit similarity). We downloaded the binary codes of NGPP [22] from "Similarity Joins" project website[6] and implemented ISH by ourselves. We reported the best performance of the two existing methods. 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 3.0GHz processor and 4 GB memory.

Datasets: We used three real datasets, DBLP[7], PubMed[8], and ACM WebPage[9]. DBLP is a computer-science publi-

[6] http://www.cse.unsw.edu.au/∼weiw/project/simjoin.html

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

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

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

cation dataset. We selected 100,000 author names as entities and 10,000 papers as documents. PubMed is a medical-publication dataset. We selected 100,000 paper titles as entities and 10,000 publication records as documents. WebPage is a set of web pages. We crawled 100,000 titles as entities and 1,000 web pages as documents (thousands of tokens). Table 4 gives the dataset statistics (len denotes the average length and token denotes the average token number). We did not consider different attributes in the entities and documents. Each entity in the dictionary is just a string.

Table 4: Datasets.

Datasets	Cardinality	len	tokens	Details
DBLP Dict	100,000	21.1	2.77	Author
DBLP Docs	10,000	123.3	16.99	Papers
PubMed Dict	100,000	52.96	6.98	Title
PubMed Docs	10,000	235.8	33.6	Papers
WebPage Dict	100,000	66.89	8.5	Title
WebPage Docs	1,000	8949	1268	Web Pages

6.1 Multi-Heap vs Single Heap

In this section, we compared the multi-heap-based method with the single-heap-based method (without using pruning techniques in Section 4). We tested the performance of the two methods for different similarity functions on the three datasets. Figure 13 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 DBLP dataset with edit-distance threshold $\tau = 3$, the multi-heap-based method took more than 10,000 seconds and the single-heap-based method took about 180 seconds. On PubMed dataset with EDS similarity threshold $\delta = 0.9$, the multi-heap-based method took more than 14,000 seconds and the single-heap-based method took only 600 seconds. 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. 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 Effectiveness of Pruning Techniques

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 14 gives the results. In the figure, we tested edit distance on 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, our method used parameters of $q = 16, 8, 5, 4, 3$ 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.

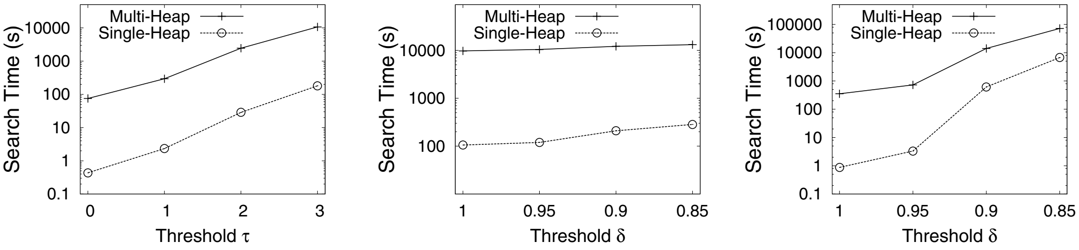
We observe that our proposed pruning techniques can prune large numbers of candidates. For example, on the DBLP dataset, for $\tau = 3$, the method without any pruning techniques involved 11 billion candidates, and the lazy-count pruning reduced the number to 860 million. The bucket-count pruning further 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 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.

Then we evaluated the performance benefit of the pruning techniques. Figure 15 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 seconds, and the lazy-count pruning decreased the time to 43 seconds. The binary span and shift pruning reduced the time to 25 second. On PubMed dataset, for $\delta = 0.9$, the pruning techniques can improve the time from 600 seconds to 8 seconds. This shows that our pruning techniques can improve the performance. In the remainder of this paper, we compared the single-heap-based method using the binary span and shift pruning with existing methods.

6.3 Comparison with State-of-the-art Methods

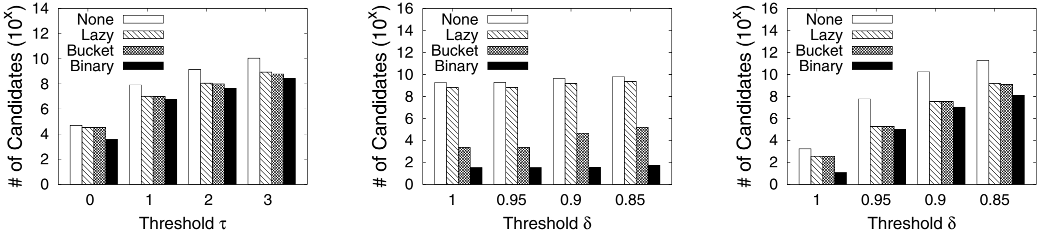
In this section, we compared our algorithm Faerie with state-of-the-art methods NGPP [22] (which only supports edit distance) and ISH [4] (which supports edit similarity and jaccard similarity). We tuned the parameters of NGPP and ISH (e.g., prefix length of NGPP) to make them achieve the best performance. Figure 16 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 1000 seconds. Faerie reduced the time to 8 seconds. 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 δ=1 and $T_l = 10$ for δ=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 larger thresholds. On DBLP dataset, for $\tau = 3$, NGPP con-



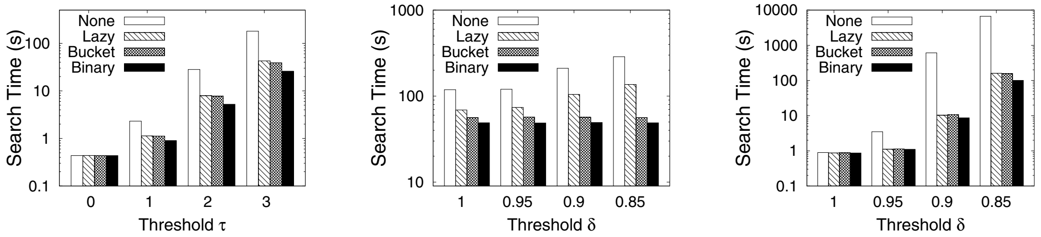
(a) ED (DBLP) (b) JAC (WebPage) (c) EDS (PubMed)

Figure 13: Performance comparison of multi-heap-based methods and single-heap-based methods.



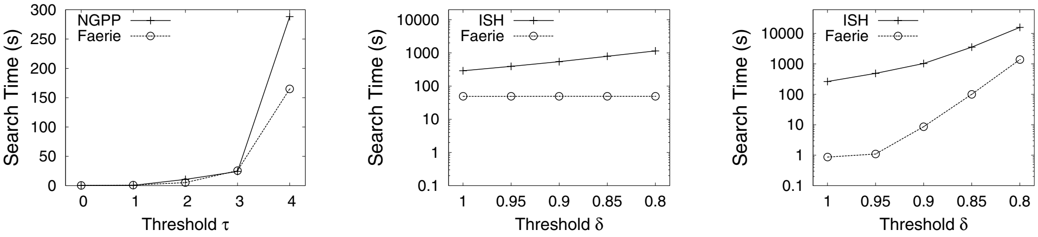
(a) ED (DBLP) (b) JAC (WebPage) (c) EDS (PubMed)

Figure 14: Number of candidates with different pruning techniques.



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

Figure 15: Performance comparison with different pruning techniques.



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

Figure 16: Comparison with existing methods.

sumed about 43 MB index size. The index size of Faerie was only 7 MB ($q = 4$). This result consists with that in [22]: 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.

6.4 Scalability with Dictionary Sizes

This section evaluates the scalability of our proposed 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 17 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 seconds for 20, 000 entities and 25 seconds 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 seconds for 20, 000 entities and 48 seconds 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.

Table 5: Scalability of index sizes.

(a) DBLP ($q = 5$)

# of Entities	20k	40k	60k	80k	100k
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

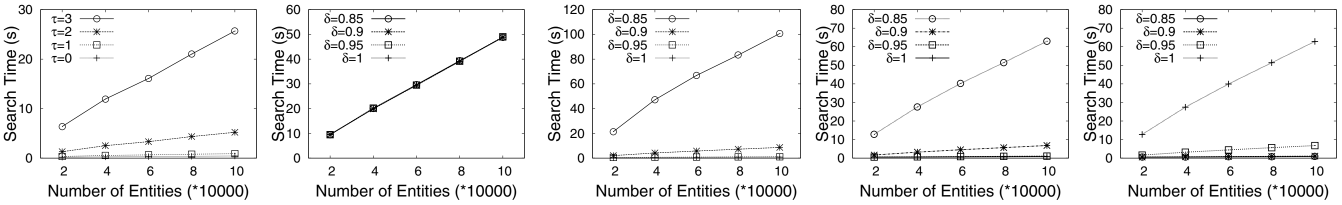
# of Entities	20k	40k	60k	80k	100k
Inverted Index (MB)	0.8	1.63	2.45	3.3	4.2
Heap+Array (KB)	38	38	38	38	38

(c) PubMed ($q = 7$)

# of Entities	20k	40k	60k	80k	100k
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

7. RELATED WORKS

There have been some recent studies on approximate entity extraction [22, 9, 20, 1, 4, 5]. Wang et al. [22] proposed neighborhood-generation-based methods for approximate entity extraction with edit-distance thresholds. They



(a) ED (DBLP) (b) JAC (WebPage) (c) EDS (PubMed) (d) DICE (PubMed) (e) COS (PubMed)

Figure 17: Scalability of performance for various similarity functions on different datasets.

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. However 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. [20] 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.

Many studies have been proposed to address the approximate-string-search problem [4, 14, 2, 8, 6, 11, 18, 19, 15, 25, 12] and the similarity-join problem [10, 2, 3, 7, 21, 23, 24]. 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[22] 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[22] and ISH[4].

In addition, there have been many studies on estimating selectivity for approximate string queries [13, 16, 17].

8. CONCLUSION

In this paper, we have studied the problem of approximate 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 have implemented our method, 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.

Acknowledgement

The authors thank Professor Wei Wang, Professor Chen Li, and the three anonymous reviewers for their comments and suggestions that definitely help us improve the paper.

9. REFERENCES

[1] S. Agrawal, K. Chakrabarti, S. Chaudhuri, and V. Ganti. Scalable ad-hoc entity extraction from text collections. *PVLDB*, 1(1):945–957, 2008.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

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

[4] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.

[5] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.

[6] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[8] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *ICDE*, pages 865–876, 2005.

[9] S. Chaudhuri, V. Ganti, and D. Xin. Mining document collections to facilitate accurate approximate entity matching. *PVLDB*, 2(1):395–406, 2009.

[10] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[11] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.

[12] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[13] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 1(1):201–212, 2008.

[14] M.-S. Kim, K.-Y. Whang, J.-G. Lee, and M.-J. Lee. n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In *VLDB*, pages 325–336, 2005.

[15] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[16] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.

[17] H. Lee, R. T. Ng, and K. Shim. Power-law based estimation of set similarity join size. *PVLDB*, 2(1):658–669, 2009.

[18] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[19] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[20] J. Lu, J. Han, and X. Meng. Efficient algorithms for approximate member extraction using signature-based inverted lists. In *CIKM*, pages 315–324, 2009.

[21] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[22] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, 2009.

[23] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

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

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