An Efficient Sliding Window Approach for Approximate Entity Extraction with Synonyms

Jin Wang
University of California, Los Angeles
jinwang@cs.ucla.edu

Chunbin Lin
Amazon AWS
lichunbi@amazon.com

Mingda Li
University of California, Los Angeles
limingda@cs.ucla.edu

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

ABSTRACT

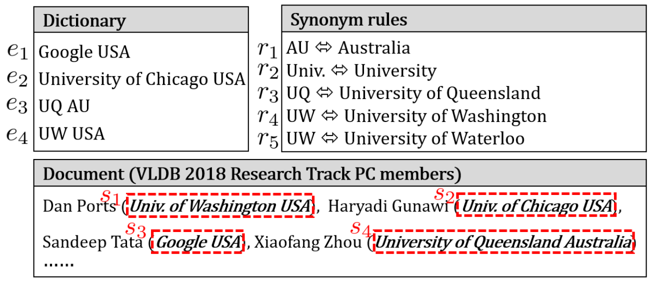
Dictionary-based entity extraction from documents is an important task for several real applications. To improve the effectiveness of extraction, many previous studies focused on the problem of approximate dictionary-based entity extraction, which aims at finding all substrings in documents that are similar to pre-defined entities in the reference entity table. However, these studies only consider syntactical similarity metrics, such as Jaccard and edit distance. In the real-world scenarios, there are many cases where syntactically different strings can express the same meaning. For example, *MIT* and *Massachusetts Institute of Technology* refer to the same object but they have a very low value of syntactic similarity. Existing approximate entity extraction work fails to identify such kind of semantic similarity and will definitely suffer from low recall.

In this paper, we come up with the new problem of approximate dictionary-based entity extraction with synonyms and propose an end-to-end framework Aeetes to solve it. We propose a new similarity measure Asymmetric Rule-based Jaccard (JACCAR) to combine the synonym rules with syntactic similarity metrics and capture the semantic similarity expressed in the synonyms. We propose a clustered index structure and several pruning techniques to reduce the filter cost so as to improve the overall performance. Experimental results on three real world datasets demonstrate the effectiveness of Aeetes. Besides, Aeetes also achieves high performance in efficiency and outperforms the state-of-the-art method by one to two orders of magnitude.

1 INTRODUCTION

Dictionary-based entity extraction [11] identifies all substrings from a document that match predefined entities in a reference entity table i.e. the dictionary. Compared with other kinds information extraction approaches, such as rule-based, machine learning and hybrid ones, dictionary-based entity extraction is good at utilizing extra domain knowledge encoded in the dictionary [24]. Therefore, it has been widely adopted in many real world applications that required Named entity recognition (NER), such as *academic search*, *document classification*, and *code auto-debugging*.

A typical application scenario is the *product analysis and reporting system* [10]. These systems maintain a list of well-defined products and require to find the mentions of product names in the online acquired documents. More precisely, these systems receive many consumer reviews, then they extract the substrings that mentioned reference product names from those reviews. Such mentions of referenced entities serve as a crucial signal for further analyzing the review documents, such sentiment analysis, opinion mining and recommendation. High-quality extraction of such mentions will significantly improve the effectiveness of these systems. Furthermore, the large volume of documents such systems receive turns improving the efficiency of extraction into a critical requirement.



	Dictionary
e_1	Google USA
e_2	University of Chicago USA
e_3	UQ AU
e_4	UW USA

	Synonym rules
r_1	AU ⇔ Australia
r_2	Univ. ⇔ University
r_3	UQ ⇔ University of Queensland
r_4	UW ⇔ University of Washington
r_5	UW ⇔ University of Waterloo

Document (VLDB 2018 Research Track PC members)

Dan Ports (s_1 *Univ. of Washington USA*), Haryadi Gunawi (s_2 *Univ. of Chicago USA*), Sandeep Tata (s_3 *Google USA*), Xiaofang Zhou (s_4 *University of Queensland Australia*) ......

Figure 1: Example of institution name extraction.

To provide high-quality entity extraction results and improve the recall, some prior work [12, 13, 35] studied the problem of Approximate dictionary-based Entity Extraction (AEE). As entities in the dictionary are represented as strings, they employ syntactic similarity functions (e.g. Jaccard and Edit Distance) to measure the similarities between entities and substrings from a document. The goal is to find not only the exactly matched substrings but also those *similar* to entities in the dictionary.

Though prior work has achieved significant degree of success in identifying the syntactic similar substrings from documents, they would still miss some substrings that are semantically similar to entities. In many cases, syntactically different strings can have very close semantic meaning. For example, consider a substring *"Mitochondrial Disease"* in a biomedical document and an entity *"Oxidative Phosphorylation Deficiency"* in the dictionary. Prior studies on AEE problems [13, 35] would fail in identifying this substring since they have very low similarity score under any syntactic similarity metric. However, *"Mitochondrial Disease"* and *"Oxidative Phosphorylation Deficiency"* actually refer to the same disease and they are expected to be included in the result. Therefore, it is necessary to propose a new framework to take both syntactic similarity and semantics carried by synonyms into consideration.

We can capture such synonyms by applying synonym rules on the basis of syntactic similarity. A `synonym rule` *r* is a pair of strings with the form ⟨ *lhs*⇔ *rhs*⟩ that express the same semantics. Here both `lhs` and `rhs` are token sequences. For example ⟨ *Big Apple*⇔ *New York*⟩ is a synonym rule as *"Big Apple"* is actually a

© 2019 Copyright held by the owner/author(s). Published in Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), March 26-29, 2019, ISBN 978-3-89318-081-3 on OpenProceedings.org.

nickname for *"New York"*. Example 1.1 shows a real-life scenario demonstrating the effectiveness of applying synonym rules in entity extraction.

Example 1.1. Figure 1 provides an example document which contains PVLDB 2018 PC members. The dictionary includes a list of institution names, and the synonym rule table contains a list of synonym rules. The exact match approach only finds s_3 as it is the only one having an exact match in the dictionary. AEE based approaches like Faerie [13] can find s_3 and s_2 but misses s_1 and s_4. By applying rules, we can find all similar results s_1, s_2, s_3 and s_4.

As motivated above, applying synonym rules can significantly improve the effectiveness of approximate entity extraction. In this paper, we formally introduce the problem of Approximate Entity Extraction with Synonym (AEES) from dictionaries.

Though the application of synonym rules could improve effectiveness, it also brings significant challenges in computational performance. To address this issue, we study and propose new solutions for the AEES problem, along with techniques that optimize the performance. In fact, we propose an end-to-end framework called **A**pproximat**e** Dictionary-based **Ent**i**t**y **E**xtraction with **S**ynonyms (Aeetes) that effectively and efficiently solves the AEES problem. We first propose a new similarity metric Asymmetric Rule-based Jaccard (JACCAR) to evaluate the similarity between substrings in documents and entities in the dictionary by considering both syntactic similarity and semantic relevance brought by synonyms. By properly designing the similarity measure, we can reduce the overhead of applying the synonym rules and capture the rich semantics at the same time. To support efficient extraction, we devise a clustered inverted index structure which enables skipping dissimilar entities when traversing the index. We also apply efficient sliding-window based pruning techniques to accelerate the filtering process by leveraging the overlaps between adjacent substrings in the document. We evaluate our proposed methods on three popular datasets with real application scenarios. Experimental results demonstrate the superiority of our method in both effectiveness and efficiency.

To summarize, we make the following contributions.

- We identify and formally define a new problem dictionary-based Approximate Entity Extraction from documents with Synonyms. And we propose an end-to-end framework Aeetes to efficiently address the problem.
- We devise a clustered index structure and several pruning techniques to improve the performance. Specifically, we proposed a dynamic prefix maintenance algorithm and a lazy candidate generation method to take advantage of the shared computation between substrings in a document so as to reduce the filter cost.
- We conduct an extensive empirical evaluation on three real-world datasets to evaluate the efficiency and effectiveness of the proposed algorithms. Experimental results demonstrate the effectiveness of our method. In addition, our method also achieved good efficiency: it outperforms the baseline methods by one to two orders of magnitude in extraction time.

The rest of this paper is organized as follows. We formalize the problem of AEES and introduce the overall framework in Section 2. We propose a clustered inverted index structure in Section 3. We devise the sliding window based filter techniques in Section 4. We make necessary discussions about some important issues in Section 5. The experimental results are reported in Section 6. We summarize the related work in Section 7. Finally, conclusions are made in Section 8.

2 PRELIMINARY

In this section, we first define some basic terminology to describe our work (Section 2.1). We then formulate the AEES problem and justify its definition (Section 2.2). Finally we provide an overview of our framework (Section 2.3).

2.1 Basic Terminology

Entity. An `entity` e is modeled as a token sequence, i.e, $e = e[1], ..., e[|e|]$ where $|e|$ is the number of tokens in e. For example, the entity e_2 =*"Purdue University USA"* in Figure 1 has three tokens and $e_2[3]$ = *"USA"*. We use $e[i, j]$ to denote the subsequence of tokens $e[i], ..., e[j]$ of e.

Applicable synonym rule. Given an entity e, a synonym rule r is an `applicable rule` for e if either *lhs* or *rhs* is a subsequence of e. In some cases, if two applicable rules have overlapping tokens and cannot be applied simultaneously, we call them `conflict rules`. For example, r_4 and r_5 are conflict rules as they have an overlapping token "UW". In order to generate derived entities, we need to obtain the optimal set of non-conflict rules, which includes all possible combinations. Unfortunately, finding the optimal set of non-conflict rules requires exponential time. To improve the performance, we propose a greedy algorithm to select the set of non-conflict rules whose cardinality is as large as possible (details in Section 5). We use $\mathcal{A}(e)$ to denote the sets of non-conflict rules of the entity e.

Derived entity. Given an entity e and one applicable rule r ($\langle$ *lhs*$\Leftrightarrow$ *rhs*$\rangle$), without the loss of generality, we assume `lhs` is a subsequence of e. Applying r to e means replacing the `lhs` in the subsequence of e with *rhs* in r. The i^{th} new generated entity e^i is called `derived entity` of e. And e is called `origin entity` of e^i. And the set of all derived entities of e is denoted as $\mathcal{D}(e)$.

According to the previous study [3], we get a derived entity e^i of e by applying rules in a subset of $\mathcal{A}(e)$. In this process, each original token is rewritten by at most one rule [1]. Similar to previous studies [3, 29], different combination of rules in $\mathcal{A}(e)$ will result in different different derived entities. Following this routine, we can get $\mathcal{D}(e)$ by enumerating the combination of applicable rules. The cardinality of $|\mathcal{D}(e)|$ is $O(2^n)$ where $|\mathcal{A}(e)| = n$.

Consider the example data in Figure 1 again. For the entity e_3=*"UQ AU"* in the dictionary, the applicable rules $\mathcal{A}(e_3) = \{r_1, r_3\}$. Thus, $\mathcal{D}(e_4)$ can be calculated as following: { *"UQ AU"*, *"University of Queensland AU"*, *"UQ Australia"*, *"University of Queensland Australia"*}.

For a dictionary of entities $\mathcal{E}_0$, we can generate the `derived dictionary` $\mathcal{E} = \bigcup_{e \in \mathcal{E}_0} \mathcal{D}(e)$.

2.2 Problem Formulation

For the problem of approximate string join with synonyms (ASJS), previous studies have already defined some synonym-based similarity metrics, such as *JaccT* [3], *SExpand* [19] and *pkduck* [29]. In the problem setting of ASJS, we know the threshold in off-line step and need to deal with synonym rules and two collections of strings in the on-line step. So the above similarity metrics apply

[1] As shown in [3], if a new generated token is allowed to apply rules again, then it becomes a non-deterministic problem.

the synonym rules on both strings during the join processing. Suppose the lengths of two strings involved in join is S_1 and S_2, then the search space for joining the two strings is $O(2^{S_1} \cdot 2^{S_2})$

However, for our AEES problem, we obtain the entity dictionary and synonym rules in the off-line step but need to deal with the documents and threshold in the on-line step. Moreover, the length of a document will be much larger than that of entities. Unlike ASJS which computes the similarity between two strings, AEES aims at identifying `substrings` from a document that are similar to entities. Therefore, applying rules onto documents in the on-line step will be too expensive for the AEES problem. Suppose the size of a document is D and the length of an entity is e, if we directly use the similarity metrics of ASJS problem, the search space for extracting e would be $O(2^D \cdot 2^e)$. While r and s will always be similar in ASJS problem, in AEES problem the value of D is usually 10-20 times larger than e. Therefore such a complexity is not acceptable.

Based on the above discussion, we devise our asymmetric similarity metric JACCAR (for Asymmetric Rule-based Jaccard). Unlike previous studies on the ASJS problem, we only apply the synonym rules on the entities to generate derived entities in the off-line step. In the on-line extraction step, instead of applying rules on substrings in the document, we just compute the similarity between the substrings and all derived entities which have been generated in the off-line step. Here we use Jaccard to evaluate the syntactic similarity. Our techniques can also be easily extended to other similarity metrics, such as Overlap, Cosine and Dice. To verify the JACCAR value between an entity e and a substring s from the document, we first find $\mathcal{A}(e)$ and generate all derived entities of e. Then for each $e^i \in \mathcal{D}(e)$, we calculate the value of JAC (e^i, s). Finally we select the maximum JAC (e^i, s) as the value of JACCAR (e,s). The detailed definition is formalized in Definition 2.1.

Definition 2.1 (Asymmetric Rule-based Jaccard). Given an entity e in the dictionary and a substring s in the document, let $\mathcal{D}(e)$ be the full set of derived entities of e by applying rules in $\mathcal{A}(e)$. Then JACCAR(e, s)) is computed as follows:

$$\mathsf{ARJ}(e,s) = \max_{e^i \in \mathcal{D}(e)} Jaccard(e^i, s) \qquad (1)$$

We want to highlight that the main difference between previous synonym-based similarity metrics for ASJS and JACCAR is that *previous approaches apply synonyms on both records that are involved into the join process; while* JaccAR *only applies synonyms on the entities in the dictionary*. Recall the above time complexity, by using JACCAR instead of similarity metrics for ASJS problem, we can reduce the time complexity from $O(2^D \cdot 2^e)$ to $O(D \cdot 2^e)$. The intuition behind JACCAR is that some rules have the same lhs/rhs, which might lead to potentially dissimilar derived entities. In order to identify a similar substring, we should focus on the derived entity that is generated by the set of synonym rules that is related to the context of substrings. By selecting the derived entity with the largest syntactic similarity, we can reach the goal of using the proper set of synonym rules to extract similar substrings from a document. JACCAR can achieve such a goal by avoiding the synonym rules which would decrease the similarity and applying those which increases the similarity.

Using the definition of Asymmetric Rule-based Jaccard, we can now characterize the AEES problem by Definition 2.2 below. Following previous studies of ASJS [3, 19], it is safe to assume that the set of synonym rules are given ahead of time. As many studies can effectively discover synonyms [2], our work can be seamlessly integrated with them. Although a recent study [29] supports discovering rules from dataset, it can only deal with abbreviations while our work here needs to support the general case of synonyms.

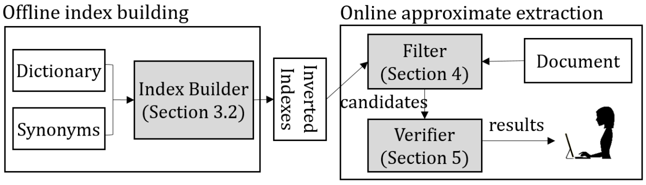


Figure 2: Architecture of Aeetes.

Definition 2.2 (AEES). Given a dictionary of entities $\mathcal{E}_0$, a set of synonym rules $\mathcal{R}$, a document d, and a threshold of Asymmetric Rule-based Jaccard τ, the goal of AEES is to return all the (e, s) pairs where s is a substring of d and $e \in \mathcal{E}_0$ such that JACCAR$(e, s) \geq \tau$.

Consider the dataset in Figure 1 again. Assume the threshold value is 0.9, then AEES returns the following pairs as results: (e_2, s_2), (e_1, s_3), (e_3, s_4). The JACCAR scores of the above three pairs are all 1.0.

2.3 Overview of Framework

As shown in Figure 2, Aeetes is an end-to-end framework which consists of two stages: off-line preprocessing and on-line extraction. The whole process is displayed in Algorithm 1. In the off-line preprocessing stage, we first find applicable synonym rules for each entity in the dictionary. Then we apply them to entities and generate the derived dictionary (line: 3). Next we create a clustered inverted index for the derived entities, which will be explained later in Section 3 (line: 4).

In the on-line extraction stage, we have a similarity threshold τ and a document d as input, and the goal is to extract all similar substrings from d. To this end, we propose a filter-and-verification strategy. In the filter step, if a derived entity e^i is similar to a substring $s \in d$, we will regard its corresponding origin entity e as the candidate of s (line: 5). In this way, we can adopt the filter techniques of Jaccard to get candidates for JACCAR. We propose effective pruning techniques in Section 4 to collect such candidates. In the verification phase, we verify the real value of JACCAR for all candidates (lines: 6-9).

3 INDEX CONSTRUCTION

In this section, we first review the length and prefix filter techniques, which serves as the cornerstone of our approaches (Section 3.1). Then we devise a clustered inverted index to facilitate the filter techniques (Section 3.2).

3.1 Filtering Techniques Revisit

In order to improve the performance of overall framework, we need to employ effective filtering techniques. As the length of a document is much larger than that of an entity, we should be able to exactly locate mentions of entities in the documents and avoid enumerating dissimilar candidates. To describe the candidate substrings obtained from the document, we use the following terminologies in this paper. Given a document d, we denote a

[2] These are introduced in Section 7, whereas generalizations of this approach are further discussed in Section 5.

Algorithm 1: Aeetes ($\mathcal{E}_0$, $\mathcal{R}$, d, τ)

```
Input: $\mathcal{E}_0$: The dictionary of entities; $\mathcal{R}$: The set of synonym
       rules; $d$: The given Docuemnt; $\tau$: The threshold of
       Asymmetric Rule-based Jaccard
Output: $\mathcal{H} = \{< e, s >| e \in \mathcal{E} \wedge s \in d \wedge \mathrm{JACCAR}(e, s) \geq \tau\}$
begin
    Initialize $\mathcal{H}$ as $\emptyset$;
    Generate a derived dictionary $\mathcal{E}$ using $\mathcal{E}_0$ and $\mathcal{R}$;
    Construct the inverted index for all derived entities in $\mathcal{E}$;
    Traverse substrings $s \in d$ and the inverted index, generate
    a candidate set $C$ of $< s, e >$ pairs;
    for each pair $< s, e > \in C$ do
        Verify the value of $\mathrm{JACCAR}(s, e)$;
        if $\mathrm{JaccAR}(s, e) \geq \tau$ then
            Add $< s, e >$ into $\mathcal{H}$;
    return $\mathcal{H}$;
end
```

substring with start position p and length l as $\mathcal{W}_p^l$. A token $t \in d$ is a valid token if there exists a derived entity $e_i^j \in \mathcal{E}$ containing t. Otherwise it is an invalid token. We call a group of substrings with the same start position p in the document as a window, denoted as $\mathcal{W}_p$. Suppose the maximum and minimum length of substrings in $\mathcal{W}_p$ is l_{max} and l_{min} respectively, this window can further be denoted as $\mathcal{W}_p(l_{min}, l_{max})$.

One primary goal of the filter step is to prune dissimilar candidates. To this end, we employ the state-of-the-art filtering techniques: length filter [25] and prefix filter [9].

Length filter. The basic idea is that if two strings have a large difference between their lengths, they cannot be similar. Specifically, given two strings e, s and a threshold τ, if $|s| < \lfloor |e| * \tau \rfloor$ or $|s| > \lceil \frac{|e|}{\tau} \rceil$, then we have $\mathrm{JAC}(s, e) < \tau$.

Suppose in the derived dictionary $\mathcal{E}$, the minimum and maximum lengths of derived entities are denoted as $|e|_\perp = \min\{|e| \mid e \in \mathcal{E}\}$ and $|e|_\top = \max\{|e| \mid e \in \mathcal{E}\}$, respectively. Then given a threshold τ, we can safely claim that only the substring $s \in d$ whose length $|s|$ is within the range $[\mathcal{E}_\perp\ \mathcal{E}_\top]$ could be similar to the entities in the dictionary where $\mathcal{E}_\perp = \lfloor |e|_\perp * \tau \rfloor$, $\mathcal{E}_\top = \lceil \frac{|e|_\top}{\tau} \rceil$.

Prefix filter. It first fixes a global order $\mathcal{O}$ for all tokens from the dataset (details in next subsection). Then for a string s, we sort all its tokens according to $\mathcal{O}$ and use $\mathcal{P}_\tau^s$ to denote the τ-prefix of string s. Specifically, for Jaccard similarity, we can filter out dissimilar strings using Lemma 3.1.

LEMMA 3.1 (PREFIX FILTER [9]). *Given two strings e, s and a threshold τ, the length of $\mathcal{P}_\tau^s$ ($\mathcal{P}_\tau^e$) is $\lfloor (1-\tau)|s| + 1 \rfloor$ ($\lfloor (1-\tau)|e| + 1 \rfloor$). If $\mathcal{P}_\tau^s \cap \mathcal{P}_\tau^e = \emptyset$, then we have* $\mathrm{JAC}(s, e) < \tau$.

3.2 Index structure

With the help of length filter and prefix filter, we can check quickly whether a substring $s \in d$ is similar to an entity $e \in \mathcal{E}_0$. However, enumerating s with all the entities one by one is time consuming due to the huge number of derived entities.

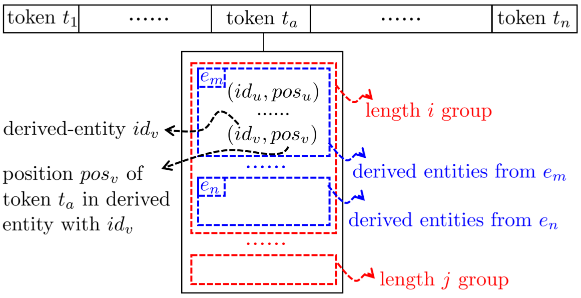
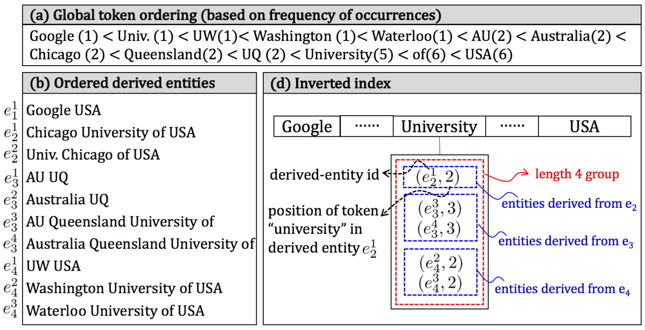


Figure 3: Index structure.



(a) Global token ordering (based on frequency of occurrences)

Google (1) < Univ. (1) < UW(1)< Washington (1)< Waterloo(1) < AU(2) < Australia(2) < Chicago (2) < Queensland(2) < UQ (2) < University(5) < of(6) < USA(6)

(b) Ordered derived entities

e_1^1	Google USA
e_2^1	Chicago University of USA
e_2^2	Univ. Chicago of USA
e_3^1	AU UQ
e_3^2	Australia UQ
e_3^3	AU Queensland University of
e_3^4	Australia Queensland University of
e_4^1	UW USA
e_4^2	Washington University of USA
e_4^3	Waterloo University of USA

(d) Inverted index

Google
......
University
......
USA
derived-entity id
$(e_2^1, 2)$
$(e_3^3, 3)$
$(e_3^4, 3)$
$(e_4^2, 2)$
$(e_4^3, 2)$
position of token "university" in derived entity e_2^1
length 4 group
entities derived from e_2
entities derived from e_3
entities derived from e_4

Figure 4: Example of index structure.

To accelerate this process, we build a *clustered inverted index* for entities in the derived dictionary. The inverted index of t, denoted as $\mathcal{L}[t]$, is a list of (e^i, pos) pairs where e^i is the identifier of a derived entity containing the token t and pos is the position of t in the ordered derived entity. For all tokens in the derived entities, we assign a global order $\mathcal{O}$ among them. Then for one derived entity, we sort its tokens by the global order and pos is the position of t in e^i under this order. Here as is same with previous studies [5, 36], we use the ascending order of token frequency as the global order $\mathcal{O}$. It is natural to deal with invalid tokens: in the on-line extraction stage, if a token $t \in d$ is an invalid token, we will regard its frequency as 0. With the help of pos, we can prune dissimilar entities with the prefix filter.

According to a recent experimental survey [21], the main overhead in set-based similarity queries comes from the filter cost. To reduce such overhead caused by traversing inverted index, we can skip some dissimilar entries by leveraging length filter: in each $\mathcal{L}[t]$, we group all the (e^i, pos) pairs by the length $l = |e^i|$. And such a group of derived entities is denoted as $\mathcal{L}_l[t]$. Then when scanning $\mathcal{L}[t]$ for $t \in s$, if l and $|s|$ does not satisfy the condition of length filter, we can skip $\mathcal{L}_l[t]$ in batch to reduce the number of accessed entries in the inverted index.

In addition, by leveraging the relationship between origin and derived entities, we can further cluster $e^i \in \mathcal{D}(e)$ within each group $\mathcal{L}_l[t]$ according to their original entity. Here we denote the group of entries with origin entity e and length l as $\mathcal{L}_l^e$. When looking for candidate entities for a substring s, if a derived entity e^i is identified as a candidate, *we will regard its origin entity e rather than the derived entity itself* as the candidate of s. If an origin entity e has already been regarded as the candidate of s, we can skip $\mathcal{L}_l^e[t]$ in batch when traversing $\mathcal{L}[t]$ where $t \in s$. Figure 3 visualizes the index structure.

Example 3.2. To have a better understanding of the index structure, we show the index (in Figure 4) built for the example data in Figure 1. As we can see, the token t="*University*" appears in five derived entities e_2^1, e_3^3, e_3^4, e_4^2, and e_4^3. And the positions of "*University*" in the corresponding ordered derived entities are 2, 3, 3, 2, and 2. Therefore, we store five (id, pos) pairs, i.e., $(e_2^1, 2)$, $(e_3^3, 3)$, $(e_3^4, 3)$, $(e_4^2, 2)$ and $(e_4^3, 2)$, in the inverted list $\mathcal{L}[University]$. The five pairs are organized into three groups (see the blue boxes) based on their original entities. For instance, $(e_3^3, 3)$ and $(e_3^4, 3)$ are

Algorithm 2: Index Construction($\mathcal{E}_0$, $\mathcal{R}$)

```
   Input: $\mathcal{E}_0$: The dictionary of entities; $\mathcal{R}$: The set of synonym
          rules.
   Output: $\mathcal{CI}$: The Clustered Inverted Index of entities
1  begin
2      Generate a derived dictionary $\mathcal{E}$ using $\mathcal{E}_0$ and $\mathcal{R}$;
3      Initilize $\mathcal{CI} = \emptyset$ and obtain the global order $\mathcal{O}$;
4      foreach derived entitiy $e' \in \mathcal{E}$ do
5          $l \leftarrow |e'|$, $e \leftarrow$ origin entity of $e'$;
6          foreach token $t \in e'$ do
7              Add the pair $(e', pos)$ into the corresponding
               group in inverted list $\mathcal{L}_l^e[t]$;
8      foreach $\mathcal{L}[t]$ do
9          $\mathcal{CI} = \mathcal{CI} \cup \mathcal{L}[t]$
10     return $\mathcal{CI}$;
11 end
```

grouped together as both e_3^3 and e_3^4 derived entities are from the same original entity e_3. In addition, they are further clustered into a group based on their lengths (see the red boxes). In this example, these five pairs are grouped into a length-4 group as the length of the derived entities are 4.

Algorithm 2 gives the details of constructing an inverted index. It first applies synonyms to the entities in the dictionary to get a derived dictionary (line 2). Then for each token t in the derived entities, it stores a list of (e', pos) pairs where e' is the identifier of a derived entity containing t and pos is the position of t in this derived entity according to the global order $\mathcal{O}$ (line: 4-7). The (e', pos) pairs in each list are organized into groups based on the length l and their corresponding origin entity e (line 5). Finally, we aggregate all the inverted lists and get the clustered index (line: 9).

4 SLIDING WINDOW BASED FILTERING

Based on the discussion in Section 3, we can come up with a straightforward solution for the AEES problem: we slide the window $\mathcal{W}_p(\mathcal{E}_\perp, \mathcal{E}_\top) \in d$ from the beginning position of document d. For each window, we enumerate the substrings and obtain the prefix of each substring. Next, we recognize the valid tokens from the prefix for each substring and scan the corresponding inverted lists to obtain the candidates. Finally, we verify the candidates and return all truly similar pairs.

Although we can prune many dissimilar substrings by directly applying length filter and prefix filter with the help of inverted index, the straightforward method needs to compute the prefix for a large number of substrings. Thus it would lead to low performance. In this section, we propose a sliding window based filtering mechanism to efficiently collect the candidates from a given document. To improve the overall efficiency, we devise effective techniques based on the idea of sharing computations between substrings and windows. We first devise a dynamic (incremental) prefix computation technique to take advantage of the overlaps between adjacent substrings and windows in Section 4.1. Next we further propose a lazy strategy to avoid redundant visits on the inverted index in Section 4.2.

4.1 Dynamic Prefix Computation by Shared Computation

We have the following observations w.r.t the windows and substrings. On the one hand, for two substrings in the same window with different length i.e. $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_p^{l_j}(\mathcal{E}_\perp \le l_i < l_j \le \mathcal{E}_\top)$, they share l_i common tokens. On the other hand, two adjacent windows $\mathcal{W}_p(\mathcal{E}_\perp, \mathcal{E}_\top)$ and $\mathcal{W}_{p+1}(\mathcal{E}_\perp, \mathcal{E}_\top)$ share $\mathcal{E}_\top - 1$ common tokens. This is very likely that there is a large portion of common tokens between the prefixes of $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_p^{l_j}$ and those of $\mathcal{W}_p^{l_i}$ and $\mathcal{W}_{p+1}^{l_i}$.

Motivated by these observations, we can improve the performance of the straightforward solution by dynamically computing the prefix for substrings in the document. Here we use $\mathcal{P}_\tau^{p,l}$ to denote the set of tokens of the τ-prefix(i.e., prefix of length $\lfloor (1-\tau) * l + 1 \rfloor$) of substring $\mathcal{W}_p^l$. Then we can obtain the prefix of one substring by utilizing that of a previous one. Specifically, for a given window $\mathcal{W}_p$, we first directly obtain $\mathcal{P}_\tau^{p,0}$ and then incrementally compute $\mathcal{P}_\tau^{p,l}$ on the basis of $\mathcal{P}_\tau^{p,l-1}$. Then for each substring $\mathcal{W}_p^l \in \mathcal{W}_p$, we scan the inverted lists of the valid tokens and collect the candidate entities. Similarly, for a substring $\mathcal{W}_{p+1}^l \in \mathcal{W}_{p+1}$, we can obtain its prefix $\mathcal{P}_\tau^{p+1,l}$ from $\mathcal{P}_\tau^{p,l}$. Then we can collect the candidate entities for each substring in $\mathcal{W}_{p+1}$ with the same way above.

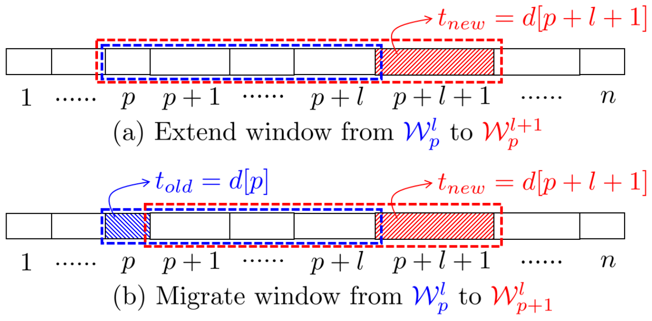
To reach this goal, we propose two operations Window Extend and Window Migrate to dynamically compute the prefix of substrings and collect candidate pairs for the above two scenarios.

Window Extend This operation allows us to obtain $\mathcal{P}_\tau^{p,l+1}$ from $\mathcal{P}_\tau^{p,l}$. Figure 5(a) gives an example of extending the window $\mathcal{W}_p^l$ to $\mathcal{W}_p^{l+1}$. As shown, when performing Window Extend, the length of the substring increases by 1. In this case, the length of the τ-prefix of $\mathcal{W}_p^{l+1}$ (i.e. $\lfloor (1-\tau) * (l+1) + 1 \rfloor$) can either increase by 1 or stay the same compared with the length of the τ-prefix of $\mathcal{W}_p^l$ (i.e. $\lfloor (1-\tau) * l + 1 \rfloor$). Then we can perform maintenance on the prefix accordingly:

- If the length of τ-prefix stays the same, we need to check whether the newly added token $d[p+l+1]$ will replace a token in $\mathcal{P}_\tau^{p,l}$. If so, we need to replace a lowest ranked token $t \in \mathcal{W}_p^l$ with $d[p+l+1]$ in the new prefix. Otherwise, there is no change in the prefix i.e. $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l}$.
- If the length of τ-prefix increases by 1, then we need to discuss whether the newly added token $d[p+l+1]$ belongs to the new prefix $\mathcal{P}_\tau^{p,l+1}$. If so, we can just have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l} \cup d[p+l+1]$. Otherwise, we should find a token $t \in \mathcal{W}_p^l$ and $t \notin \mathcal{P}_\tau^{p,l}$ with the highest rank. Then we have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l} \cup t$.

Example 4.1. Assume $\tau = 0.8$, when extending window from $\mathcal{W}_3^3$ to $\mathcal{W}_3^4$ (see Figure 6(a)), $|\mathcal{P}_\tau^{3,3}| = \lfloor (1-0.8) * 3 + 1 \rfloor = 1$ and $|\mathcal{P}_\tau^{3,4}| = \lfloor (1-0.8) * 4 + 1 \rfloor = 1$. So the length of τ-prefix stays the same. $\mathcal{P}_\tau^{3,3} = \{t_4\}$ as t_4 has the highest rank in window $\mathcal{W}_3^3$. The rank of new token t_6 in window $\mathcal{W}_3^4$ is 18, so t_6 will not replace a token in $\mathcal{P}_\tau^{3,3}$, so $\mathcal{P}_\tau^{3,4} = \mathcal{P}_\tau^{3,3} = \{t_4\}$. If the rank of t_6 is 2 instead of 18, then t_6 will replace a token in $\mathcal{P}_\tau^{3,3}$. In this case, $\mathcal{P}_\tau^{3,4} = \mathcal{P}_\tau^{3,3} - \{t_4\} \cup \{t_6\} = \{t_6\}$.

Now let's see the example in Figure 6(b) where we extend window from $\mathcal{W}_3^4$ to $\mathcal{W}_3^5$. The length of $\mathcal{P}_\tau^{3,4}$ is $\lfloor (1-0.8) * 4 + 1 \rfloor = 1$ and $\mathcal{P}_\tau^{3,4} = \{t_4\}$. But the length of $\mathcal{P}_\tau^{3,5}$ now is $\lfloor (1-0.8) * 5 + 1 \rfloor = 2$. The newly added token t_7 with rank 2 is in $\mathcal{P}_\tau^{3,5}$, so $\mathcal{P}_\tau^{3,5} = \mathcal{P}_\tau^{3,4} \cup \{t_7\} = \{t_4, t_7\}$. If the rank of t_7 is 10 instead of 2, then t_7 should not be a token in $\mathcal{P}_\tau^{3,5}$. In this case,

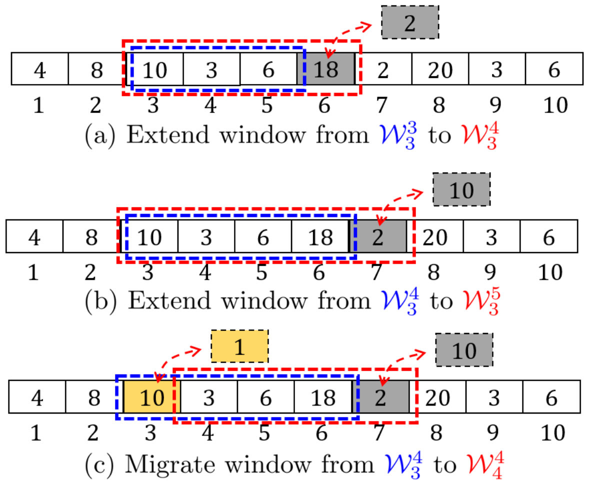


$t_{new} = d[p+l+1]$
1 ...... p $p+1$ ...... $p+l$ $p+l+1$ ...... n
$t_{old} = d[p]$
$t_{new} = d[p+l+1]$
1 ...... p $p+1$ ...... $p+l$ $p+l+1$ ...... n

(a) Extend window from $\mathcal{W}_p^l$ to $\mathcal{W}_p^{l+1}$

(b) Migrate window from $\mathcal{W}_p^l$ to $\mathcal{W}_{p+1}^l$

Figure 5: Example of window extend and window migrate.



(a) Extend window from $\mathcal{W}_3^3$ to $\mathcal{W}_3^4$

(b) Extend window from $\mathcal{W}_3^4$ to $\mathcal{W}_3^5$

(c) Migrate window from $\mathcal{W}_3^4$ to $\mathcal{W}_4^4$

Figure 6: Example of the Window Extend operator and the Window Migrate operator. Values in the boxes are the ranks of tokens according to a global order. Value k means the ranking of the token is top-k.

$\mathcal{P}_\tau^{3,5} = \mathcal{P}_\tau^{3,4} \cup \{t_5\} = \{t_4, t_5\}$ as t_5 has the highest rank, $t_5 \in \mathcal{W}_3^4$ and $t_5 \notin \mathcal{P}_\tau^{3,4}$.

Window Migrate This operation allows us to obtain the prefix of $\mathcal{W}_{p+1}^l$ from that of $\mathcal{W}_p^l$. Figure 5(b) shows an example of migrating the window $\mathcal{W}_p^l$ to $\mathcal{W}_{p+1}^l$. We can see that when performing Window Migrate, the length of substring will stay the same. In this case, for a substring $\mathcal{W}_{p+1}^l$, the token $t_{old} = d[p]$ will be removed and the token $t_{new} = d[p+1+l]$ will be inserted. Then we discuss the maintenance of prefix according to t_{old}: .

- If $t_{old} \notin \mathcal{P}_\tau^{p,l}$, it makes no influence on the prefix, we only need to check whether t_{new} will replace a token in $\mathcal{P}_\tau^{p,l}$ to form $\mathcal{P}_\tau^{p+1,l}$. If so, we need to replace the lowest ranked token $t \in \mathcal{W}_p^l$ with t_{new} in $\mathcal{P}_\tau^{p+1,l}$. Otherwise, we have $\mathcal{P}_\tau^{p,l+1} = \mathcal{P}_\tau^{p,l}$.
- If $t_{old} \in \mathcal{P}_\tau^{p,l}$, we still need to check whether t_{new} will appear in $\mathcal{P}_\tau^{p+1,l}$ in a similar way. If so, we need to replace t_{old} with t_{new} to generate $\mathcal{P}_\tau^{p+1,l}$; Otherwise, we need to replace t_{old} with a token t s.t. $t \in \mathcal{W}_p^l$ and $t \notin \mathcal{P}_\tau^{p,l}$ with the highest rank.

Example 4.2. Consider the case when migrating window from $\mathcal{W}_3^4$ to $\mathcal{W}_4^4$ (see Figure 6(c)), both $\mathcal{W}_3^4$ and $\mathcal{W}_4^4$ have the length $\lfloor(1-0.8)*3+1\rfloor = 1$ (assume $\tau = 0.8$). $\mathcal{P}_\tau^{3,4} = \{t_4\}$ as t_4 has the highest rank in window $\mathcal{W}_3^4$. After this migration, we have $t_{old} = t_3$ (with rank 10) and $t_{new} = t_7$ (with rank 2). Then we know $t_{old} \notin \mathcal{P}_\tau^{3,4}$ and t_{new} will replace a token in $\mathcal{P}_\tau^{3,4}$, thus we have $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} - \{t_4\} \cup \{t_7\} = \{t_7\}$. If the rank of t_7 is 10 rather than 2 (see the blocks with gray color), then t_{new} will not replace a token in $\mathcal{P}_\tau^{3,4}$ and $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} = \{t_4\}$.

Further, if the rank of t_3 is 1 instead of 10 (see the blocks with yellow color), then $\mathcal{P}_\tau^{3,4} = \{t_3\}$ as now t_3 has the highest rank in the window $\mathcal{W}_3^4$. Then we know $t_{old} \in \mathcal{P}_\tau^{3,4}$ and t_{new} (with rank 2) will occur in $\mathcal{P}_\tau^{4,4}$, so $\mathcal{P}_\tau^{4,4} = \mathcal{P}_\tau^{3,4} - \{t_{old}\} \cup \{t_{new}\} = \{t_7\}$. If the rank of t_7 is 10 rather than 2 (see the blocks with gray color), then we need to replace t_{old} with t_4 as t_4 has the highest rank such that $t_4 \in \mathcal{W}_3^4$ and $t_4 \notin \mathcal{P}_\tau^{3,4}$. Therefore, $\mathcal{P}_\tau^{4,4} = \{t_4\}$.

We show the steps of candidate generation with dynamic prefix computation in Algorithm 3. To implement the operations defined above, we need to use some helper functions. First, we define the function *ScanInvetedIndex* which takes the inverted index and $\mathcal{P}_\tau^{p,l}$ as input and return all the candidate entities of $\mathcal{W}_p^l$ by visiting the inverted indexes that are corresponding to valid tokens in $\mathcal{P}_\tau^{p,l}$. For a valid token $t \in \mathcal{P}_\tau^{p,l}$, we can obtain the candidates from the inverted index $\mathcal{L}[t]$. Note that since we have grouped all items in the inverted index by length, for a group $\mathcal{L}_{l_e}[t] \in \mathcal{L}[t]$, if l_e and l does not satisfy the length filter, we can skip $\mathcal{L}_{l_e}[t]$ in batch. Similarly, if the position of t is beyond the τ-prefix of a derived entity e^i, we can also discard e^i.

Then we devise the function *ExtCandGeneration* to support the Window Extend operation. It first derives the prefix of the current substring from the prefix of previous substring according to the above description; then it obtains the candidates for the current substring. Similarly, we also devise the *MigCandGeneration* function to support Window Migrate operation. Due to the space limitation, we omit their pseudo codes here.

```
Algorithm 3: Candidate Generation($\mathcal{CI}$, $d$, $\tau$)
   Input: $\mathcal{CI}$: The Inverted Index; $d$: The given Document; $\tau$:
          The threshold of Asymmetric Rule-based Jaccard
   Output: $C$: The set of candidate pairs
1  begin
2      Initialize $C \leftarrow \emptyset, p \leftarrow 1$;
3      Obtain the prefix $\mathcal{P}_\tau^{p,\mathcal{E}_\perp}$;
4      $C = C\cup$ ScanInvetedIndex($\mathcal{CI}, \mathcal{P}_\tau^{p,\mathcal{E}_\perp}$);
5      for $len \in [\mathcal{E}_\perp + 1, \mathcal{E}_\top]$ do
6          $C_p^{len}, \mathcal{P}_\tau^{p,len} \leftarrow$ ExtCandGeneration($\mathcal{P}_\tau^{p,len-1}$);
7          $C = C \cup C_{len}$;
8      while $p < |d| - \mathcal{E}_\perp$ do
9          if $C_{p-1} \neq \emptyset$ then
10             for $len \in [\mathcal{E}_\perp, \mathcal{E}_\top]$ do
11                 $C = C\cup$ MigCandGeneration($\mathcal{P}_\tau^{p-1,len}$);
12         else
13             Obtain the candidates of $C_p$ in the same way of
               line 6 to line 7;
14         $p \leftarrow p + 1$;
15     Perform Window Extend on the last window with length
       $|d| - \mathcal{E}_\perp + 1$ to collect candidates;
16     return $C$;
17 end
```

The whole process of the algorithm is as follows. We first initialize the candidate set C and start from the first position of the document (line: 2). Here we denote the candidates from window $\mathcal{W}_p$ as C_p. For the first window, we perform Window Extend and collect all the candidates of substrings $\mathcal{W}_0^l$ (line: 4-line: 7). Next we enumerate the start position of the window and look at the previous window. If the previous window has candidate substrings, we will perform Window Migrate on the previous window to obtain the prefix for each substring in the current window and then collect the candidates for each substring (line: 9). Otherwise, we will obtain the prefix of each substring with Window Extend (line: 12). Such processes are repeated until we reach the end of the document. Finally, we return all the candidates and send them to the verification step (line: 16).

Example 4.3. Consider the example data in Figure 1 again. We have $\mathcal{E}_\perp = 1$ and $\mathcal{E}_\top = 5$. Assume document d has 1000 tokens. The total number of the function calls of ExtCandGeneration and MigCandGeneration is $1000 \times (\mathcal{E}_\top - \mathcal{E}_\perp) = 4000$. Notice that, both ExtCandGeneration and MigCandGeneration compute the prefix incrementally. However, the straightforward method needs to compute the prefix for $500,500$ substrings, as it requires to enumerate all possible substrings and compute the prefix for each of them independently.

4.2 Lazy Candidate Generation

With the dynamic prefix computation method, we avoid obtaining the prefix of each substring from scratch. However, there is still room for improvement. We can see that in Algorithm 3, we need to traverse the inverted indexes and generate candidates for all the valid tokens after obtaining the prefix. As substrings within the same window could have a large overlap in their prefix, they could also share many valid tokens. Moreover, a valid token t is also likely to appear anywhere within the same document. Even if t belongs to disjoint substrings, the candidate entities are still from the same inverted index $\mathcal{L}(t)$.

To further utilize such overlaps, we come up with a lazy strategy for candidate generation. The basic idea is that after we compute the prefix of a substring, we will not traverse the inverted indexes to obtain candidates immediately. Instead, we just collect the valid tokens for each substring and construct a global set of valid tokens $\mathcal{T}$. Finally, we will postpone the visits to inverted indexes after we finish recognizing all the valid tokens and corresponding substrings. In this way, *for each valid token $t \in \mathcal{T}$, we only need to traverse its associated inverted index $\mathcal{L}_t$ once during the whole process of extraction for one document.*

The difficulty of implementing the lazy strategy is two-fold. The first problem, i.e. the one of large space overhead required is discussed next, whereas the second one, i.e. the one related to different substring lengths is discussed later. Since we do not collect candidates immediately for each substring, we need to maintain the valid tokens for each substring. As the number of substrings is rather large, there will be heavy space overhead. To solve this problem, we take advantage of the relationship between substrings within the same window. Here we denote the valid token set of a substring $\mathcal{W}_p^l$ as $\Phi_p(l)$. For one window $\mathcal{W}_p$, we only keep the full contents of $\Phi_p(\mathcal{E}_\perp)$. To obtain $\Phi_p(l), l > \mathcal{E}_\perp$, we utilize a light-weighted structure `delta valid token`, which is represented as a tuple $< t, \circ >$. Here t is the valid token that is different from the previous substring; $\circ$ is a symbol to denote the operation on t. If $\circ$ is $+$ $(-)$, it means we need to insert t into(remove t from) the previous valid token set. We denote the set of delta valid tokens of substring $\mathcal{W}_p^l$ as $\Delta\phi(l)$. And then we have:

$$\Phi_p(l+1) = \Phi_p(l) \biguplus \Delta\phi(l) \qquad (2)$$

where $\biguplus$ means applying all operations of the corresponding token in $\Delta\Phi(l)$ on the given valid token set. If $\Delta\phi(l) = \emptyset$, it means that $\Phi_p(l+1) = \Phi_p(l)$. Then we can obtain all valid tokens and the corresponding candidate substrings of window $\mathcal{W}_p$ as:

$$\Psi(p) = \bigcup_{l \in [\mathcal{E}_\perp, \mathcal{E}_\top]} < \mathcal{W}_p^l, \Phi_p(\mathcal{E}_\perp) \biguplus \Sigma_{i=\mathcal{E}_\perp}^{l} \Delta\phi_p(i) > \qquad (3)$$

Example 4.4. Consider the example document in Figure 6 again. Assume we have $\mathcal{E}_\perp = 1$, $\mathcal{E}_\top = 4$ and $\tau = 0.6$. $\Phi_3(1) = \{t_3\}$ as t_3 is the only valid token in $\mathcal{P}_\tau^{3,1}$. Then $\Phi_3(2) = \Phi_3(1) \uplus \{< t_3, - >, < t_4, + >\}$ as t_3 is not a valid token for $\mathcal{W}_3^2$ but t_4 is. Similarly, we have $\Phi_3(3) = \Phi_3(2) \uplus \{< t_5, + >\}$ and $\Phi_3(4) = \Phi_3(3)$. Therefore, according to Equation 3, the valid tokens and the corresponding candidate substrings of window $\mathcal{W}_3^4$ can be expressed as:

$$\begin{aligned}\Psi(3) = & < \mathcal{W}_3^1, \{t_3\} > \bigcup < \mathcal{W}_3^2, \{t_4\} > \bigcup \\ & < \mathcal{W}_3^3, \{t_4, t_5\} > \bigcup < \mathcal{W}_3^4, \{t_4, t_5\} >\end{aligned}$$

The second issue follows from the fact that the candidate substrings have different lengths. For one valid token $t \in \mathcal{T}$, it might belong to multiple $\Phi_p(l)$ with different values of l. Then we should be able to identify $\mathcal{W}_p^l$ with different l by scanning $\mathcal{L}[t]$ only once. To reach this goal, we propose an effective data structure to connect the candidate substrings and list of entities. Specifically, after moving to the end of the document using Window Extend and Window Migrate, we collect the first valid token set $\Phi_p(\mathcal{E}_\perp)$ and delta valid token sets $\Delta\phi(l)$ for all windows $\mathcal{W}_p$. Next we obtain $\Psi(p)$ using Equation 3 and construct an inverted index $\mathcal{I}$ for candidate substrings. Here a substring $\mathcal{W}_p^l \in \mathcal{I}[t]$ means that $\mathcal{W}_p^l$ is a candidate for entities in $\mathcal{L}[t]$. Then to meet the condition of length and prefix filter, we also group $\mathcal{W}_p^l \in \mathcal{I}[t]$ by length l, denoted as $\mathcal{I}_l[t]$. For substrings $s \in \mathcal{I}_l[t]$, only the entities in groups $\mathcal{L}_{|e|}[t]$ s.t. $|e| \in [\lfloor l * \tau \rfloor, \lceil \frac{l}{\tau} \rceil]$ can be candidate of s. In this way, we can obtain the entity for all $\mathcal{W}_p^l$ with $t \in \Phi_p(l)$ by scanning $\mathcal{L}[t]$ only once. Then for a candidate substring $\mathcal{W}_p^l$, the set of entities can be obtained by $\bigcup_{t \in \mathcal{W}_p^l} \mathcal{L}_{|e|}[t]$.

Algorithm 4 demonstrates the steps of lazy candidate generation. We first collect $\Phi_p(0)$ and $\Delta\phi_p(l)$ for each window using the same method in Algorithm 3. We then initialize the global token dictionary and inverted index for substrings (line: 3). But unlike Algorithm 3, here we only track the valid tokens for each substring instead of generating the candidates. Next, we generate the valid token set for each substring using Equation 2 (line: 4). And we can collect all the valid tokens and their corresponding substrings from them (line: 5- 7). With such information, we can build a mapping between the groups with different lengths $|e|$ in the inverted index and the candidate substrings with different lengths l s.t. $\lfloor |e| * \tau \rfloor \le l \le \lceil \frac{|e|}{\tau} \rceil$(line: 9). Then we scan the inverted list only once and collect the candidates. Finally, the entities for a candidate substring can be obtained from the union of the inverted indexes of all its valid tokens (line: 11). We summarize the correctness of Lazy Candidate Generation in Theorem 4.5.

THEOREM 4.5 (CORRECTNESS). *The Lazy Candidate Generation method will not involve any false negative.*

```
Algorithm 4: Lazy Candidate Generation($\mathcal{CI}$, $d$, $\tau$)
Input: $\mathcal{CI}$: The Inverted Index; $d$: The Document; $\tau$: The
       threshold of JAC
Output: $C$: The candidates
begin
    Use similar methods in Algorithm 3 to generate $\Phi_p(0)$
    and $\Delta\phi_p(l)$ for each window $\mathcal{W}_p$.
    Initialize $\mathcal{T}$ and $\mathcal{I}$;
    Generate all valid token set $\phi_l(p)$ using Equation 2;
    foreach $\phi_p(l)$ do
        collect the valid tokens, update $\mathcal{T}$;
        Construct the inverted index for substrings $\mathcal{I}$;
    foreach $t \in \mathcal{T}$ do
        Map entities in $\mathcal{L}_{|e|}[t]$ with candidate substrings in
        $\mathcal{I}_l[t]$ s.t. length filter;
        Scan $\mathcal{L}[t]$, obtain candidates for each $l$;
        $C = C \cup < \mathcal{W}_p^l, \bigcup_{t \in \mathcal{W}_p^l} \mathcal{L}_{|e|}[t] >$;
    return $C$;
end
```

5 DISCUSSION

In this section, we discuss about the scope as well as the generalization of our work.

Gathering Synonym Rules We first discuss about the way to obtain synonym rules. In our work, we make an assumption that the set of synonyms are known ahead of time. But it will not influence the generalization of our Aeetes framework. For a collection of documents, there are multiple sources of the synonyms rules. We list some of them below:

- Firstly, the synonym rules can come from common sense as well as knowledge bases. For example, we can use the synonyms provide by WordNet [3] as the input of Aeetes. The common sense knowledge can also provide rich source of synonyms, such as the abbreviation of institutes, address and locations used in our DBWorld and USJob datasets.
- Secondly, some domain specific applications provided the set of synonyms. For example, in the PubMed dataset, the synonyms are created by domain experts, which are very important information in understanding the medical publications. Therefore, performing AEES on such kinds of applications is with great values of practical application.
- Lastly, we can also discover the synonyms from documents with existing systems. For example, the output of [7] and [22] can be utilized directly as the input of our framework.

There are also some previous studies about data transformation and learning by example, which are summarized in Section 7. These studies are orthogonal to our work as they focused on detecting high-quality set of rules while our problem is about how to perform approximate entity extraction with predefined synonym rules. The synonyms discovered by them can also be used as the input of our framework.

Generation of Non-conflict Rule Set Given an entity e, let $\mathcal{A}_c(e)$ be the set of complete applicable rules of e and lhs_i be the left-hand of the rule r_i ($\langle\ lhs_i \Leftrightarrow rhs_i \rangle$). Without loss of generality, we assume the lhs of an applicable rule is a subsequence of the entity. Two rules r_i and r_j are conflict rules if $lhs_i \cap lhs_j \neq \emptyset$. Our goal is to choose a non-conflict set $\mathcal{A}(e) \subseteq \mathcal{A}_c(e)$ such that (i) all rules in $\mathcal{A}(e)$ are non-conflict and (ii) the cardinality of $\mathcal{A}(e)$ is as large as possible.

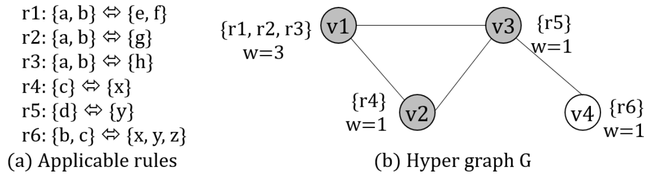


Figure 7: Hypergraph for the applicable rules of the entity $\{a, b, c, d\}$.

The non-conflict rule set can be obtained in the following way:

(1) First build a hypergraph $G = (V, E, W)$ for $\mathcal{A}_c(e)$. Each vertex $v \in V$ corresponds to a set of applicable rules whose left-hands are the same. The number of rules in the vertex v is the weight w of v. There is an edge $e \in E$ between two vertices whose rules are non-conflict.

(2) With such a hypergraph, we can obtain the set of non-conflict rules by finding the maximum weighted clique in the hypergraph G.

Example 5.1. Consider the set of complete applicable rules $\mathcal{A}_c(e)$ for the entity $e = \{a, b, c, d\}$ in Figure 7(a). The corresponding hypergraph G is shown in Figure 7(b). For instance, v_1 contains $\{r_1, r_2, r_3\}$ as they have identical left-hands, i.e., $\{a, b\}$. There is an edge between v_1 and v_2 since $\{a, b\} \cap \{c\} = \emptyset$. In this hypergraph, $\{v_1, v_2, v_3\}$ is the maximal weighted clique. Therefore, the final non-conflict applicable rules $\mathcal{A}(e) = \{r_1, r2, r_3, r_4, r_5\}$.

Unfortunately, finding the maximal weighted clique is a well-known NP-Complete problem. In order to efficiently find $\mathcal{A}(e)$ with large enough cardinality, we adopt a greedy algorithm with the following steps. Firstly, we choose the vertex v^* with maximal weight as a start point. Next we pick the next vertex v with the maximal weight among the unseen vertices such that adding v to the current clique is still a clique. Then we repeat step 2 until no more vertex can be added into the result set.

Example 5.2. Consider the set of complete applicable rules $\mathcal{A}_c(e)$ for entity $e = \{a, b, c, d\}$ in Figure 7 again. The greedy algorithm first chooses v_1 as it has the maximal weight. Then the algorithm picks v_2 since , it is still a clique after adding v_2. Similarly, v_3 is also added. Finally, the clique is $\{v_1, v_2, v_3\}$ and the corresponding non-conflict applicable rules are $\mathcal{A}(e) = \{r_1, r2, r_3, r_4, r_5\}$. Here the greedy algorithm achieves the optimal result.

6 EXPERIMENTS

6.1 Environment and Settings

In this section, we evaluated the effectiveness and efficiency of all proposed algorithms on three real-life datasets:

- PubMed. It is a medical publication dataset. We selected $100,000$ paper abstracts as documents and keywords from $10,000,000$ titles as entities to construct the dictionary. In addition, we collect $50,476$ synonym Mesh [4] (Medical Subject Headings)[5] term pairs, which are provided by the domain experts.

[3] https://wordnet.princeton.edu/

[4] https://www.ncbi.nlm.nih.gov/mesh

[5] Mesh is the NLM controlled vocabulary thesaurus used for indexing articles for PubMed.

- DBWorld. We collected 1,000 message contents as documents and keywords in 1,414 titles as entities in the dictionary. We also gather 1076 synonym rules including conference names, university names and country names which are common sense knowledge.
- USJob. We chosen 22,000 job descriptions as documents, and 1,000,000 keywords in the job titles as entities in the dictionary. In addition, we collected 24,305 synonyms including the abbreviations of company names, state names and the different names of job positions.

Table 1 gives the statistics of datasets, including the average number of tokens in documents (avg $|d|$), average number of tokens in entities (avg $|e|$), and average number of applicable rules on one entity (avg $|\mathcal{A}(e)|$).

Table 1: Dataset statistics.

	# docs	# entities	# synonyms	avg $\lvert d\rvert$	avg $\lvert e\rvert$	avg $\lvert\mathcal{A}(e)\rvert$
PubMed	8,091	370,836	24,732	187.81	3.04	2.42
DBWorld	1,414	113,288	1,076	795.89	2.04	3.24
USJob	22,000	1,000,000	24,305	322.51	6.92	22.7

All experiments were conducted on a server with an Intel(R) Xeon(R) CPU processor (2.7 GHz), 16 GB RAM, running Ubuntu 14.04.1. All the algorithms were implemented in C++ and compiled with GCC 4.8.4.

6.2 Evaluation of Effectiveness

First, we evaluate the effectiveness of our metric JACCAR by comparing with the state-of-the-art syntactic similarity metrics.

Ground truth For our task, there is no ground truth on these datasets. Borrowing the idea from previous studies on ASJS problem [19, 29], we manually create the ground truths as following: We marked 100 substrings in the documents of each dataset such that each of the marked substrings has one matched entity in the origin entity dictionary. Each pair of marked substring and the corresponding entity is a ground truth. For example, (*Primary Hypertrophic Osteoarthropathy*, *Idiopathic Hypertrophic Osteoarthropathy*), (*Univ. of California Berkeley USA*, *UC Berkeley USA*), and (*SDE in FB*, *software development engineer in Facebook*) are ground truths in PubMed, DBWorld and USJob dataset respectively.

Baseline methods To demonstrate the effectiveness of applying synonym rules, we compare JACCAR with two state-of-the-art syntactic similarity metrics: (i) Jaccard, which is the original Jaccard similarity; and (ii) Fuzzy Jaccard (FJ), which is proposed in [32]. As they are just based on syntactic similarity, they cannot make use of the synonym rules.

We evaluate the effectiveness of all the similarity measures by testing the *Precision* (short for "P"), *Recall* (short for "R"), and *F-measure* (short for "F"), where F-measure = $\frac{2\times P\times R}{P+R}$ on the three datasets.

Results Table 2 reports the precision, recall and the F-measure of all similarity measures on three datasets. We have the following observations. Firstly, JACCAR obtains higher F-measure scores than Jaccard and FJ. The reason is that JACCAR can use synonym rules to detect the substrings that are semantically similar to the entities, which indicates the advantage of integrating syntactic metrics with synonyms in the entity extraction problem. Secondly, FJ has higher Precision scores than Jaccard as FJ can identify tokens with minor typos. However, Jaccard has higher Recall scores than FJ because FJ may increase the similarity score for substrings that are not valid ground truth.

We present a sample of ground truth for each dataset in Figure 8 to perform a case study on the quality of three similarity metrics. We can see that in PubMed, both Jaccard and FJ are equal to 0. This is because the ground truth substring has no common (or similar) tokens with the entity. JACCAR =1.0 as JACCAR can apply the second synonym to the entity. In DBWorld, FJ has higher similarity score than Jaccard. The reason is that Jaccard can only find three common tokens "The University of " between the substring and the entity. But FJ can get extra benefit by identifying "Aukland" in the document is similar to "Auckland" in the entity (as their edit-distance is only 1). JACCAR achieves the highest score as JACCAR can apply the first synonym on the entity to obtain two more common tokens, i.e., "New Zealand". Similarly, in USJob, FJ has a higher score than Jaccard and JACCAR achieves the highest score.

6.3 Evaluation of Efficiency

Next we look at the efficiency of proposed techniques. We use the average extraction time per document as the main metric for evaluation.

End-to-end performance First we report the end-to-end performance. As there is no previous study on the AEES problem, we extend Faerie [13], which reports the best performance in AEE task, and propose FaerieR to serve as the baseline method. In order to let FaerieR handle the AEES problem, for each dataset we perform a preprocessing by using all applicable synonym rules to all entities in the dictionary so as to construct a derived dictionary. Then we use such a derived dictionary as the input of Faerie. After that, we conduct post-processing to recognize the pairs of origin entity and substrings in the document. For FaerieR, we omit the preprocessing and post-processing time and only report the extraction time by the original Faerie framework. For the implementation of Faerie, we use the code obtained from the original authors.

We compare the overall performance of Aeetes and FaerieR with different threshold values ranging from 0.7 to 0.9 on all three datasets. As shown in Figure 9, Aeetes outperforms FaerieR by one to two orders of magnitudes. The main reason is that we proposed a series of pruning strategies to avoid duplication computation came from applying synonyms and the overlaps in documents.

In the experiment, we observe that the bottleneck of memory usage is index size. And we report it for Aeetes and FaerieR as following. In PubMed, the index sizes of Aeetes and FaerieR are 10.6 MB and 6.9 MB, respectively. In DBWorld, the index sizes of Aeetes and FaerieR are 4.2 MB and 1.9 MB, respectively. While in USJob, the results are 113.2 MB and 54.3 MB, respectively. We can see that compared with FaerieR, the clustered inverted index of Aeetes has around twice larger size than FaerieR. The main reason is that we need to record the group relation for clustered index and use hashing tables to accelerate the query processing. But Aeetes can achieve much better performance by proper designing the search algorithm and utilizing the memory space smartly.

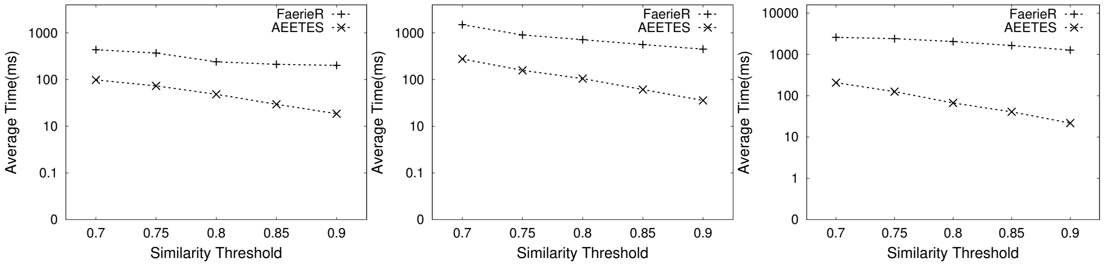
Optimizations Techniques We evaluate the filtering techniques proposed in Section 4. We implement four methods: Simple is the straightforward method to directly apply length and prefix filter by enumerating substrings; Skip is the method that adopts the

Table 2: Quality of similarity measures (P: short for Precision, R: short for Recall, F: short for F-measure).

	PubMed									DBWorld									USJob								
θ	Jaccard			FJ			JaccAR			Jaccard			FJ			JaccAR			Jaccard			FJ			JaccAR		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
0.7	0.24	0.73	0.36	0.35	0.73	0.47	0.96	0.89	0.92	0.24	0.67	0.35	0.39	0.68	0.50	0.92	0.91	0.92	0.20	0.71	0.31	0.37	0.77	0.50	0.94	0.94	0.94
0.8	0.14	0.88	0.24	0.34	0.77	0.47	0.95	0.93	0.94	0.24	0.89	0.38	0.36	0.84	0.50	0.90	0.94	**0.92**	0.20	0.83	0.32	0.29	0.85	0.43	0.92	0.97	0.94
0.9	0.12	0.92	0.21	0.28	0.85	0.42	0.95	0.98	**0.96**	0.23	0.92	0.37	0.35	0.90	0.50	0.88	0.93	0.90	0.18	0.90	0.30	0.25	0.86	0.39	0.92	0.98	**0.95**

PubMed	DBWorld	US Job
Entity: moschcowitz disease **Synonyms**: *moschcowitz disease ⇔ familial thrombotic thrombocytopenia purpura* *moschcowitz disease ⇔ thrombotic thrombocytopenic purpura* **Document**: "... the diagnostic challenge of acquired **thrombotic thrombocytopenic purpura** in children ..."	**Entity**: The University of Auckland NZ **Synonyms**: *NZ ⇔ New Zealand* *University ⇔ Univ.* **Document**: "... Gillian Dobbie, **The University of Aukland New Zealand**. Walid Aref, Purdue Univ. USA. Guoliang Li, Tsinghua University. Holger Pirk, MIT ..."	**Entity**: amazon database administrator **Synonyms**: *amazon ⇔ amzn* *database ⇔ databases* **Document**: "... position as a **databases administrator in amzn**. This position will give someone the ability to have input in to systems design ..."
Jaccard = 0.0 | FJ = 0.0 | **JaccAR = 1.0**	Jaccard = 0.38 | FJ = 0.54 | **JaccAR = 0.71**	Jaccard = 0.17 | FJ = 0.37 | **JaccAR = 0.75**

Figure 8: Three examples to illustrate the quality of similarity measures. The substrings with red font in the documents are marked as ground truth results.



(a) Pubmed

(b) DB World

(c) US Job

Figure 9: End-to-end performance

clustered inverted index to skip dissimilar groups; Dynamic is the method that dynamically computes the prefix for substrings in a document; Lazy is the method that generates candidates in a lazy manner.

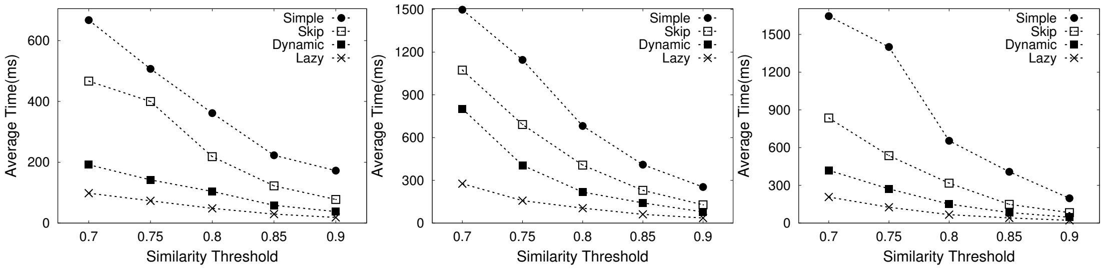
The results of the average extraction time are shown in Figure 10. We can see that Lazy achieves the best results. This is because it only needs to scan the inverted index of each token once. Although it requires some extra processing to allocate the candidate substrings with entities, the overhead can be very slight with proper implementation. The performance of Dynamic ranks second as it can dynamically maintain the prefix and does not need to compute from scratch. The reason it has worse performance than Lazy is that if a valid token exists in different windows, it needs to scan the same inverted index multiple times, which leads to heavy filter cost. Skip performs better than Simple as it utilizes the clustered inverted index to avoid visiting groups of entities that do not satisfy the requirement of length filter.

To further demonstrate the effect of filter techniques, we report in Figure 11 the average number of accessed entries in the inverted indexes, which provides a good metric to evaluate the filter cost. We can see that the results are consistent with those in Figure 10. For example, on the PubMed dataset when $\tau = 0.8$, Simple needs to access $326,631$ inverted index entries per document; Skip reduces the number to $126,895$; while the numbers of Dynamic and Lazy are $16,002$ and $6,120$, respectively.

Scalability Finally, we evaluate the scalability of Aeetes. In Figure 12, we vary the number of entities in each dataset and test the average search time for different threshold values. We can see that as the number of entities increased, Aeetes scales very well and achieves near linear scalability. For example, on the USJob dataset for $\tau = 0.75$, when the number of entities ranges from $200,000$ to $1,000,000$, the average running time is 43.26, 48.82, 62.71, 80.43 and 125.52 ms respectively.

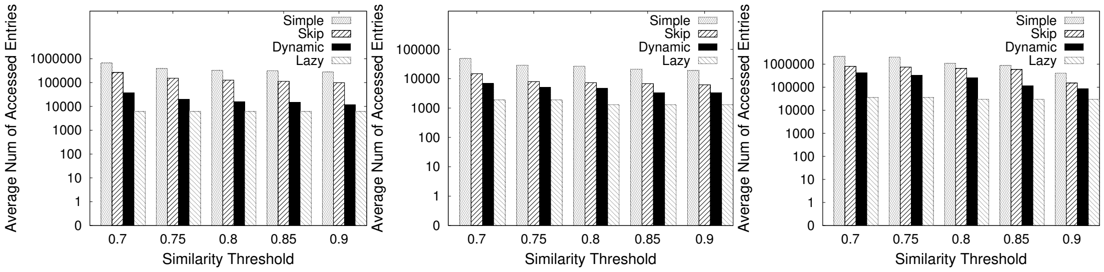
7 RELATED WORK

Approximate dictionary-based Entity Extraction Previous studies focusing on this problem only consider syntactic similarity. Chakrabarti et al. [8] proposed a hash-based method for membership checking. Wang et al. [35] proposed a neighborhood generation-based method for AEE with edit distance constraint, while Deng et al. [12] proposed a trie-based framework to improve the performance. Deng et al. [13] proposed Faerie, an all-purposed framework to support multiple kinds of similarity metrics in AEE problem. Wang et al. [34] addressed the local similarity search problem, which is a variant of AEE problem but with more limitations. All above methods only support syntactic similarity and cannot take synonyms into consideration.



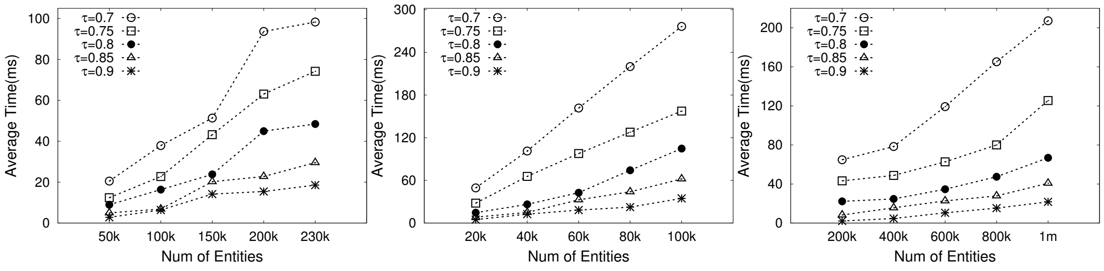
(a) Pubmed (b) DB World (c) US Job

Figure 10: Effect of Filtering techniques: Query Time



(a) Pubmed (b) DB World (c) US Job

Figure 11: Effect of Filtering techniques: Number of Accessed Entries



(a) Pubmed (b) DB World (c) US Job

Figure 12: Scalability: varying number of entities.

String Transformation and Synonym Discovery In some scenarios, the synonym rules may not exist, and it is impractical for a human to manually create a large set of rules. To solve this problem, some previous studies learn the rules using both supervised and unsupervised techniques. Arasu et al. [4] learned general syntactic rules from a small set of given matched records. Singh et al. improved the performance by leveraging language models [27] and semi-supervised learning [26]. Abedjan at al. [1] proposed the DataXFormer system to discover general transformations from web corpus. Singh et al. [28] addressed the same problem using program synthesis rules. Chakrabarti et al. [7] proposed novel similarity functions for synonym discovery from web scale data while He et al. [16] focused on finding synonyms in web tables. Qu et al. [22] discovered synonyms from text corpus with the help of knowledge base.

Entity Matching Entity matching has been a popular topic for decades. An extensive survey is conducted in [14]. Bilenko et al. [6] treated entity matching as a classification problem and proposed machine learning based solutions. Argrawal et al. [2] improved the quality of entity matching by considering errors in words and proposed efficient indexing techniques to improve performance. Wang et al. [33] proposed a learning-based framework to automatically learn the rules for entity matching. Firmani et al. [15] adopted a graph based model to develop an on-line framework for entity matching. Verroios et al. [30] integrated human ratings into entity matching and designed a crowdsourcing framework. Lin et al. [18] proposed a novel ranking mechanism to investigate the combinations of multiple attributes. Such studies mainly worked on collections of entities, while our problem requires to recognize approximate matching entities from documents. It could be an interesting direction of the future work to extend our framework to support other semantic similarity functions proposed here.

String Similarity Query Processing Approximate dictionary-based Entity Extraction (AEE) is a typical application in the field of string similarity query processing. There are also many studies on string similarity queries. Most of them only support syntactic similarity metrics. Among them some are designed for token-based similarity metrics, i.e. Jaccard, Cosine and Overlap, such as [9, 23, 36, 39, 40]; Others are designed for character-based similarity metrics i.e. edit distance, [17, 31, 37, 38]. Wang et al. [32]

combined above two categories of similarity metrics and proposed an efficient framework to support string similarity join. Some previous works tried to support synonym rules in the problem of string similarity join. They proposed some similarity functions to integrate the semantic of synonym rules into Jaccard similarities, such as *JaccT* [3], *SExpand* [19, 20] and *pkduck* [29]. However, they cannot be applied in the AEES problem as we have discussed in Section 2.

8 CONCLUSION AND FUTURE WORK

In this paper, we formally introduced the important problem of Approximate dictionary-based Entity Extraction with Synonyms and proposed an end-to-end framework Aeetes as the solution. We proposed a new similarity metrics to combine syntactic similarity metrics with synonyms and avoid the large overhead of on-line processing documents. We then designed and implemented a filter-and-verification strategy to improve the efficiency. Specifically, for the filtering step, we proposed a dynamic prefix computing mechanism and a lazy candidate generation method to reduce the filter cost. Experimental results on real world dataset demonstrated both the efficiency and effectiveness of our proposed framework.

For future work, we will (i) devise techniques to improve the verification step; (ii) extend our framework to support character-based similarity functions such as Edit Distance for tolerating typos in documents; (ii) support weighted synonym rules by assigning different weights to different rules; and (iii) integrate our techniques into open-source database systems.

REFERENCES

[1] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *ICDE*, pages 1134–1145, 2016.

[2] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *SIGMOD*, pages 927–938, 2010.

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, pages 40–49, 2008.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1):514–525, 2009.

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

[6] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

[7] K. Chakrabarti, S. Chaudhuri, T. Cheng, and D. Xin. A framework for robust discovery of entity synonyms. In *KDD*, pages 1384–1392, 2012.

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

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

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

[11] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *SIGKDD*, pages 89–98, 2004.

[12] D. Deng, G. Li, and J. Feng. An efficient trie-based method for approximate entity extraction with edit-distance constraints. In *ICDE*, pages 762–773, 2012.

[13] D. Deng, G. Li, J. Feng, Y. Duan, and Z. Gong. A unified framework for approximate dictionary-based entity extraction. *VLDB J.*, 24(1):143–167, 2015.

[14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[15] D. Firmani, B. Saha, and D. Srivastava. Online entity resolution using an oracle. *PVLDB*, 9(5):384–395, 2016.

[16] Y. He, K. Chakrabarti, T. Cheng, and T. Tylenda. Automatic discovery of attribute synonyms using query logs and table corpora. In *WWW*, pages 1429–1439, 2016.

[17] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[18] C. Lin, J. Lu, Z. Wei, J. Wang, and X. Xiao. Optimal algorithms for selecting top-k combinations of attributes: theory and applications. *VLDB J.*, 27(1):27–52, 2018.

[19] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384, 2013.

[20] J. Lu, C. Lin, W. Wang, C. Li, and X. Xiao. Boosting the quality of approximate string matching by synonyms. *ACM Trans. Database Syst.*, 40(3):15:1–15:42, 2015.

[21] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[22] M. Qu, X. Ren, and J. Han. Automatic synonym discovery with knowledge bases. In *KDD*, pages 997–1005, 2017.

[23] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *ICDE*, pages 1059–1070, 2017.

[24] S. Sarawagi. Information extraction. *Foundations and Trends in Databases*, 1(3):261–377, 2008.

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

[26] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[27] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8):740–751, 2012.

[28] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden, P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[29] W. Tao, D. Deng, and M. Stonebraker. Approximate string joins with abbreviations. *PVLDB*, 11(1):53–65, 2017.

[30] V. Verroios, H. Garcia-Molina, and Y. Papakonstantinou. Waldo: An adaptive human interface for crowd entity resolution. In *SIGMOD*, pages 1133–1148, 2017.

[31] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

[32] 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.

[33] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[34] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005, 2016.

[35] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, pages 759–770, 2009.

[36] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

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

[38] X. Yang, Y. Wang, B. Wang, and W. Wang. Local filtering: Improving the performance of approximate queries on string collections. In *SIGMOD*, pages 377–392, 2015.

[39] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan. An efficient framework for exact set similarity search using tree structure indexes. In *ICDE*, pages 759–770, 2017.

[40] Y. Zhang, J. Wu, J. Wang, and C. Xing. A transformation-based framework for knn set similarity search. *IEEE Trans. Knowl. Data Eng.*, 2019.