# An Efficient Trie-based Method for Approximate Entity Extraction with Edit-Distance Constraints

Dong Deng          Guoliang Li          Jianhua Feng

*Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.*

`dd11@mails.tsinghua.edu.cn, liguoliang@tsinghua.edu.cn, fengjh@tsinghua.edu.cn`

*Abstract*—Dictionary-based entity extraction has attracted much attention from the database community recently, which locates substrings in a document into predefined entities (e.g., person names or locations). To improve extraction recall, a recent trend is to provide *approximate* matching between substrings of the document and entities by tolerating minor errors. In this paper we study dictionary-based approximate entity extraction with edit-distance constraints. Existing methods have several limitations. First, they need to tune many parameters to achieve high performance. Second, they are inefficient for large edit-distance thresholds. To address these limitations, we propose a trie-based method to support efficient entity extraction. We develop a partition scheme to partition each entity into a set of segments and use a trie structure to index segments. To extract similar entities, we search segments from the document, and extend the matching segments in both entities and the document to find similar pairs. We develop an extension-based method to efficiently find similar string pairs by extending the matching segments. We optimize our partition scheme and select the best partition strategy to improve the extraction performance. Experimental results show that our method achieves much higher performance, compared with state-of-the-art studies.

## I. Introduction

Entity extraction (also known as entity recognition and entity identification) is an important operation in information extraction that locates substrings from a document into predefined entities, such as person names, locations, organizations, etc. Dictionary-based entity extraction has attracted much attention from the database community, which identifies substrings from a document that match the predefined entities in a given dictionary. For example, consider a document "*An efficient filter for approximates membership checking. kaushit chekrabarti, suraijt chauduri, vankatesh ganti, dong xin.*" and a dictionary with two entities "`surajit chaudhuri`" and "`dong xin`." Dictionary-based entity extraction locates entity "`dong xin`" from the document.

However, the document may contain orthographical or typographical errors and the same entity may have different representations [26]. For example, the substring "*suraijt chauduri*" 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`." To improve extraction recall, approximate entity extraction is a recent trend [26], [18], which finds substrings from the document that *approximately* match the predefined entities. This problem has many real applications in bioinformatics, molecular biology, and natural language processing.

To quantify the similarity between two strings, many similarity functions have been proposed. Edit distance is a well-known function which is widely adopted for tolerating typing mistakes and spelling errors. The edit distance between two strings is the minimum number of single-character edit operations (i.e., insertion, deletion, and substitution) needed to transform the first one to the second one. For instance, the edit distance between entity "`surajit chaudhuri`" and substring "*suraijt chauduri*" in the above document is 3. Suppose we use edit distance with threshold 3. Approximate entity extraction can locate the substring "*suraijt chauduri*" from the document which is similar to entity "`surajit chaudhuri`."

Faerie [18] and NGPP [26] have been proposed to address this problem, however they have some limitations. Firstly, they need to tune parameters to achieve a high performance, which is a tedious and troublesome process (see Section II-B). Secondly, they are inefficient for large edit-distance thresholds. To address these problems, we propose a trie-based method for dictionary-based approximate entity extraction with edit-distance constraints, called TASTE. TASTE does not need to tune parameters. Moreover TASTE achieves much higher performance, even for large edit-distance thresholds.

To achieve our goal, we propose a partition scheme to partition entities into several segments. We develop an efficient algorithm based on the fact that if a substring of the document is similar to an entity, the substring must contain a segment of the entity [23]. To this end, we first search segments of entities from the document, and then extend the matching segments in both entities and documents in order to find similar pairs. To facilitate the segment identification, we use a trie structure to index the segments and develop an efficient trie-based algorithm. We develop an efficient extension-based framework to find similar pairs by extending the matching segments. To summarize, we make the following contributions.

- We propose a partition scheme to partition entities into segments and utilize these segments to address the dictionary-based approximate entity extraction problem.
- We build a trie-structure for segments of entities to facilitate searching segments from the document. We propose an extension-based framework to efficiently find similar string pairs based on the matching segments.
- As they may be large numbers of partition strategies to generate segments of entities, we study how to select the best partition strategy to improve performance.

- We have implemented our method, and the experimental results show that our method achieves much higher performance, compared with state-of-the-art studies.

The rest of this paper is organized as follows. We first formulate the problem of dictionary-based approximate entity extraction in Section II. We propose a trie-based framework in Section III. Section IV gives efficient algorithms to find similar pairs. We discuss how to select the best partition strategy in Section V. Experiment results are provided in Section VI. We make a conclusion in Section VII.

## II. PRELIMINARIES

We first formulate the problem of dictionary-based approximate entity extraction, and then review related works.

### A. Problem Formulation

To tolerate inconsistencies between substrings and entities, in this paper, we use edit distance to quantity the similarity between two strings. 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, $\text{ED}(\text{marios}, \text{maras}) = 2$. In this paper two strings are *similar*, if their edit distance is not larger than a predefined edit-distance threshold $\tau$. Next we formulate our problem.

*Definition 1 (Approximate Entity Extraction):* Given a dictionary of entities $E = \{e_1, e_2, \ldots, e_n\}$, a document $D$, and a predefined edit-distance threshold $\tau$, approximate entity extraction finds all "*similar*" pairs $\langle s, e_i \rangle$ such that $\text{ED}(s, e_i) \leq \tau$, where $s$ is a substring of $D$ and $e_i \in E$.

*Example 1:* Consider dictionary $E$ and document $D$ in Table I. Suppose the edit-distance threshold $\tau = 2$. $\langle$"*surajit chaudhuri*", "surajit chaudri"$\rangle$ and $\langle$"*kaushit chekrab*", "caushit chakrab"$\rangle$ are two example answers.

TABLE I
A DICTIONARY OF ENTITIES AND A DOCUMENT.

| (a) Dictionary $E$ | | |
|---|---|---|
| ID | Entities | Len |
| 1 | vancouver | 9 |
| 2 | vanateshe | 9 |
| 3 | surajit chaudri | 15 |
| 4 | caushit chaudui | 15 |
| 5 | caushit chakrab | 15 |

| (b) Document $D$ |
|---|
| Document |
| *an efficient filter for approximates membership checking. kaushit chekrabarti, surajit chaudhuri, vankatesh ganti, dong xin. vancouver, canada. sigmod 2008.* |

### B. Related Works

**Approximate Entity Extraction:** There have been some recent studies on approximate entity extraction [26], [9], [20], [1], [4], [5]. Li et al. [18] and Wang et al. [26] studied the same problem as ours. The former (Faerie) proposed a unified framework to support various similar functions. Although Faerie can support edit distance, it is not specially designed for edit distance. In addition, Faerie used gram-based index structures which involved larger index sizes than our method. The latter (NGPP) used a neighborhood-generation-based method. NGPP first partitions entities into different partitions, and guarantees that an entity and a substring are

similar if they have two *similar* partitions with edit distance no larger than 1. Then NGPP generates neighborhoods of each partition by deleting one character, and the edit distance between two partitions is not larger than 1 if the two partitions have a common neighbor. NGPP involves larger indexes than gram-based methods [26]. To achieve a high performance, Faerie needs to tune the parameter gram length $q$ and NGPP needs to tune the parameter prefix length $l_p$. In addition, they are inefficient for large edit-distance thresholds(Section VI-C).

Chakrabarti et al. [4] proposed an inverted signature-based hash-table for membership checking, using a matrix identification based method. Lu et al. [20] improved this method [4] by using a tighter threshold. Agrawal et al. [1] used inverted lists for ad-hoc entity extraction. Chandel et al. [5] studied the problem of batch top-$k$ search for dictionary-based exact entity extraction. Chaudhuri et al. [9] mined document collections to expand a reference dictionary.

**Approximate String Search & Similarity Join:** Many methods [4], [13], [2], [8], [6], [10], [16], [17], [15], [28], [29], [11], [12] have been proposed to study the approximate string search problem, which, given a set of strings and a query string, finds all similar strings of the query string from the set. Existing methods usually employ a filter-and-verify framework. Similarity join has also been extensively studied [28], [27], [3], [7], [25], [24], which, given two sets of strings, finds all similar string pairs from the two sets. Existing methods usually employ a prefix-filtering-based framework to address this problem. Although we can extend the methods for approximate string search and similarity join, they are inefficient for approximate entity extraction (also proved in [26], [18]). The main reason is that they need to enumerate large numbers of overlapped substrings of the document. Different from [19] which used a partition-based method to support similarity joins, we use the partition scheme to support approximate entity extraction and develop effective trie-based indexes and search algorithms. Moreover, we propose effective techniques to optimize the partition scheme.

It has been shown that approximate entity extraction can improve extraction recall [26]. In this paper, we emphasize on improving the performance with a predefined threshold $\tau$.

## III. TRIE-BASED FRAMEWORK

We first introduce a partition scheme to partition each entity into different disjoint segments (Section III-A), and then use a trie structure to index the segments (Section III-B). Finally, we propose a framework to use the trie structure to efficiently find similar pairs (Section III-C).

### A. Partition Scheme

Given an entity $e$, we partition it into $\tau + 1$ disjoint segments, $e^1, e^2, \cdots, e^{\tau+1}$, and the length of each segment is not smaller than 1*. For example, consider entity $e_2$="vanateshe". Suppose $\tau = 2$. We have several ways to partition $e_2$ into 3 segments, such as {"vana","tes","he"} and {"van","ate","she"}.

---

*The length of entity $e(|e|)$ should be larger than $\tau$, i.e., $|e| \geq \tau + 1$.

For a substring $s$ of document $D$, if $s$ has no substring which is exactly a segment of $e$, $s$ cannot be similar to entity $e$ [23], which can be proved using the pigeon-hole principle.

Given an entity, there could be many strategies to partition the entity into $\tau + 1$ segments. Here we give an intuitive method, and we will discuss how to select the best partition strategy in Section V. Intuitively, the shorter a segment is, the higher probability it appears in a substring of the document, and thus the more substrings will be taken as candidates of those entities which contain the segment. In this way, we do not want to keep a short segment in the partition. In other words, each segment should nearly have the same length. Based on this observation, we propose an *even-partition* scheme. Consider an entity $e$ with length $|e|$. Let $k = |e| - \lfloor \frac{|e|}{\tau+1} \rfloor * (\tau + 1)$. In even partition, the first $k$ segments have length $\lceil \frac{|e|}{\tau+1} \rceil$, and others have length $\lfloor \frac{|e|}{\tau+1} \rfloor$, thus the maximal length difference between two segments is 1. For example, in even partition $e_2$="vanateshe" has three segments {"van", "ate", "she"}.

### B. Trie Index

For each substring of the document, we need to check whether it contains a segment of every entity. For efficient checking, we use a trie structure to index the segments of every entity. Each segment corresponds to a unique path from the root of the trie to a leaf node. The label of the root node is $\epsilon$, where $\epsilon$ is a special mark and denotes the empty string. Each of other nodes on the path has a label of a character in the segment. For simplicity, a node is mentioned interchangeably with its corresponding string in the remainder of the paper. Each leaf node has an inverted list of IDs of entities that contain the corresponding segment.

*Example 2:* Consider entities in Table I. Suppose we use the even partition and $\tau = 2$. Figure 1 shows the trie structure for the segments. The inverted list of node 25 ("it cn") is $3, 4, 5$, since entities $e_3, e_4, e_5$ contain the segment.


Fig. 1. Trie structure for segments in Table I.

**Space Complexity:** Each entity has $\tau + 1$ segments, thus the space complexity of inverted lists is $\mathcal{O}\big((\tau + 1) \times |E|\big)$, where $|E|$ is the number of entities. We use the compression ratio to evaluate degree of prefix sharing, which is the ratio of the sum of entity length to the number of nodes on the corresponding trie structure, denoted by $\lambda$. On real datasets $\lambda \in [3, 10]$. Thus the space complexity of the trie is $\mathcal{O}\big(\frac{\sum_{e \in E} |e|}{\lambda}\big)$.

### C. Trie-based Framework

We propose a trie-based framework to find substrings from document $D$ that approximately match entities. For each substring $s$ of $D$, we use the trie structure to find its similar entities. A naive method is to use every substring of $s$ to search the trie structure. If a substring of $s$ corresponds to a leaf node, the pair of $s$ and every entity in the inverted list of the leaf node is a candidate pair. However this method is rather inefficient as $s$ may have large numbers of substrings.

To improve the performance, we give an alternative method. For each suffix of substring $s$, we find the suffix in the trie structure. If we reach a leaf node, $s$ has a substring corresponding to the leaf node. We retrieve the entities in the inverted list, which may be similar to substring $s$.

**Valid Substrings:** We observed that some substrings of document $D$ will not be similar to any entity. For instance, the substring "*approximates membership*" cannot be similar to any entity, as its length is too large. To address this problem, we define *valid substrings* that are potentially similar to some entities. Let $L_{min}$ and $L_{max}$ respectively denote the minimal entity length and the maximal entity length in the dictionary. Obviously all the substrings of document $D$ with length smaller than $L_{min} - \tau$ or larger than $L_{max} + \tau$ can be pruned based on length filtering. We call the substrings of document $D$ with length between $L_{min} - \tau$ and $L_{max} + \tau$ *valid substrings*, which may have similar entities in the dictionary. For instance, consider the dictionary and document in Table I. $L_{min} = 9$ and $L_{max} = 15$. Suppose $\tau = 2$. The substrings with length between 7 and 17 could have similar entities, and others can be pruned. Thus we only enumerate valid substrings and use the above method to find their similar entities. This method is called TRIE-SEARCH. However many substrings share common segments and TRIE-SEARCH involves duplicated computations. Consider valid substrings "*surajit chaudhuri*", "*urajit chaudhuri*", and "*surajit chaudhur*". They share a common segment "it ch". TRIE-SEARCH searches "it ch" multiple times and accesses entities in the inverted list of "it ch" repeatedly. To avoid duplicated computations, we propose efficient algorithms in Section IV.

## IV. TRIE-BASED ALGORITHMS

We first propose a search-and-extension-based algorithm (Section IV-A), and then develop a trie-based pruning technique to improve the performance (Section IV-B).

### A. Search-and-Extension-based Algorithm

To avoid the duplicated computations on the shared segment across different substrings, we propose a search-and-extension-based algorithm. For each segment shared by multiple substrings, we only access the inverted list of the segment once. To achieve our goal, we first use a SEARCH operation to locate the segment using the trie and then employ an EXTENSION operation to find similar entities. For example, consider valid substrings "*surajit chaudhuri*", "*urajit chaudhuri*", and "*surajit chaudhur*". We first locate the segment "it ch". Then we extend the segment to find similar pairs, such as $\langle$"*surajit chaudhuri*", "surajit chaudri"$\rangle$.

For ease of presentation, we introduce several notations. Let $D[i,j]$ denote the substring of $D$ starting with the $i$-th character and ending with the $j$-th character. Let $D[i]$ denote the $i$-th character of $D$. Specially, $D$ starts with the 1-st character and ends with the $|D|$-th character, i.e., $D = D[1, |D|]$. Next we introduce the two operations:

- **SEARCH:** For each $1 \leq i \leq |D|$, the SEARCH operation checks whether each substring of $D$ starting with the $i$-th character, e.g., $D[i,j](j \geq i)$, is a trie leaf node (i.e., a segment), and finds all such leaf nodes, which are called *candidate nodes*.
- **EXTENSION:** For each candidate node $D[i,j]$ found in the SEARCH operation, the entities in its inverted list also contain segment $D[i,j]$, thus they may be similar to substrings that contain $D[i,j]$. The EXTENSION operation extends $D[i,j]$ to substrings $D[m,n](m \leq i, n \geq j)$ which are similar to an entity in the inverted list.

Our algorithms first calls the SEARCH operation for $1 \leq i \leq |D|$ to find substrings of the document starting with $D[i]$ that correspond to trie leaf nodes, e.g., $D[i,j]$. Then for each candidate node $D[i,j]$, the EXTENSION operation extends $D[i,j]$ to find similar substrings for every entity in the inverted list of $D[i,j]$. Iteratively, we can find all similar pairs.

*Example 3:* Consider $D[59, 73]$=“*kaushit chekrab*”. Let $i = 59$. The SEARCH operation finds leaf nodes corresponding to substrings starting with $D[59]$, and there is no such node. Next the SEARCH operation searches for $i = 60, 61, 62$, and so on. When $i = 64$, the SEARCH operation finds candidate node 25 (“it ch”=$D[64, 68]$). Entities $e_3, e_4, e_5$ in the inverted list may be similar to substrings containing $D[64, 68]$. Next the EXTENSION operation extends $D[64, 68]$ to find substrings which are similar to one of the three entities, e.g., $D[59, 73]$ similar to $e_5$=“*caushit chakrab*”.

**Implementation of the SEARCH operation:** Given $D[i]$, we find the candidate nodes from the root as follows. If the root has no child with a label $D[i]$, we terminate the SEARCH operation on $D[i]$; otherwise we visit its child with label $D[i]$ and check whether the node is a leaf node; if so, it is a candidate node. Next we continue to check whether the node has a child with label $D[i+1]$ and repeat the above steps. When reaching the node corresponding to $D[j]$ which has no child with label $D[j+1]$, we terminate the SEARCH operation on $D[i]$. This is because if there is no trie node corresponding to $D[i, j+1]$, there will not exist a node corresponding to $D[i, k](k > j)$. For example, consider $D[64, 70]$= “it chek”. Suppose $i = 64$. The SEARCH operation first checks whether the root has a child with label $D[64]$=“i”, and locates node 21. Then it locates nodes 22, 23, 24, 25. Node 25 is a candidate node as it is a leaf node. As node 25 has no child with label $D[69]$=“e”, we terminate the SEARCH operation for $i = 64$.

The complexity of this method is $\mathcal{O}(|D| \times L)$ where $L$ is the depth of the trie. To improve the performance, we can extend the Knuth-Morris-Pratt (KMP) algorithm [14] to support our SEARCH operation. KMP is used to search for occurrences of a “word” within a “text string”. Its main idea
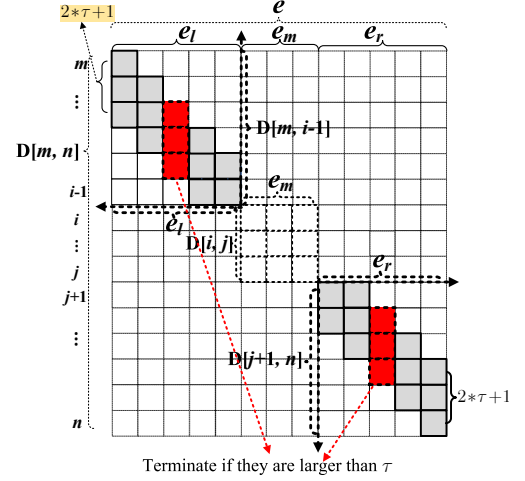


Fig. 2. EXTENSION on $D[i,j]$ for entity $e$.

is when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, and thus we can bypass re-examination of previously matched characters. For example, consider we search for “abcac” in “abcabcabcac”. When we find that “abcac” doest not match “abcab”, we do not need to search from the second character. Instead we search it from the current mismatch position, that is searching “ab<u>cac</u>” from “abca<u>bcabcac</u>”. To this end, we add some pointers on the trie structure to implement the “partial match table” [14] (a.k.a. “failure function”) and use the “partial match table” to bypass re-examination of previously matched characters. Thus we can improve the complexity to $\mathcal{O}(|D|)$.

**Implementation of the EXTENSION operation:** Consider a candidate node $D[i,j]$. Suppose entity $e$ is in the inverted list of $D[i,j]$(entity $e$ contains segment $D[i,j]$). Based on the segment, we can partition entity $e$ into three parts: the left part of $e$ before the segment (denoted by $e_l$), the matching part (denoted by $e_m = D[i,j]$), and the right part of $e$ after the segment (denoted by $e_r$). Next we find the set of substrings similar to $e_r$ starting with $D[j+1]$, denoted by $S_r$, and the set of substrings similar to $e_l$ ending with $D[i-1]$, denoted by $S_l$. For each $(s_l=D[l, i-1]) \in S_l$ and $(s_r=D[j+1, r]) \in S_r$, if ${\rm ED}(s_l, e_l) + {\rm ED}(s_r, e_r) \leq \tau$, the concatenate string of $D[l, i-1]$, $D[i,j]$, and $D[j+1, r]$, i.e., $D[l, r]$, is similar to entity $e$. We propose two operations to compute $S_r$ and $S_l$ (Figure 2).

(1) RIGHT EXTENSION: It extends $D[i,j]$ on the right side to generate $S_r$. Based on the length pruning, the maximal length of strings similar to $e_r$ is $|e_r|+\tau$, thus we at most extend it to $D[j+1, n]$ where $n = j + |e_r| + \tau$. We can use the dynamic-programming algorithm to compute the edit distance between $e_r$ and $D[j+1, n]$ (Figure 2). Note that in the Figure, we only need to compute the shaded entries [22]. If all the entries in a same column are larger than $\tau$, we can do an early termination and prune entity $e$; otherwise, we get the $2 \times \tau + 1$ entries of the last column, denoted by $M_{e_r}[n - 2 \times \tau \cdots n]$, which are respectively edit distances between $e_r$ and $D[j+1, n-2 \times \tau], \ldots, D[j+1, n]$. For $k \in [n - 2 \times \tau, n]$, if $M_{e_r}[k] \leq \tau$, $D[j+1, k]$ is similar to $e_r$. Thus $S_r = \{D[j+1, k] \mid M_{e_r}[k] \leq \tau, n - 2 \times \tau \leq k \leq n\}$.

(2) LEFT EXTENSION: It is similar to RIGHT EXTENSION and extends $D[i, j]$ on the left side to generate $S_l$. Based on the length pruning, we at most extend it to $D[m, i-1]$ where $m = i - (|e_l| + \tau)$. We can use the dynamic-programming algorithm to compute the edit distance between the reversed string of $e_l$ and the reversed string of $D[m, i-1]$ (Figure 2). Finally, we get the entries of the last column, denoted by $M_{e_l}[m \cdots m + 2 \times \tau]$, which are respectively edit distances between $e_l$ and $D[m, i-1], D[m+1, i-1], \ldots, D[m+2\times\tau, i-1]$. Obviously $S_l = \{D[k, i-1] \mid M_{e_l}[k] \leq \tau, m \leq k \leq m + 2 \times \tau\}$.

After RIGHT EXTENSION and LEFT EXTENSION, we get a set of similar substrings of $e_r$ ($S_r$) and a set of similar substrings of $e_l$ ($S_l$). ==Then we generate strings by concatenating a substring in $S_l$, $D[i, j]$, and a substring in $S_r$. If the sum of the edit distance of the first substring and that of the second substring is not larger than $\tau$, the pair of the generated string and $e$ is a similar pair.== The correctness of our method is formalized in Theorem 1.

*Theorem 1:* The search-and-extension-based method can find all similar pairs correctly and completely.

Here we give the high-level idea about the correctness of Theorem 1. Firstly if a substring $s$ is similar to an entity, $s$ must contain a segment of the entity. Consider a transformation from entity $e$ to substring $s$ with $\text{ED}(e, s)$ operations. In the transformation, there must exist a segment of $e$ such that the number of edit operations in the segment is 0 (otherwise $\text{ED}(e, s) \geq \tau + 1 > \tau$). We partition $e(s)$ into three parts: (1) the part before the segment, denoted by $e_l(s_l)$; (2) the matching part, denoted by $e_m(s_m)$; and (3) the part after the segment, denoted by $e_r(s_r)$. Obviously we have $\text{ED}(e, s) = \text{ED}(e_l, s_l) + \text{ED}(e_m, s_m) + \text{ED}(e_r, s_r)$. In this way, our extension-based method can find all such similar entities. Note that although a segment may have multiple occurrences in entity $e$, our method will consider all such cases. Thus our search-and-extension-based method finds results completely.

*Example 4:* For candidate node $D[64, 68]$ =“it ch”, its inverted list contains three entities, $e_3$, $e_4$, and $e_5$. Consider $e_5$=“caushit chakrab”. $e_l$=“caush”, $e_m$=“it ch”, and $e_r$=“akrab”. For $e_r$, we use the dynamic-programming algorithm to compute the edit distance between $e_r$ and $D[68 + 1, 68 + 5 + 2]$=“ekrabar”, and get $S_r = \{D[69, 72]$=“ekra”, $D[69, 73]$=“ekrab”, $D[69, 74]$=“ekraba”$\}$. Similarly we compute the edit distance between the reversed string of $e_l$ and the reversed string of $D[64-(5+2), 64-1]$=“.~kaush”, and get $S_l=\{D[58, 63]$=“~kaush”, $D[59, 63]$=“kaush”, $D[60, 63]$=“aush”, $D[61, 63]$=“ush”$\}$. Finally, we enumerate $S_l$ and $S_r$ to generate answers, e.g., the concatenate string of $D[59, 63]$, $D[64, 68]$, $D[69, 73]$ (i.e., $D[59, 73]$=“kaushit chekrab”) is similar to $e_5$.

To concatenate strings in $S_r$ and $S_l$, we do not need to do Cartesian product on $S_r$ and $S_l$. Instead we first partition substrings in $S_r$ and $S_l$ into different buckets based on their edit distances. We only concatenate the strings in two buckets with the sum of their edit distances no larger than $\tau$. In
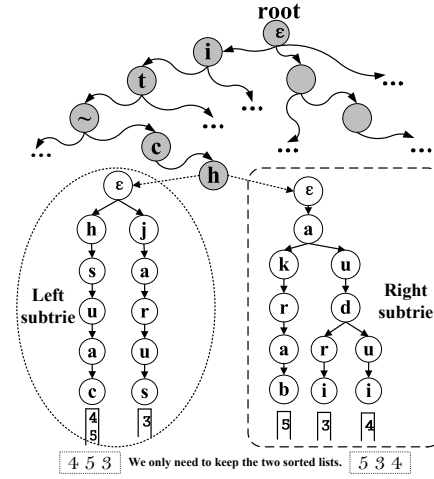


Fig. 3. Two-level trie structure.

addition, without loss of generality, suppose we first compute $S_r$. ==Let the minimal edit distance of substrings in $S_r$ be $\tau_r$. To compute $S_l$, we will use the threshold $\tau - \tau_r$,== instead of using $\tau$. In this way, we can avoid many unnecessary computations.

We give the time complexity. For each entity $e$ in the inverted lists of a candidate node, the extension time complexity is $\mathcal{O}\big((2\tau + 1) \times (|e| - |e_m|)\big) = \mathcal{O}\big(\frac{\tau(2\tau+1)}{\tau+1}|e|\big) = \mathcal{O}(\tau|e|)$, where $e_m$ is the matching segment. Suppose the total number of entities for all candidate nodes is $C$ and the average entity length is $L_{avg}$. The extension complexity is $\mathcal{O}\big(\tau L_{avg} C\big)^\dagger$. Thus the total time complexity of the search-and-extension-based method is $\mathcal{O}\big(|D| + \tau L_{avg} C\big)$. We can use the average entity length in the dictionary do estimate $L_{avg}$. We discuss how to compute $C$ in Section V-A.

### B. Improving The Extension Operation

In the EXTENSION operation, there may be large numbers of entities in the inverted list of a candidate node. It is expensive to do the EXTENSION operation for every entity. To address this issue, we propose a trie-based method.

In the trie structure, for each leaf node, we will not keep an inverted list of entities. Instead we maintain two subtries: one subtrie for right parts of the entities in the inverted list, and the other subtrie for the reversed strings of left parts of the entities. This new trie structure is called *two-level trie*. For example, Figure 3 shows a two-level trie. Consider leaf node “it ch” in the upper level trie. Entities $e_3$, $e_4$, and $e_5$ contain the segment. In the right subtrie of “it ch”, we keep right parts of the three entities, i.e., “audri”, “audui”, “akrab”. In the left subtrie, we keep the reversed strings of left parts, i.e., “suraj”, “caush”, “caush”.

The two-level trie has the following advantages. Firstly, as some entities share a common prefix (i.e., some trie nodes have a same ancestor), we can share the computations on the common prefix of these entities. Secondly, if the common prefix (i.e., a trie node) is not similar enough to a substring, we can prune all entities having the prefix (i.e., leaf nodes under the trie node w.r.t. the common prefix). This called subtrie

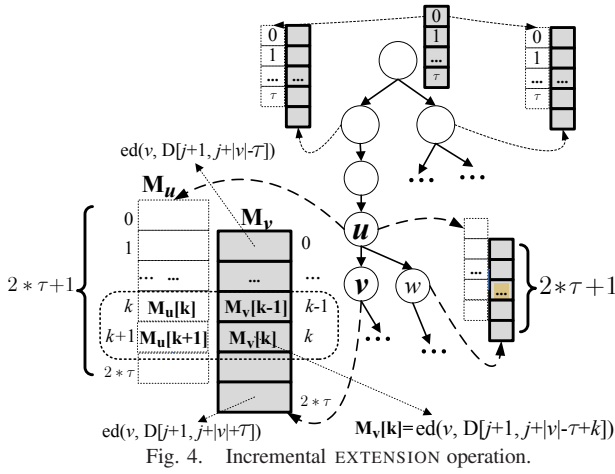$^\dagger$The total time complexity of concatenating strings in $S_r$ and $S_l$ is $\mathcal{O}(C)$.

**Fig. 4.** Incremental EXTENSION operation.

pruning [21]. Based on these observations, we use the two-level trie structure to improve the EXTENSION operation.

**Incrementally Computing $S_r$ and $S_l$:** Given a candidate node $D[i, j]$, we use the right subtrie and left subtrie to compute $S_r$ and $S_l$ respectively. We propose an *incremental extension operation* (Figure 4). We first consider the right subtrie. For any trie node $u$, we keep an array $M_u[0 \cdots 2 \times \tau]$, which is used to maintain the edit distance between node $u$ and its similar substrings $D[j+1, j+|u|-\tau], \ldots, D[j+1, j+|u|+\tau]$, where $|u|$ is the level of node $u$ in the subtrie (the level of the root node is 0). Obviously $M_u[k]$ corresponds to the edit distance between $D[j + 1, j + |u| - \tau + k]$ and node $u$.

Initially, for the root node, only the substring $D[j+1, j+k](0 \le k \le \tau)$ [‡] is similar to the empty string $\epsilon$ (i.e., the root) with edit distance $k$, thus we set $M_\epsilon[k] = k$. Next we discuss how to compute $M_u$ incrementally.

Suppose we have computed $M_u$. Then we use $M_u$ to compute $M_v$, where $v$ is a child of $u$. Based on the dynamic-programming algorithm (Figure 4), for $0 \le k \le 2 \times \tau$, we have

$$M_v[k] = \min(M_u[k] + c, M_u[k+1] + 1, M_v[k-1] + 1),$$

where $c = 0$ if the character $D[j + |v| - \tau + k]$ is the same as the label of $v$; otherwise $c = 1$. Especially, $M_v[-1] = \text{ED}(v, D[j+1, j+|v|-\tau-1]) > \tau$. As we only keep the value no larger than $\tau$, we set $M_v[0] = \min(M_u[0] + c, M_u[1] + 1)$. Similarly, as $M_u[2 \times \tau + 1] > \tau$, we set $M_v[2 \times \tau] = \min(M_u[2 \times \tau] + c, M_v[2 \times \tau - 1] + 1)$.

Based on the recursion formula and initial values, we can compute $M_v$ based on $M_u$. If each value in $M_v$ is larger than $\tau$, we will not visit $v$' children (subtrie pruning); otherwise, we continue the EXTENSION operations for $v$'s children. Finally, for each leaf node $l$, if $M_l[k] \le \tau$, the entities in $l$'s inverted list are similar to $D[j + 1, j + |l| - \tau + k]$. Thus for each entity $e_r$ in the inverted list of $l$, if $M_l[k] \le \tau$, we insert $D[j+1, j+|l|-\tau+k]$ into $S_r$ for $e_r$. Similarly, we compute $S_l$ using the left subtrie. The correctness of the incremental extension based method is formalized in Theorem 2.

*Theorem 2:* The incremental extension based method computes $S_r$ and $S_l$ correctly and completely.

[‡]Especially $D[j + 1, j] = \epsilon$.

The basic idea of Theorem 2 is to use the dynamic programming algorithm to compute $S_r$ and $S_l$ incrementally. For example, consider the two-level trie in Figure 3. For the right subtrie, we compute $M_{\text{audr}}$ and $M_{\text{audu}}$ based on $M_{\text{aud}}$. Thus we avoid the duplicated computations on the common prefix "aud". For substring $D[69, 71]=$"*ekr*", as it is not similar to "aud", we prune the subtrie rooted at "aud".

**SORT-BASED EXTENSION METHOD:** However, the trie-based extension needs to construct subtries for each leaf node which will involve large indexes. To address this issue, we propose a SORT-BASED EXTENSION METHOD, which does not construct the subtries but achieves the same high performance. Consider a leaf node in the upper level trie. We traverse its subtrie in preorder. When reaching a node $v$, we compute $M_v$ using its parent array $M_u$ (which has been computed in preorder). We keep $M_v$ as it will be used by its children. When backtracking to node $v$, we delete $M_v$ as its children have been visited and $M_v$ will not be used for subsequent nodes. The number of kept arrays is at most the depth of the subtrie. To this end, for each leaf node, we sort the right parts of entities in its inverted list in lexicographic order (also sort the reversed strings of left parts). Instead of keeping two subtries, we maintain two sorted lists of IDs of entities based on their lexicographic order. For RIGHT EXTENSION, we access entity IDs in order. Consider two adjacent entities $e$ and $e'$. Suppose we have computed $M_e$ for the right part of $e$. Then we compute $M_{e'}$ as follows. We first find the longest common prefix of right parts of $e$ and $e'$, denoted by $c$. As we have computed $M_c$ when computing $M_e$, we use $M_c$ to compute $M_{e'}$ (Figure 4).

*Example 5:* Consider the sorted right parts of $e_5, e_3, e_4$ ("akrab", "audri", "audui"). As "audri" and "audui" share a prefix "aud", when computing the array for "audui", we use that of "aud" which has been computed by "audri".

Next we give the space and time complexity analysis of the sort-based method. The space complexity of the trie structure is still $\mathcal{O}\left(\frac{\sum_{e \in E} |e|}{\lambda}\right)$. For the inverted lists, for each leaf node, we keep two sorted lists, thus the space complexity of inverted lists is $\mathcal{O}\left(2(\tau+1) \times |E|\right)$. For time complexity, we also share the computations of common prefixes for the right parts and left parts. Suppose the compression ratio is $\lambda$[§]. The total time complexity of the sort-based method is $\mathcal{O}\left(|D| + \frac{\tau L_{avg} C}{\lambda}\right)$.

## V. OPTIMIZING PARTITION SCHEME

In this section, we first introduce how to evaluate the quality of a partition strategy (Section V-A) and then study how to select the best partition strategy (Section V-C). Finally we develop effective pruning techniques and algorithms to efficiently find the best partition strategy (Section V-B).

### A. Evaluating Partition Strategies

Given an entity $e_i \in E$, let $P_{e_i}$ denote the set of partition strategies on $e_i$. It is easy to figure out that the set of partition strategies for all entities is $P = P_{e_1} \times \cdots \times P_{e_n}$. Consider a

[§]Although the compression ratio is a little different from that of the trie structure, they are similar. Thus here we use $\lambda$ for simplicity.

partition strategy $p \in P$. Let $G_p$ denote the set of segments in partition strategy $p$. For each segment $g \in G_p$, let $L_g$ denote the number of entities containing $g$, i.e., the size of the inverted list of leaf node w.r.t. $g$. Let $W_g$ denote the occurrence number of $g$ in the document, which is called the *weight* of $g$.

Recall our search-and-extension based method, for the SEARCH operation, we need to find candidate nodes. For each candidate node, we need to do the EXTENSION operation for the entities in the inverted lists of the candidate node. We call such entities *candidates*. Obviously for a partition $p$, there are $C_p = \sum_{g \in G_p} W_g \times L_g$ candidates. Based on the time complexity $\mathcal{O}(|D| + \frac{\tau L_{avg} C_p}{\lambda})$, the smaller $C_p$, the higher performance. Thus we want to find the best partition strategy $p \in P$ to minimize $C_p$, that is,

$$argmin_{p \in P} \sum_{g \in G_p} W_g \times L_g. \tag{1}$$

*Example 6:* Consider entity $e_2 = $ "vanateshe". The even partition has three segments "van", "ate", "she". The occurrence number of "van" in the document is 2, thus its weight is 2. Similarly the weights of "ate" and "she" are respectively 2 and 1. In the EXTENSION operation, $e_2$ will be considered five times. If we partition $e_2$ into "vana", "tes", "he" with weights respectively 0, 1, 0, we only extend $e_2$ once. Thus we want to select the best partition scheme.

A brute-force algorithm to find the best partition strategy first enumerates all partition strategies, and then computes the number of candidates for each partition strategy. Finally it selects the partition strategy with the minimum candidate number. However this method is inefficient as there are larger numbers of partition strategies. Consider entity $e_i$. To partition $e_i$ into $\tau + 1$ segments, there are $|P_{e_i}| = \binom{|e_i|-1}{\tau}$ partition strategies. On the other hand, the number of entities is large. This algorithm needs to enumerate $|P| = |P_{e_1} \times \cdots \times P_{e_n}|$ partition strategies, which is nearly incalculable. To address this issue, we propose an efficient algorithm in Section V-B.

### B. Algorithms to Minimize $C_p$

Given a partition strategy $p$, as $L_g$ is the number of entities that contain segment $g$, we have $L_g = \sum_{e_i \in E} \sum_{g \in G_{p_{e_i}}} 1$, where $p_{e_i}$ is the partition strategy for entity $e_i$ in partition $p$. Based on this observation, we rewrite Formula 1 as follows.

$$argmin_{p_1 \in P_{e_1}, \ldots, p_n \in P_{e_n}} \sum_{e_i \in E} \sum_{g \in G_{p_{e_i}}} W_g. \tag{2}$$

That is for each entity $e_i$, we only need to find the best partition strategy $p_{e_i} \in P_{e_i}$ to minimize $\mathcal{W}_{p_{e_i}} = \sum_{g \in G_{p_{e_i}}} W_g$ ($\mathcal{W}_{p_{e_i}}$ is called the partition weight of $p_{e_i}$), i.e.,

$$argmin_{p_{e_i} \in P_{e_i}} \sum_{g \in G_{p_{e_i}}} W_g. \tag{3}$$

A straightforward algorithm first enumerates all partition strategies for entity $e_i$, and then selects the partition strategy $p_{e_i}$ with the minimum weight $\mathcal{W}_{p_{e_i}}$. However there are $\binom{|e_i|-1}{\tau}$ partition strategies. If $\tau$ is large, the algorithm is

rather expensive. To address this issue, we propose a dynamic-programming algorithm.

Given an entity $e = c_1 c_2 \cdots c_{|e|}$, we consider its last segment. Note that only $c_{|e|}$, $c_{|e|-1} c_{|e|}$, $\cdots$, $c_{\tau+1} c_{\tau+2} \cdots c_{|e|}$ could be the last segment. Suppose the last segment is $c_k \cdots c_{|e|}$ ($\tau + 1 \leq k \leq |e|$). We need to partition the other part $c_1 \cdots c_{k-1}$ into $\tau$ segments. Suppose the minimal weight to partition $c_1 \cdots c_{k-1}$ into $\tau$ segments is $\mathcal{W}_{k-1}$, and the minimal weight to partition $e$ into $\tau + 1$ segments is $\mathcal{W}_{|e|}$. We have $\mathcal{W}_{|e|} = \min\{\mathcal{W}_{k-1} + W_{c_k \cdots c_{|e|}} \mid \tau + 1 \leq k \leq |e|\}$. In this way, we can devise a dynamic-programming algorithm.

We use a $(\tau + 1) \times |e|$ matrix $M$ to compute the best partition strategy. In the matrix, $M[i][j]$ denotes the minimal weight to partition $c_1 \cdots c_j$ into $i$ segments. Specially $M[\tau + 1][|e|]$ is the minimal weight to partition $e$ into $\tau + 1$ segments. Initially $M[1][j] = W_{c_1 \cdots c_j}$ for $1 \leq j \leq |e|$. Next for $i > 1$ and $i \leq j \leq |e|$, Equation 4 gives the recursion formula.

$$M[i][j] = \min \begin{cases} M[i-1][i-1] + W_{c_i \cdots c_j} \\ \vdots \\ M[i-1][j-1] + W_{c_j} \end{cases} \tag{4}$$

Based on the initial values and the above recursion formula, we can easily compute all values in the matrix and get the partition strategy with the minimum weight using backtracking. The time complexity for partitioning entity $e$ is $\mathcal{O}(\tau |e|^2)$, and the total partition complexity is $\mathcal{O}(\tau \times \sum_{e \in E} |e|^2)$.

If we partition entities offline, the dynamic-programming algorithm is acceptable, since entities are not very long. If we need to partition entities online (Section V-C), the partition time is included in the extraction time and we need to improve the partition performance. To this end, we propose several pruning techniques.

**(1) Using Segment Length to Do Pruning.** Given an entity, consider the matrix $M[\tau + 1][|e|]$. For the $i$-th row $M[i][*]$, we do not need to compute all entries from $j = 1$ to $j = |e|$. Next we give the start position and the end position of $j$. Consider an entry $M[i][j]$, which denotes partitioning the first $j$ characters of $e$ into $i$ segments. As each segment has at least 1 character, we have $j \geq i$. In addition, as we partition $e$ into $\tau + 1$ segments, there are $\tau + 1 - i$ segments in the last $|e| - j$ characters. Thus $|e| - j \geq \tau + 1 - i$, that is $j \leq |e| - (\tau + 1 - i)$. In this way the start position is $i$ and the end position is $|e| - (\tau + 1 - i)$. Thus in the $i$-th row, we only need to compute $|e| - \tau$ entries $M[i][i] \cdots M[|e| - (\tau + 1 - i)]$. The complexity is improved to $\mathcal{O}(\tau(|e| - \tau)^2)$.

**(2) Using Even-Partition Weight as An Upper Bound.** As the weight of the best partition strategy is not larger than that of the even partition, we can use the weight of the even partition as an upper bound when selecting the best partition strategy. Suppose the weight of even partition is $\mathtt{W_{even}}$. If we can easily deduce that $M[i][j] > \mathtt{W_{even}}$, we do not need to compute the value $M[i][j]$. The above length based technique only uses the fact that each segment has at least one character. By using the bound of even partition, next we give tighter start position and the end position of $j$.

Suppose $e = c_1 c_2 \cdots c_{|e|}$ where $c_i$ is a character. We have $W_{c_i \cdots c_j} \geq W_{c_i \cdots c_{j+1}}$ for $i < j$. We find the shortest substring of $e$ starting with the first position, denoted by $c_1 \cdots c_{s_1}$, whose weight is not large than $\mathsf{W_{even}}$, that is

$$W_{c_1 \cdots c_{s_1}} \leq \mathsf{W_{even}} \quad \& \quad W_{c_1 \cdots c_{s_1 - 1}} > \mathsf{W_{even}} \P.$$

Thus the end position of any segment starting with the first position must be no smaller than $s_1$. Next from the position $s_1 + 1$, we find the shortest substring of $e$, denoted by $c_{s_1+1} \cdots c_{s_2}$, whose weight is not larger than $\mathsf{W_{even}}$, that is

$$W_{c_{s_1+1} \cdots c_{s_2}} \leq \mathsf{W_{even}} \quad \& \quad W_{c_{s_1+1} \cdots c_{s_2 - 1}} > \mathsf{W_{even}}.$$

Similarly we generate $\tau$ positions $s_1, s_2, \cdots, s_\tau$. The value $s_i$ means that if a prefix of $e$ has $i$ segments, the prefix length cannot be smaller than $s_i$. In other words, for the $i$-th row, the start position of $j$ must be not smaller than $s_i$, i.e., $j \geq s_i$. We do not need to compute $s_{\tau+1}$, as in the $(\tau + 1)$-th row, we only compute $M[\tau + 1][|e|]$ and do not use the bounds.

Similarly we can also find $\tau$ positions from the end position of $e$ and get $d_1, d_2, \cdots, d_\tau$ such that for $1 \leq i \leq \tau$,

$$W_{c_{d_i} \cdots c_{d_{i-1}+1}} \leq \mathsf{W_{even}} \quad \& \quad W_{c_{d_i+1} \cdots c_{d_{i-1}+1}} > \mathsf{W_{even}}.$$

Note that $d_1 > d_2 > \cdots > d_\tau$. Especially $d_0 + 1 = |e|$.

For the $i$-th row, the end position of $j$ must be not larger than $d_{\tau+1-i}$, i.e., $j \leq d_{\tau+1-i}$, thus we only compute $M[i][s_i], \cdots, M[i][d_{\tau+1-i}]$ as formalized in Lemma 1.

*Lemma 1:* Consider $j < s_i$ or $j > d_{\tau+1-i}$. If we partition the first $j$ characters of $e$ into $i$ segments and other characters into $\tau + 1 - i$ segments, then the partition weight must be larger than $\mathsf{W_{even}}$.

In addition, we observe that the weight of a string is not larger than the weight of any of its substrings. That is for $x \leq j$, if $W_{c_x \cdots c_j} > \mathsf{W_{even}}$, we have

$$\mathsf{W_{even}} < W_{c_x \cdots c_j} \leq W_{c_{x+1} \cdots c_j} \leq \cdots \leq W_{c_j}.$$

Let $d_{min}$ denote the start position of the shortest substring with end position $j$, whose weight is not larger than $\mathsf{W_{even}}$. We call $d_{min}$ the minimal end position for $j$. When computing $M[i][j]$, we only use entries $M[i-1][x]$ for $x < d_{min}$ and prune the entries $M[i-1][y]$ for $y \geq d_{min}$, as $M[i-1][y] + W_{c_{y+1} \cdots c_j} > \mathsf{W_{even}}$.

Based on Lemma 1 and this observation, we give a new method to compute the best partition strategy. We still use the matrix $M$ but we only need to compute some entries of $M$. Initially for $s_1 \leq j \leq d_\tau$, $M[1][j] = W_{c_1 \cdots c_j}$. Next for $i > 1$ and $s_i \leq j \leq d_{\tau+1-i}$, Equation 5 gives the recursion formula.

$$M[i][j] = \min \begin{cases} M[i-1][s_i] + W_{c_{s_i+1} \cdots c_j} \\ \vdots \\ M[i-1][d_{min} - 1] + W_{c_{d_{min}} \cdots c_j} \end{cases} \quad (5)$$

where $d_{min}$ is the minimal end position of $j$.

Moreover, we observed that for any $i$ and $j$, $M[i][j-1] \geq M[i][j]$. Thus we can deduce a tighter start position bound

$(s_i)$. When calculating the $i$-th row of the matrix $M$, if $M[i][j-2]$ is larger than $\mathsf{W_{even}}$ and $M[i][j-1]$ is not larger than $\mathsf{W_{even}}$, we can set $s_i = j - 1$. Then when computing the $(i+1)$-th row, we can deduce a new tighter start position bound $s_{i+1}$ using the old bound $s_i$.

Based on this observation, when computing $M[i][j]$, we use $M[i][j-1]$ as a tighter bound which is not larger than $\mathsf{W_{even}}$. To this end, after computing $M[i][j-1]$, we can keep the position $s_{max}$ such that $M[i-1][s_{max}-1] > M[i][j-1]$ and $M[i-1][s_{max}] \leq M[i][j-1]$. $s_{max}$ is called the maximal start position for $j$. Then to compute $M[i][j]$, we can directly start from the $s_{max}$-th position, i.e. $M[i-1][s_{max}] + W_{c_{s_{max}+1} \cdots c_j}$. Then we update the position $s_{max}$ for $M[i][j]$, that is $M[i-1][s_{max}-1] > M[i][j]$ and $M[i-1][s_{max}] \leq M[i][j]$.

In this way, we use the following formula to compute the matrix $M$, which prunes many unnecessary entries.

$$M[i][j] = \min \begin{cases} M[i-1][s_{max}] + W_{c_{s_{max}+1} \cdots c_j} \\ \vdots \\ M[i-1][d_{min} - 1] + W_{c_{d_{min}} \cdots c_j} \end{cases} \quad (6)$$

where $d_{min}$ is the minimal end position and $s_{max}$ is the maximal start position for $j$.

Next we give the time analysis of the method using this pruning technique. Usually the weight of a segment to the segment length roughly follows a power law distribution. That is $F(x) = \alpha \times x^\beta$, where $F(x)$ is the weight of segment with length $x$. Thus we have $\mathsf{W_{even}} = \alpha \times (\frac{|e|}{\tau+1})^\beta \times (\tau + 1)$. Consider the minimal segment length is $l_m$. If $\alpha \times l_m^\beta > \alpha \times (\frac{|e|}{\tau+1})^\beta \times (\tau + 1)$, we can prune the segment. In other words we have $l_m \leq |e|(\tau+1)^{\frac{1}{\beta} - 1}$. In this way, in the $i$-th row, we only need to compute the columns $M[i][j]$ for $i \times l_m \leq j \leq |e| - (\tau+1-i) \times l_m$. Thus for each row, we at most compute $|e| - (\tau+1) \times l_m$ entries. It is easy to figure out that this pruning technique reduces the time complexity to $\mathcal{O}(\tau(|e| - |e|\tau^{\frac{1}{\beta}})^2)$.

### C. Determining Segment Weights

In this section, we discuss how to determine the weight $W_g$ of segment $g$. If the document is given, we can construct a suffix tree. The suffix tree for document $D$ is a tree structure, where the paths from the root to leaves have a one-to-one correspondence with the suffixes of $D$. For ease of presentation, we use a trie to represent the suffix tree.

To construct the trie structure, a straightforward method inserts all suffixes of $D$ into the trie structure. For each trie node, the string from the root to the node corresponds to a possible segment, and the number of its leaf descendants is exactly its occurrence number in the document, i.e., the weight of the segment. In this way, we can easily get the weight of each segment using the trie structure.

However, we do not need to use all the suffixes of $D$ to construct the trie structure, because some long suffixes will not correspond to any segment. For example, consider the suffix "*vancouver, canada. sigmod 2008*". As its length is longer than the maximal entity length, we do not need to insert it into the trie structure. Based on this observation, we propose

an alternative method. Suppose $L_{max}$ is the maximal entity length. The maximal length of a segment is $L_{max} - \tau$. We only need to insert the substrings of $D$ with length no larger than $L_{max} - \tau$ into the trie structure. In this way, we can reduce the size of the suffix trie. The total number of inserted substrings is $\sum_{l=1}^{L_{max}-\tau} |D| - l + 1$. The time complexity of inserting a substring with length $l$ is $\mathcal{O}(l)$, thus the total time complexity is $\mathcal{O}\left(\sum_{l=1}^{L_{max}-\tau} l(|D| - l + 1)\right)$.

Actually, if we use the document to determinate weights, the time of constructing suffix trie and partitioning the entities should be included in the running time for online extraction. Note that it is rather time and space expensive to build a suffix trie for a very large document, which is even longer than the extraction time using the even partition. To address this problem, we propose two alternative methods.

**(1) Using Even Partition (EVEN).** We use the even partition-based method. We can partition the entities offline. Thus the online time complexity is $\mathcal{O}\left(|D| + \frac{\tau L_{avg} C_E}{\lambda}\right)$, where $C_E$ is the number of candidates using the even-partition-based method.

**(2) Using Both Dictionary and Document to Determine The Weights (DICT+DOC).** We use both dictionary and documents to accurately compute the segment weights based on the following observation. If a substring $g$ of document $D$ does not appear in the dictionary ($g$ is not a segment of any entity), we have $L_g = 0$. In this way, $g$ will not contribute to any partition strategy. In other words, we only need to use the weights of substrings which appear both in the dictionary and the document. To this end, we first build a suffix trie using the dictionary offline. Then for each suffix of document $D$, we check whether it appears in the trie structure. If yes, we add the weight by one. However this method is expensive and we can also extend the KMP algorithm [14] to improve the performance as follows.

For each trie node for a substring $c_1 c_2 \cdots c_x$, where $c_i$ is a character for $1 \leq i \leq x$, we add a pointer to the trie node w.r.t. its suffix "$c_2 \cdots c_x$". Then given the document $D = c_1 c_2 \cdots c_{|D|}$, we search the document from the root node and find the *maximal matching node* which matches the longest substring of $D$. For example we find trie node w.r.t $c_1 c_2 \cdots c_i$, and it has no child node matching $c_{i+1}$. Then based on the pointer, we locate to node $c_2 \cdots c_i$ and check whether it has a child matching $c_{i+1}$. If yes, we find the maximal matching node under $c_2 \cdots c_{i+1}$; otherwise we locate to node $c_3 \cdots c_i$. Iteratively we can find all *the maximal matching nodes*. The time complexity of finding maximal matching nodes is $\mathcal{O}(|D|)$. Next for each maximal matching node, we increase the weight of its ancestors by one (including itself). The time complexity of increasing the weight is $\mathcal{O}(\sum |maxnode|)$, where $|maxnode|$ denotes the depth of a maximal matching node.

Notice that constructing the suffix trie structure can be done offline, and determining weights $\left(\mathcal{O}(|D| + \sum |maxnode|)\right)$, selecting partition strategy $\left(\mathcal{O}(\sum_{e \in E} \tau(|e| - |e|\tau^{\frac{1}{\beta}})^2)\right)$, and building trie and inverted lists $\left(\mathcal{O}(\sum_{e \in E} |e|)\right)$ are included in the online extraction time. Thus the total time complexity of

this method is $\mathcal{O}(|D| + \sum |maxnode|) + \mathcal{O}(\sum_{e \in E} \tau(|e| - |e|\tau^{\frac{1}{\beta}})^2) + \mathcal{O}(\sum_{e \in E} |e|) + \mathcal{O}(|D| + \frac{\tau L_{avg} C_D}{\lambda})$, where $C_D$ is the number of candidates using the DICT+DOC based method. Note that the partition time is very small compared with the extraction time (Section VI-B), and the DICT+DOC based method can minimize the number of candidates, thus the DICT+DOC based method can improve the performance.

## VI. EXPERIMENTAL STUDY

In the paper we focus on evaluating the performance of different methods. We have implemented our proposed algorithms and compared with state-of-the-art methods NGPP [26] and Faerie [18]. We downloaded the binary codes of NGPP [26] from its project website$^{\parallel}$. We implemented Faerie by ourselves. All the algorithms were implemented in C++ and compiled using GCC 4.2.3 with -O3 flag. All the experiments were run on a Ubuntu machine with an Intel Core E5420 2.5GHz CPU and 4 GB memory.

We used three real datasets, DBLP**, PUBMED$^{\dagger\dagger}$, and Wiki WEBPAGE$^{\ddagger\ddagger}$. PUBMED is a medical-publication dataset. We selected 300k author names as entities and 10k publication records as documents. DBLP is a computer-science publication dataset. We selected 100k paper titles as entities and 10k paper records as documents. WEBPAGE is a set of web pages including articles and URLs. We selected 100k URLs with highest PageRank scores from Sogou URL dataset$^{10}$ as entities. We used 1k Webpages as documents (each document contains thousands of tokens). Table II shows the statistics of the three datasets, where len, max, min respectively denote the average length, maximal length, and minimal length.

TABLE II
DATASETS.

| Datasets | Size | len | max | min | Details |
|---|---|---|---|---|---|
| PUBMED Dict | 300k | 16.55 | 19 | 13 | Author |
| PUBMED Docs | 10k | 161.9 | 481 | 56 | Papers |
| DBLP Dict | 100k | 56.7 | 73 | 36 | Title |
| DBLP Docs | 10k | 1056 | 1964 | 82 | Papers |
| WEBPAGE Dict | 100k | 26.5 | 78 | 20 | URL |
| WEBPAGE Docs | 1k | 2879 | 14771 | 71 | Web Pages |

### A. Search-Extension vs Sort-based Extension

In this section, we compare our search-and-extension-based method (SEARCH-EXTENSION, see Section IV-A) with the sort-extension-based method (SORT-EXTENSION, see Section IV-B). As TRIE-SEARCH (Section III-C) was very slow, even 2-3 orders of magnitude slower than SEARCH-EXTENSION, we did not report its results. In addition, the trie-based extension achieved the same performance as SORT-EXTENSION, but at expense of involving large space for maintaining the two-level trie structure. Thus we only reported the results of SORT-EXTENSION. In this section, we used the even partition to partition entities. Figure 5 shows the results.
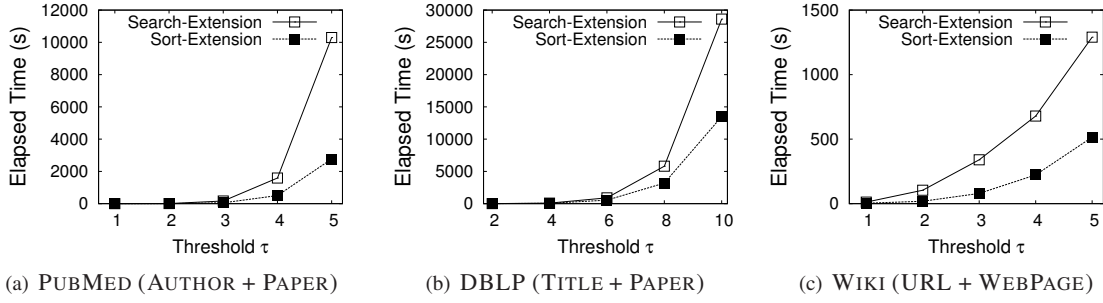
Fig. 5.    Performance comparison: SEARCH-EXTENSION vs SORT-EXTENSION.
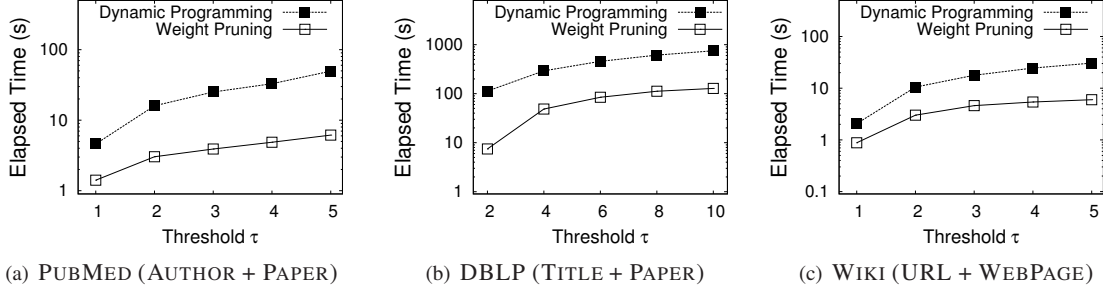


Fig. 6.    Partition comparison: Dynamic Programming vs Weight Pruning.

On the PUBMED (AUTHOR + PAPER) and WIKI (URL + WEBPAGE) datasets, as entities have small lengths (about 20, see Table II), we used thresholds $1, 2, 3, 4, 5$. On the DBLP (TITLE + PAPER) dataset, as entities have large lengths (about 60, see Table II), we used thresholds $2, 4, 6, 8, 10$. From Figure 5, we can see that SORT-EXTENSION was much better than SEARCH-EXTENSION, especially for large edit-distance thresholds. For example, on the PUBMED (AUTHOR + PAPER) dataset, for $\tau = 5$, SORT-EXTENSION was about 3-4 times faster than SEARCH-EXTENSION. This is because for each entity SEARCH-EXTENSION needs to compute its edit distances with large numbers of substrings, and neglects that many entities share common prefixes. SORT-EXTENSION shares the computation on the common prefixes for different entities, thus can improve the performance. Obviously the larger $\tau$, the higher overhead to compute the edit distances for every entity. As SORT-EXTENSION shares more duplicated computations on larger thresholds, SORT-EXTENSION significantly outperforms SEARCH-EXTENSION for larger thresholds.

### B. Evaluation on Partition Strategies

In this section, we compare the performance of different partition strategies. We first evaluated the partition time [11]. We implemented two methods (see Section V-B): (1) Dynamic Programming; (2) The weight pruning which used the weight of even partition as an upper bound to do effective pruning. Figure 6 shows the results. We can see that the weight pruning technique can significantly improve the partition time, even 1-2 orders of magnitude better than the dynamic programming. For example, on the PUBMED (AUTHOR + PAPER) dataset, for $\tau = 5$, dynamic programming took 60 seconds and the weight pruning technique can improve the time to 5 seconds. This is because we can prune many unnecessary entries in the matrix using the weight of even partition as an upper bound.

[11]The time for determining the weights is very small and usually less than 1 second.

Compared with the extraction time in Figure 5, for $\tau = 5$, the extraction time was larger than 2000 seconds, and the partition time was only 5 seconds. Thus we can select a better partition strategy to improve the performance of even partition.

Then we compared the overall performance of different partition methods. We implemented two partition methods as discussed in Section V-C. (1) EVEN: We used the even partition. (2) DICT+DOC: We used the dictionary and documents to determine the segment weights. Figure 7 shows the performance comparison. We can see that DICT+DOC outperformed EVEN and achieved higher performance. This is because DICT+DOC minimized the number of candidates and decreased the extension time significantly. Although EVEN did not need to partition entities on-the-fly, it may involve large numbers of candidates for large thresholds and thus may lead to low performance. Notice that the partition time was very small compared with the extraction time. For example, on the DBLP (TITLE + PAPER) dataset, for $\tau=10$, EVEN took 14,000 seconds and DICT+DOC improved the time to 3500 seconds.

In addition, we compared the number of candidates for different methods. Figure 8 shows the results. We can see that DICT+DOC had smaller numbers of candidates that EVEN. On the WIKI (URL + WEBPAGE) dataset, for $\tau = 1$, EVEN had 40 million candidates and DICT+DOC decreased it to 20 thousand. Thus DICT+DOC can improve the performance.

### C. Comparison with Existing Methods

In this section, we compare our algorithms TASTE-EVEN (using EVEN partition) and TASTE-DICT+DOC (using DICT+DOC partition) with state-of-the-art methods Faerie [18] and NGPP [26]. We tuned parameters $l_p$ for NGPP and $q$ for Faerie, and reported their best performance. The three methods constructed the dictionary index offline, and thus the indexing time was not included in the runtime. The indexing time of our even partition method was very small compared with the extraction time. For example, on the PubMed dataset with
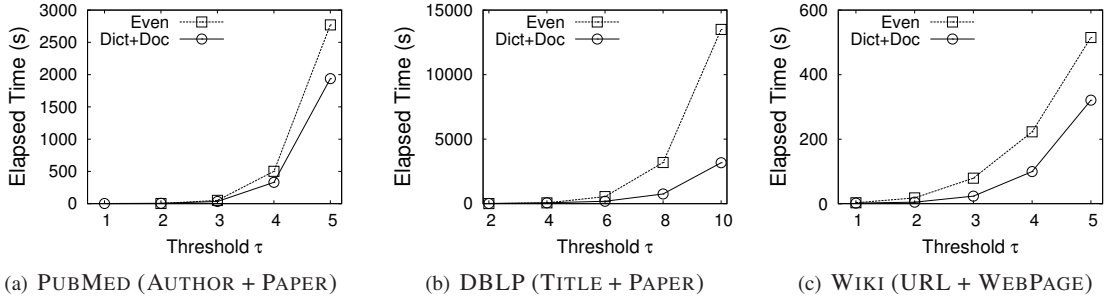
Fig. 7. Partition strategy selection: performance comparison of different partition strategies.

(a) PUBMED (AUTHOR + PAPER)     (b) DBLP (TITLE + PAPER)     (c) WIKI (URL + WEBPAGE)



(a) PUBMED (AUTHOR + PAPER)     (b) DBLP (TITLE + PAPER)     (c) WIKI (URL + WEBPAGE)
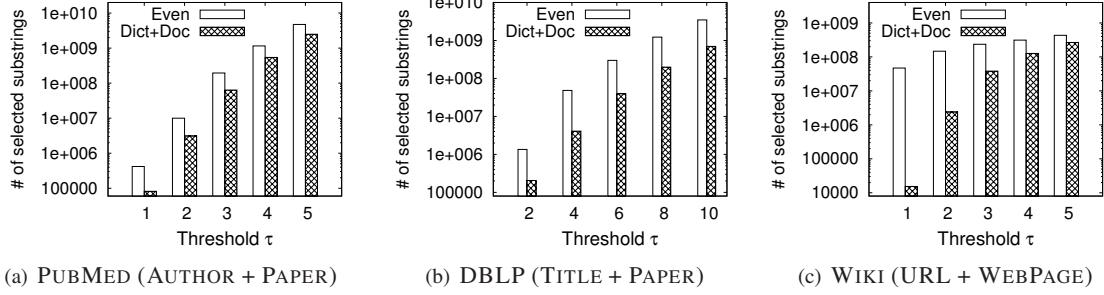
Fig. 8. Partition strategy selection: the numbers of candidates of different partition strategies.

$\tau = 3$, the indexing time was less than 1 second while the extraction time was about 60 seconds. The indexing time of our DICT+DOC partition method was included in the online time, since it partitioned entities on the fly. NGPP and TASTE have the limitation that they need to build different indexes for different $\tau$ values. Faerie has no such limitation.

Figure 9 shows the results. We see that our methods outperformed Faerie and NGPP significantly, especially for large thresholds. On the PUBMED (AUTHOR + PAPER) dataset for $\tau = 5$, on the DBLP (TITLE + PAPER) dataset for $\tau = 8, 10$, and on the WIKI (URL + WEBPAGE) for $\tau = 4, 5$, NGPP took more than 10 hours (36000 seconds). In the figures we did not show the numbers larger than 10 hours. On the DBLP (TITLE + PAPER) dataset for $\tau = 10$, Faerie also took more than 10 hours. Our method TASTE-DICT+DOC only took about 4000 seconds. On the WIKI (URL + WEBPAGE) dataset for $\tau = 4$, NGPP took more than 10 hours, Faerie took about 3000 seconds, and our method TASTE-DICT+DOC took less than 200 seconds.

Notice that although NGPP achieved high performance for small edit-distance thresholds, it is inefficient for large thresholds. The reason is that it needs to enumerate neighborhoods of entities and an entity has larger numbers of neighborhoods for larger thresholds ($l_p\tau^2$). Faerie also performed worse for large $\tau$ as it has low pruning power for large $\tau$, since it needs to select a smaller gram length for a larger threshold and a smaller gram length leads to lower pruning power.

We also compared index sizes of different methods. Table III shows the results. On DBLP (TITLE + PAPER), for $\tau = 4$, the index size of NGPP was 252 MB. Faerie involved 51.8 MB for $q = 2$. TASTE-EVEN took about 23.1 MB as it only needed to maintain a trie structure and inverted lists. TASTE-DICT+DOC needed to maintain a suffix trie, thus its index increased to 221 MB. Note that TASTE-DICT+DOC can improve the performance against TASTE-EVEN. Thus if a user prefers high performance and has a large memory, she can use TASTE-

DICT+DOC; otherwise she can use TASTE-EVEN.

NGPP had different index sizes for different edit-distance thresholds, as NGPP needs to use $\tau$ to generate neighborhoods. The larger the thresholds, the larger numbers of neighbors of an entity, and thus the larger indexes. Faerie had different sizes for different $q$ values as the number of $q$-grams of an entity is $l - |q| + 1$ where $l$ is the entity length. Note that the index sizes of Faerie are independent on $\tau$.

The inverted-list sizes of our method are linear with the increase of threshold $\tau$, as we need to keep $\tau + 1$ segments for every entity. While NGPP is quadric, as it needs to maintain $l_p\tau^2$ substrings. The trie size of our method depends on the number of distinct keywords in the datasets which is usually smaller than the dataset sizes.

TABLE III
INDEX SIZES.

| Algorithms | Index Sizes (MB) | | |
|---|---|---|---|
| | PUBMED | DBLP | WIKI |
| NGPP ($\tau = 4$) | 252 | 575 | 203 |
| Faerie | 51.8 ($q$=2) | 125 ($q$=7) | 48 ($q$=4) |
| TASTE-EVEN ($\tau = 4$) | 23.1 | 26.4 | 13.4 |
| TASTE-DICT+DOC ($\tau = 4$) | 221 | 364 | 187 |

### D. Scalability

This section evaluates the scalability of our algorithm TASTE-DICT+DOC. We varied the number of entities in the dictionary. Figure 10 shows the results on the three datasets. We observe that our method scaled well as the dictionary size increased. For example, on the PUBMED (AUTHOR + PAPER) dataset, for $\tau = 5$, our method took 600 seconds for 50k entities and 2000 seconds for 300k entities.

### VII. CONCLUSION

In this paper, we have studied the problem of approximate entity extraction with edit-distance constraints. We partition entities into different segments and use a trie structure to index the segments. We traverse the trie structure to identify the answers and propose a search-and-extension based method to
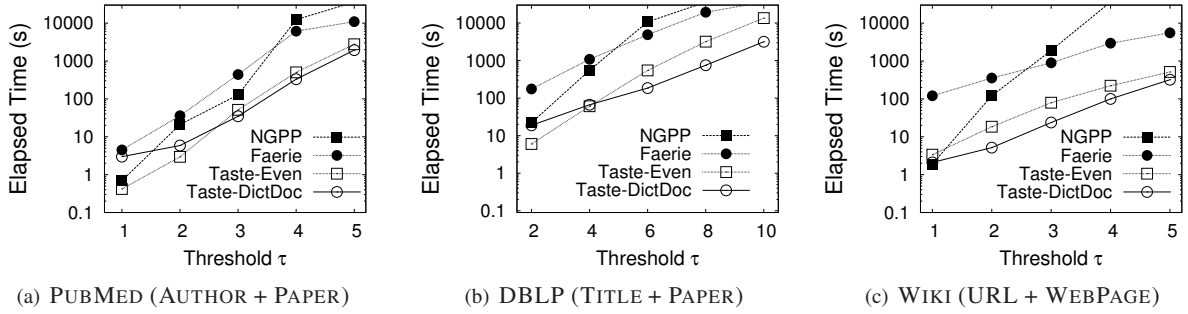
(a) PUBMED (AUTHOR + PAPER)    (b) DBLP (TITLE + PAPER)    (c) WIKI (URL + WEBPAGE)

Fig. 9.    Performance comparison with state-of-the-art studies.



(a) PUBMED (AUTHOR + PAPER)    (b) DBLP (TITLE + PAPER)    (c) WIKI (URL + WEBPAGE)
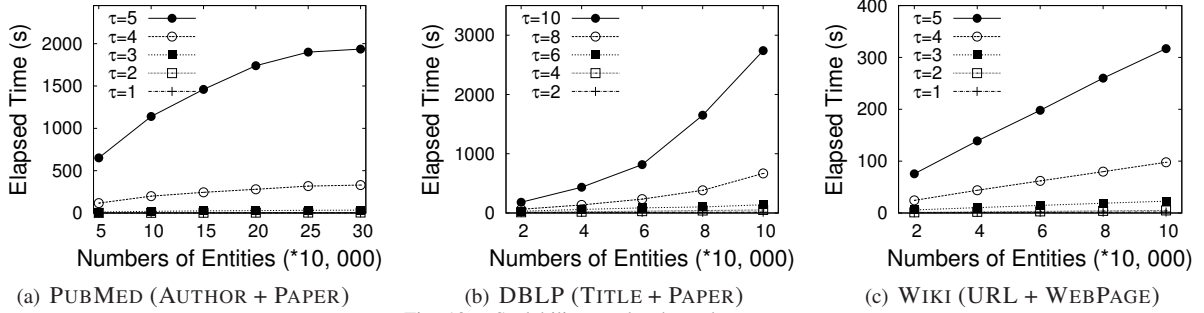
Fig. 10.    Scalability on the three datasets.

efficiently find similar entities. We develop sort-based pruning techniques and algorithms to improve the performance. We optimize the partition scheme to select the best partition strategy in order to achieve high performance. Experimental results show that our method achieves higher performance and significantly outperforms state-of-the-art approaches.

## VIII. ACKNOWLEDGEMENT

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

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

[12] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[13] 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.

[14] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

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

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

[17] 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.

[18] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD Conference*, pages 529–540, 2011.

[19] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. In *PVLDB*, 2012.

[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] H. Shang and T. H. Merrett. Tries for approximate string matching. *IEEE Trans. Knowl. Data Eng.*, 8(4):540–547, 1996.

[22] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[23] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[24] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[25] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

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

[27] 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.

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

[29] X. Yang, B. Wang, and C. Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *SIGMOD Conference*, pages 353–364, 2008.