Efficient Approximate Entity Extraction with Edit Distance Constraints

Wei Wang[*] Chuan Xiao Xuemin Lin
University of New South Wales and NICTA
Australia
{weiw, chuanx, lxue}@cse.unsw.edu.au

Chengqi Zhang
University of Technology, Sydney
Australia
chengqi@it.uts.edu.au

ABSTRACT

Named entity recognition aims at extracting named entities from unstructured text. A recent trend of named entity recognition is finding approximate matches in the text with respect to a large dictionary of known entities, as the domain knowledge encoded in the dictionary helps to improve the extraction performance.

In this paper, we study the problem of approximate dictionary matching with edit distance constraints. Compared to existing studies using token-based similarity constraints, our problem definition enables us to capture typographical or orthographical errors, both of which are common in entity extraction tasks yet may be missed by token-based similarity constraints. Our problem is technically challenging as existing approaches based on q-gram filtering have poor performance due to the existence of many short entities in the dictionary. Our proposed solution is based on an improved neighborhood generation method employing novel partitioning and prefix pruning techniques. We also propose an efficient document processing algorithm that minimizes unnecessary comparisons and enumerations and hence achieves good scalability. We have conducted extensive experiments on several publicly available named entity recognition datasets. The proposed algorithm outperforms alternative approaches by up to an order of magnitude.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Textual Databases*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Pattern Matching*

General Terms

Algorithms, Performance

[*]This work was partly done while Wei Wang was visiting the University of Technology, Sydney.

Keywords

approximate dictionary matching, named entity recognition, edit distance

1. INTRODUCTION

Named entity recognition (NER) aims at finding named entities in unstructured text. It is an important task in information extraction and integration, and serves many applications, including identifying geographical locations for geotagging [2], identifying gene and protein names from MEDLINE abstracts for text mining [35, 14], identifying names and their categories to improve Web search [31]. A promising approach is to leverage large dictionaries of known entities. This approach has been shown to substantially improve the extraction performance over traditional NER systems due to the extra domain knowledge encoded in the dictionary [15].

Since entities in the dictionary are represented as strings, the dictionary-based named entity extraction problem can be modeled as an *approximate dictionary matching* problem, that is, given a dictionary of strings and a query document, we want to find all the approximate occurrences of any dictionary string in the query document. While the exact version of this matching problem is well understood, approximate matching is much harder as every entity in the dictionary could be a candidate. Therefore, indexing the entity dictionary is a popular approach; it will identify only a subset of entities as candidates for each query, thus improving the overall performance. [10] proposed to build inverted index on tokens and focuses on computational sharing among subsequent queries. [8] employs an extended prefix filtering [12] and a superimposed code based filtering method. [1] studies a similar problem with the different setting that both the dictionary and documents are indexed.

A limitation in the above work is that it only considers *token-based* similarity measures. That is, both entities and documents are tokenized with respect to word boundaries and popular *set-based* similarity measures, such as the Jaccard similarity, are then used to retrieve similar entities for a given query string. There are several disadvantages associated with this approach.

- *It may miss some match.* While entities in the dictionary are usually cleaned and standardized, they are usually not in the document: there could be typographical, orthographical or extraction errors, or there may be no standard name for some entities (e.g., biological entities [35]). To make things worse, many entities contain very few tokens. For example, "`al qaeda`" will not match

"al qaida" unless we use a low Jaccard similarity threshold of 0.33. Still, "al-qaeda" or "al-qa'ida" won't be matched with "al qaeda" as the Jaccard similarity value is 0. In contrast, edit distance with a threshold 2 can capture all these alternative spellings of the *same* entity[1].
- *It may result in too many matches.* Set-based similarity measures disregard the order among tokens and dissimilarity between mismatched tokens. For example, "al qaeda" will match "al gore" as well as "al pacino" if we use a Jaccard similarity threshold of 0.33. Edit distance with a threshold of 2 won't report such false positive matches.

To remedy the above problems, we propose to study the entity extraction problem with the well-known *string-based* dissimilarity measure — edit distance. Edit distance measures the minimum number of edit operations (insertion, deletion, and substitution) to transform one string to another. Edit distance can effectively capture typographic errors, words with alternative spellings, and does not rely on the separation of word boundaries. Hence, edit distance and its variants have been used in named matching [42] and record linkage [7].

A widely adopted method to find approximate matches with an edit distance constraint is based on q-grams [16]. However, a unique feature in named entity matching is the existence of many short entities in the dictionary. Matching short strings approximately results in the following dilemma: we *have to* use short q-grams to ensure matching strings have at least one common q-gram; however, it is known that short q-grams suffer from poor performance problems [42, 39].

In this paper, we propose to solve the approximate dictionary matching problem with edit distance constraint by an improved *neighborhood generation*-based method. The neighborhood generation method was traditionally considered only applicable to small alphabet size and small edit errors, as the size of the neighborhood is $O(m^\tau|\Sigma|^\tau)$, where m is the string length, Σ is the alphabet, and τ is the edit distance threshold [37]. Recently, [33] proposed the FastSS algorithm, which reduces the neighborhood size to $O(m^\tau)$ and was demonstrated to outperform the q-gram-based method on the English vocabulary. However, FastSS still cannot scale up to the diverse entity lengths and error levels for typical entity extraction tasks. In this work, we improve the FastSS method by novel partitioning and prefix pruning techniques and result in a neighborhood size of $O(l_p\tau^2)$, where $l_p \leq m$ is a tunable parameter. Another novelty lies in the document processing algorithm, where we apply a semi-join [6] style reduction technique to avoid considering many unnecessary query and entity pairs, in addition to other optimizations. Experiment results show that the proposed algorithm has superior performance to other alternatives on publicly available named entity recognition datasets with up to 25x speedup.

Note that allowing approximate matching in NER will increase false positive matches. In addition, orthographical matching does not solve the issue of homonyms[2]. In this work, we aim to utilize this approach to increase the recall of the NER systems, following [35, 38].[3] Additional post-processing methods can be applied to achieve high precisions. We also focus on solving the problem exactly, thus excluding approximate (e.g., LSH) or heuristic methods (e.g., BLAST).

Our contributions can be summarized as follows:

- We study the problem of efficiently performing dictionary-based entity extraction with edit distance constraints. It captures an important class of approximately matching entities that is hard to be detected by existing methods based on token-level similarity measures.
- We address the major technical problem in existing neighborhood generation-based algorithms, thus making it a highly competitive method for entity extraction. We devise new partitioning and prefix pruning techniques to reduce the size of the neighborhood from $O(m^\tau)$ to $O(l_p\tau^2)$.
- We propose an efficient query processing algorithm. The efficiency mainly comes from two facts: we avoid considering unnecessary entity and query segment combinations and we exploit the sharing of computation.
- We have conducted extensive experiments using several named entity recognition datasets in various domains. The proposed method has been shown to outperform other alternatives by up to an order of magnitude.

2. PROBLEM DEFINITION AND PRELIMINARIES

2.1 Problem Definition

Definition 1. *Given a document D and a dictionary E of entities, the task of* approximate dictionary matching with edit distance threshold τ *is to find all substrings in D such that they are within τ edit distance from one of the entities in E, or more formally, return* $\{\,(D[i\ ..\ j], E_k) \mid \exists k,\ ed(E_k, D[i\ ..\ j]) \leq \tau\,\}$

A straight-forward algorithm would be to iterate through all the valid substrings of the document $D[i\ ..\ j]$, and issue a *similarity selection query* to the dictionary to retrieve the set of entities that satisfy the constraint. We refer to each substring as a *query segment*. As is typical in entity extraction tasks, we do not assume the documents to be matched are given before hand.

Notations. We denote the length of the shortest (longest) entity in the dictionary as $L_{\min}$ ($L_{\max}$). We use $D[i..j]$ to denote a substring of D that starts at the i-th position and ends at the j-th position. All arrays indexes start from 1. We denotes the length of a string s as $|s|$. The l-prefix of a string s is its first l characters, i.e., $s[1\ ..\ l]$. We use [abc . . .] to indicate an ordered sequence of characters.

2.2 Analysis of Previous Approaches

A widely used method for answering similarity selection or join queries with edit distance threshold is to convert the edit distance constraint into a weaker *count* constraint on matching q-grams. Given a string s, we obtain its q-gram multiset by sliding a window of width q over the string. If two strings s and t are within edit distance τ, they must share at least $LB_{s;t}$ q-grams, where $LB_{s;t} = \max(|s|, |t|) - q + 1 - q\tau$. Other filtering criteria, such as length filtering and position filtering, can be incorporated into the above count filtering [16]. An efficient way to find the candidate strings

[1]http://en.wikipedia.org/wiki/Al-Qaeda

[2]E.g., race (noun) as a contest vs. race (noun) as a taxonomic group [29].

[3]An anecdotal example is that we are able to Ronald Regan to Ronald Reagan in the CONLL dataset used in the experiment.

that share sufficient number of matching q-grams with a given query string is to use the prefix filtering technique [12].

However, the q-gram-based approach does not work well on our approximate dictionary matching problem mainly due to the fact that the q-grams used must be smaller than $\frac{L_{\min}+1}{\tau+1}$, as otherwise the lower bound of matching q-grams won't be positive. The use of short q-grams results in poor performance for the following reasons.

1. *Long postings lists.* When q is small, the vocabulary size is small and hence the postings lists for any q-gram tend to be long. All operations (e.g., reading or intersecting) regarding long postings list become expensive. It has been reported that the cost of similarity search or join using small q-grams (e.g., $q = 2$) is quite high [42, 39].
2. *Not selective for short entities.* For short entities, the lower bound of matching q-grams is usually low. This implies a long prefix for prefix-filtering-based methods. Not only more postings lists need to be retrieved, but also there is a greater possibility that some q-gram in the prefix has a extraordinarily long postings list. As a result, the set of candidate entities will be large.
3. *Hard to share computation.* Prefix filtering extracts an appropriate length of prefixes based on the decreasing order of token's *idf* values and the corresponding postings lists will be retrieved. So even if two query segments are overlapping or containing, their prefixes could be different and computation cannot be easily shared.

PartEnum [4] is another approach to tackle edit distance constraints. It generates signatures based on a two-level partitioning scheme and exhibits good performance when τ is small. However, its performance is critically dependent on the choice of partitioning parameters. As entities in the dictionary may have substantially different lengths, it is hard to set the parameters that works well for both short and long entities.

2.3 Neighborhood Generation

Another category of approaches to deal with edit distance query is the *neighborhood generation* method. Define the τ neighborhood of a string s ($U_\tau(s)$) as all strings that are at most τ edit distance away from s, i.e., $U_\tau(s) = \{ s' \mid ed(s, s') \le \tau \}$. To apply it to our problem, we need to generate and index $U_\tau(e)$ for every entity $e \in D$; at the query time, we only need to perform a single exact match using the query string s. While neighborhood generation-based approach is simple, it was mainly deemed as of theoretical interest only since the size of the neighborhood is $O(m^\tau |\Sigma|^\tau)$ for a string of length m [37].

Nevertheless, [33] recently proposes to generate the *deletion neighborhood* for both query and text, thus successfully reducing the neighborhood size to $O(m^\tau)$. This results in the FastSS algorithm, which achieves fast query performance, but is still limited to small τ and m.

In this work, we devise both the partitioning and prefix pruning technique which further reduce the size of the deletion neighborhood to $O(l_p \tau^2)$, where l_p is a tunable parameter. As a result, our method works well for a much larger range of string length than FastSS.

2.3.1 FastSS

We briefly summarize the deletion neighborhood-based method, FastSS, in order to best understand our proposed method.

We use $\delta(s, p)$ to denote the transformation of string s by deletion of the letter at position p. E.g., $\delta(\text{“notebook”}, 1) =$ “otebook” and $\delta(\text{“notebook”}, 2) =$ “ntebook”. The deletions can be applied recursively. For a number of deletions k, we use $\delta(\delta(\ldots\delta(s, p_1), p_2), \ldots, p_k)$ to denote the resulting string after k deletions, and $[\, p_1, p_2, \ldots, p_k \,]$ the *deletion list* of the resulting string. For example, $\delta(\delta(\text{“notebook”}, 1), 1) =$ “tebook”, and the deletion list is $[\, 1, 1 \,]$.

For a given string s and a number of deletions k, we call the result strings after deleting s by k characters at all possible positions the *k-variants* of s. The union of s's i-variants ($0 \le i \le k$) forms the *k-variant family* of s, denoted as $V(s, k)$. We can define it recursively as

$$V(s, 0) = \{ s \}; \qquad V(s, k) = \{ s \} \cup \bigcup_{i=1}^{|s|} V(\delta(s, i), k - 1).$$

The following lemma enables us to devise a filtering condition based on the generated variant families.

LEMMA 1 (VARIANT FILTERING PRINCIPLE [33]). *If two strings s and t are within edit distance τ, then they share at least one common variant in their τ-variant families.*

Note that two strings that pass the above filter do not necessarily satisfy the edit distance constraint. Edit distance computation is therefore needed to remove the false positives that survive the filter.

FastSS can be applied to our approximate dictionary matching problem as follows:

- In the *indexing phase*, we generate the τ-variant family for all the entities and index the variants using an inverted index.
- In the *document processing phase*, we enumerate all the appropriate query segments of the document (denoted as s); for each s, we generate its τ-variant family and probe the inverted index to find a set of candidate entities for s.
- In the *verification phase*, we compute the edit distance for all the candidate pairs and report those that are within τ as the final answer.

EXAMPLE 1. *Consider an entity* `qaeda` *in the dictionary and the current query segment is* `qaida`. *Let $\tau = 1$ and $q = 2$ (note that the two strings will have no q-gram in common if $q > 2$). Using the q-gram-based method, we need to extract the $(q\tau + 1)$-prefix of a string. We estimate bigrams' frequencies using some background corpus to derive the prefix of the query segment as*

$$\mathit{prefix}(\texttt{qaida}) = [\, \texttt{qa}, \texttt{id}, \texttt{ai} \,]$$

The q-gram-based method will retrieve and union the postings lists of `qa`, `id` *and* `ai`. *Most likely, the postings lists for* `id` *and* `ai` *will be quite long, and their union alone already results in a large candidate set to be verified.*

On the other hand, using the FastSS approach, the 1-variant family of `qaeda` *will be generated and indexed offline. At query time, the 1-variant family of the query* `qaida`, *which consists of* $\{$ `qaida`, `aida`, `qida`, `qaia`, `qaid` $\}$, *will be generated and their corresponding postings lists will be retrieved. Most likely, all of them will be very selective*[4] *and will result in a small candidate set.*

[4]See Table 1.

3. IMPROVED NEIGHBORHOOD GENERATION-BASED FILTERING

In this section, we introduce a new deletion neighborhood-based filtering method which employs new partitioning and prefix pruning techniques. Recall that the size of the deletion neighborhood size generated by the FastSS algorithm is $O(m^\tau)$. Our partitioning scheme reduces the neighborhood size to $O(m\tau+\tau^2)$, and the prefix pruning technique reduces it further to $O(l_p\tau^2)$, where l_p is a tunable parameter.

3.1 Partitioning

Our main idea is to partition the entities and queries into multiple partitions, such that we only need to generate and match 1-variants for each partitions.

Our method builds upon the following two important observations.

Error Reduction Consider a string s partitioned into k disjoint partitions, i.e., $s = s_1s_2 \dots s_k$ and another string s' resulting from applying τ edit operations on s. By a simple argument of the pigeon-hole principle, we know that there exists at least one partition, s_i, such that there is at most $\lfloor \frac{\tau}{k} \rfloor$ edit operations applied to it.

Alignment Consider any partition s_j in s and its corresponding "image" s'_j in s'. The offsets of s_j and s'_j might be different, but must be within the $[-\tau, \tau]$ range, as there are at most τ insertions or deletions in the *preceding partitions* that affect the alignment of the current partition.

A high-level description of the partitioning scheme is: we partition the entities and query segments into an appropriate number of partitions, yet still with the following guarantee: *if a query segment and an entity are within edit distance τ, there exists a query partition and its corresponding entity partition subject to appropriate amount of* shifting *and* scaling *such that the two partitions are within edit distance 1.* This will immediately give us a filtering condition. Below we derive appropriate parameter values for the above scheme and present more details.

Number of Partitions. Since our goal is to have at least one partition with at most one edit error, we require the minimum number of partitions k_τ to satisfy $2(k_\tau - 1) + 1 \geq \tau$. Therefore, we select $k_\tau = \lceil \frac{\tau+1}{2} \rceil$.

Partitioning Scheme. For a string s, we partition it into k_τ partitions in the following manner: the first $k_\tau - 1$ partitions have length $\left\lfloor \frac{|s|}{k_\tau} \right\rfloor$, and the last partition takes the rest of the string.

We define the following two operations that can be applied to a partition $s[i \mathinner{..} j]$ to generate its *partition variation* as:

Shifting by u gives us a substring of $s[(i+u) \mathinner{..} (j+u)]$.

Scaling by v gives us a substring of $s[i \mathinner{..} (j+v)]$ (i.e., fix the starting position and change the ending position).

EXAMPLE 2. *Consider two strings s = [`abcdefghijkl`] and s' = [`axxbcdefghxijkl`] obtained by inserting three* `x` *characters into s. When $\tau = 3$, $k_\tau = 2$. Therefore, the partitions of s and s' are:*

s = [`abcdef`], [`ghijkl`]
s' = [`axxbcde`], [`fghxijkl`]

Shifting the first partition of s by 3 gives us [`defghi`]. *Scaling the first partition of s by 1 gives us* [`abcdefg`]. *The two operations can be composed. E.g., shifting the first partition of s by 2 and scaling it by -1 gives us* [`cdefg`].

We define the following *transformation rules* to apply shifting and scaling to every partition of an entity.

- For the first partition, we only need to consider scaling within the range of $[-2, 2]$.
- For the last partition, we only need to consider the combination of the same amount of shifting and scaling within the range of $[-\tau, \tau]$ (so that the last character is always included in the resulting substring).
- For the rest of the partitions, we need to consider shifting within the range $[-\tau, \tau]$ and scaling within the range $[-2, 2]$.[5]

The following theorem gives us a filtering condition.

THEOREM 1 (PARTITIONED VARIANT FILTERING). *Consider a query string s and an entity e, both partitioned in our scheme into k_τ partitions. Denote P as the set of e's partition variations generated by the above transformation rules. If $ed(s, e) \leq \tau$, then there exist i, such that there is a partition variation of the i-th partition of e that is within edit distance 1 from the i-th partition of s.*

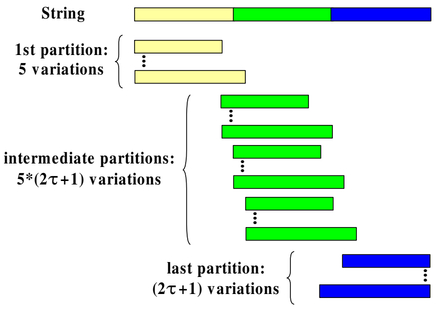


Figure 1: Illustration of Partition Variations Generated

Figure 1 gives an illustration of the partition variants generated by applying the transformation rules. The total amount of the 1-variants generated can be computed as $O(\tau m + \tau^2)$.

Special Case for $\tau = 2$ or $\tau = 3$. When $\tau \in \{2, 3\}$, $k_\tau = 2$. By analyzing all the possible combinations of insertion, deletion, and substitution errors, we find that we can have the following more strict transformation rules:

- For the first partition, we only need to consider scaling within the range of $[-1, 1]$.
- For the last partition, we only need to consider the combination of the same amount of shifting and scaling within the range of $[-1, 1]$.

EXAMPLE 3. *Continuing Example 2, we consider s as an entity and s' as a query segment. s will generate the following partition variations, together with their partition identifiers. Each of the following variants will generate their 1-variant family (not shown here), which will be indexed.*

[5]It can be shown that if two strings' length different by at most τ, after our partitioning, the length difference of their corresponding partitions (except the last one) is within $[-2, 2]$. This is the intuitive reason why scaling by $[-2, 2]$ is sufficient.

$\langle$ [abcd], 1 $\rangle$	$\langle$ [abcdefgh], 1 $\rangle$	$\underline{\langle \text{[ghijkl]}, 2 \rangle}$
$\underline{\langle \text{[abcde]}, 1 \rangle}$	$\langle$ [jkl], 2 $\rangle$	$\underline{\langle \text{[fghijkl]}, 2 \rangle}$
$\underline{\langle \text{[abcdef]}, 1 \rangle}$	$\langle$ [ijkl], 2 $\rangle$	$\langle$ [efghijkl], 2 $\rangle$
$\underline{\langle \text{[abcdefg]}, 1 \rangle}$	$\underline{\langle \text{[hijkl]}, 2 \rangle}$	$\langle$ [defghijkl], 2 $\rangle$

When the query segment s' comes in, its second partition, [fghxijkl]*, will have 1-variant match with s's partition variation* [fghijkl] *generated from s's second partition. Therefore, $\langle s', s \rangle$ will be identified as a candidate pair for further verification.*

If we use the strict transformation rules for $\tau = 3$, we only need to consider the underlined partition variations.

3.2 Prefix-based Pruning

The above scheme will generate partition variations whose size is at least $\left\lfloor \frac{m}{k_\tau} \right\rfloor - 2$. When m is large, there are still many variants generated. This has two disadvantages:

- It increases the space complexity, as the 1-variants from the entities need to be stored in the inverted index.
- It introduces overhead to query processing, because more 1-variants need to be enumerated from the query segments and probed against the index.

Meanwhile, note that the selectivity of long 1-variants will reach a diminishing return point quickly — when the expected number of matches for a length l variant is below one, it is not worthwhile to use variants longer than l.

Therefore, we have another pruning that is based on a fixed length prefix of each partition or partition variation. When the partition (variation) is longer than a prefix length l_p, we only use its l_p-prefix to generate its 1-variants. It can be shown that this pruning does not miss any result and we name it *prefix-based pruning.*

EXAMPLE 4. *Continuing the underlined partition variations in Example 3. Assume l_p is set to 3. Then 1-variants are generated from only the following prefixes.*

$\langle$ [abc], 1 $\rangle$	$\langle$ [ghi], 2 $\rangle$
$\langle$ [hij], 2 $\rangle$	$\langle$ [fgh], 2 $\rangle$

By setting $l_p \leq \left\lfloor \frac{m}{k_\tau} \right\rfloor - 2$, it can be shown that the total number of 1-variants generated is further reduced to $O(l_p \tau^2)$.

4. PROCESSING THE QUERY DOCUMENT

Entities in the dictionary usually vary substantially in length. For example, the length of gene names in the GENE dataset varies from 3 to 169. While FastSS is effective in dealing with approximate matching for short strings, its cost increases quickly when the length of the entities to be matched increases. Nevertheless, the improved neighborhood generation method we developed in the preceding section can handle long entities well.

This motivates us to use the value $k_\tau l_p + \tau$ to divide the entities in the dictionary into two parts: short and long entities. We choose the value so that each partition of potential matches for long entities is at least l_p long. This enables us to apply the prefix pruning on each partition of long entities.

In the following, we will introduce the indexing and document processing method in detail.

4.1 Indexing the Entities

We use different procedures to index short and long entities in the dictionary, and store them in two inverted indexes, I^{short} and I^{long}, respectively (See Algorithm 1).

- For each entity whose length is smaller than $k_\tau l_p + \tau$, we take a prefix of length $\min(|e|, l_p)$, generate and index its τ-variant family.
- For each entity whose length is no smaller than $k_\tau l_p$, we first partition it into k_τ partitions. Variations of each partition are generated by applying transformation rules (See Theorem 1). The l_p-prefix of each partition variation is used to generate its 1-variant family, which will be indexed. Note that the index for long entities maps a 1-variant to a list of $\langle$ *entity*, *partition_id* $\rangle$ pairs.

Algorithm 1: BuildIndex (E, τ, l_p)

```
1  for each $e \in E$ do
2      if $|e| < k_\tau l_p + \tau$ then
3          $V \leftarrow$ GenVariants($e[1 .. \min(l_p, |e|)], \tau$);
           /* The GenVariants ($s$, $k$) function generates
              the $k$-variant family of string $s$         */
4          for each $v \in V$ do
5              $I^{\text{short}}_v \leftarrow I^{\text{short}}_v \cup \{ e \}$;
6      if $|e| \geq k_\tau l_p$ then
7          $P \leftarrow$ the set of $k_\tau$ partitions of $e$;
8          for each $i$-th partition $p \in P$ do
9              $P^T \leftarrow$ TransformPartition($p$);
               /* according to the three
                  transformation rules in Section 3.1    */
10             for each partition variations $p^T \in P^T$ do
11                 $V \leftarrow$ GenVariants($p[1 .. l_p], 1$);
12                 for each $v \in V$ do
13                     $I^{\text{long}}_v \leftarrow I^{\text{long}}_v \cup \langle e, i \rangle$;
14 return $(I^{short}, I^{long})$
```

Note that entities with length between $k_\tau l_p$ and $k_\tau l_p + \tau$ will be indexed in both indexes. This is necessary to ensure no match between long entities and short query segments (or vice versa) is missed.

4.2 Processing the Query Document

With entities in the dictionary indexed, we can process the query document to find out all approximate matches. A straight-forward matching algorithm is based on the exhaustive search, i.e., to iterate through all the valid starting positions in the query document (i.e., $[1, |D| - L_{\min} + \tau + 1]$) and enumerate all possible query segment lengths (i.e., $[L_{\min} - \tau, L_{\max} + \tau]$); for each query segment, we probe the indexes to retrieve a set of candidate entities that may match the query segment within τ edit distance. Final results can be found by verifying each candidate pair by computing its edit distance. This algorithm has much room for improvement as not every query segment in the above enumeration has a match.

We propose a more efficient query processing algorithm. The idea is somewhat similar to semi-join [6]. We iterate over the query document and use a substring with limited length to find a set of candidate entities; based on these candidate entities, we then find a candidate set of query segments that might join with the candidate set of entities. The advantage of this method is that we do not need to enumerate non-promising pairs of entity and query segment as the exhaustive search does.

Given a query document, we need to extract all the approximate matches with respect to the given dictionary of entities. We iterate over all the possible *starting positions* p; we match a set of substrings starting from p in two phrases: we first match them against the long entities, and then with short entities. This is shown in Algorithm 2.

Algorithm 2: MatchDocument (D, E, τ)

```
1 for each starting position $p \in [1, |D| - L_{\min} + \tau + 1]$ do
2     SearchLong ($D[p \,..\, p + l_p - 1]$, $E$, $\tau$);
      /* matching entities no shorter than $k_\tau l_p$ */
3     SearchShort ($D[p \,..\, p + l_p - 1]$, $E$, $\tau$);
      /* matching entities of length in $[l_{\min}, k_\tau l_p)$ */
```

We shall introduce matching algorithm for long entities first, and then algorithm for short entities in Section 4.2.2.

4.2.1 Matching Long Entities

The matching process for long variants is complicated by the partitioning scheme. Since 1-variants were generated for the prefix (of length l_p) of each partition of an entity, we take the substring $D[p \,..\, p+l_p-1]$ and generate all its 1-variants. These 1-variants are matched against I^{long}, the inverted index for long entities. The result of the match gives us a set of candidate entities, and the matching partition identifiers (*pid*). However, a subtle technical issue is that for each entity e in the candidate set, they might match different substrings (or query segments) in the document. We call the procedure of determining the starting position and length of the set of query segments that matches an entity *query segment instantiation.* The inputs we have are (a) the length of the entity, and (b) the fact that the *pid*-th partition of the entity has 1-variant match with the substring $D[p \,..\, p+l_p-1]$. Given the length of the entity, we know the length of the candidate query segment, m, must be within the range $[|e| - \tau, |e| + \tau]$. For each possible length m, we can find out its partition size as $\left\lfloor \frac{m}{k_\tau} \right\rfloor$. Then its starting position can be found by going back $pid-1$ partitions, i.e., $p-(pid-1)\cdot \left\lfloor \frac{m}{k_\tau} \right\rfloor$. The above procedure is represented as the QuerySegmentInstantiation method (Line 8 in Algorithm 3).

Algorithm 3: SearchLong (s)

```
1  $R \leftarrow \emptyset$ ;                          /* holds results */
2  $C \leftarrow \emptyset$ ;                          /* holds candidates */
3  $V \leftarrow$ GenVariants($s, 1$) ;       /* gen 1-variant family */
4  for each $v \in V$ do
5      for each $\langle e, pid \rangle \in I_v^{\text{long}}$ do
6          $C \leftarrow C \cup \langle e, pid \rangle$ ;     /* duplicates removed */
7  for each $\langle e, pid \rangle \in C$ do
8      $S \leftarrow$ QuerySegmentInstantiation($e, pid$);
       /* returns
          the set of query segment candidates for $e$ */
9      for each $seg \in S$ do
10         if Verify($seg, e$) = true then
11             $R \leftarrow R \cup \langle seg, e \rangle$;
12 return $R$
```

Example 5. *Continue the previous example. Assume s is the only entity in the dictionary and s' is the query document, and the current starting position $p = 8$. Recall $\tau = 3$ and $l_p = 3$. Consider the invocation of* SearchLong *with the prefix $D[8 \,..\, (8+3-1)] =$ [fgh]. Its 1-variants family consists of* { [fgh], [fg], [fh], [gh] }. *After probing the index of 1-variants, we have three exact matching 1-variants and they are merged to a single candidate: $\langle s, 2 \rangle$. The length of entity s is 12, hence we need to consider candidate query segments whose size is between 9 and 15. If the query segment is of length 9, its partition size will be 4. This suggests the segment starts at position 4 (with length 9), which is subsequently verified against the entity s.*

In the following table, we mark the candidate set of query segments (and their two partitions) with gray background.

pos	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
$m = 9$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 10$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 11$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 12$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 13$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 14$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l
$m = 15$	a	x	x	b	c	d	e	f	g	h	x	i	j	k	l

There are several advantages of our document processing algorithm.

1. We can skip many unnecessary query segment and entity combinations. Consider Example 5 again and assume there are other entities (of different lengths) in the dictionary but they don't trigger any match with the query document. The exhaustive search method will have to enumerate query segments of all possible lengths while our method will only generate query segments relevant to a subset of entities.
2. Our algorithm leverages shared computation. Consider all the query segments starting from the same position, our algorithm fetches all the candidate entities only once while the exhaustive algorithm will repeat the same operation unnecessarily for up to $L_{\max} + \tau - l_p$ times. This is important as the entities in the dictionary could vary substantially in length and much computation can be saved in our approach.

It is possible that the above method will perform multiple verifications between a query segment and an entity, if they have 1-variant match on the prefix of more than one partition. We record all the verified pairs of query segment and entity to avoid this re-verification.

4.2.2 Matching Short Entities

Algorithm SearchShort can find all the approximately matches between the current substring with short entities in the dictionary.

The pseudo-code of the algorithm is given in Algorithm 4. It is similar to SearchLong with the following differences:

- We need to generate the τ-variant families for each possible length l between $L_{\min} - \tau$ and l_p (Lines 3 – 4).
- Since entities longer than l_p generates their τ-variant families using only their l_p prefix in the indexing phase, we need to distinguish them in the matching phase too. If the current query segment is shorter than l_p, every candidate pair formed by probing the index needs to be verified (Lines 13 – 15). Otherwise, we need to perform a simplified query segment instantiation procedure and verification for $2\tau + 1$ possible query segments (Lines 17 – 19).

Algorithm 4: SearchShort (s)

```
 1 $R \leftarrow \emptyset$ ;                               /* holds results */
 2 $C \leftarrow \emptyset$ ;                            /* holds candidates */
 3 for $l = L_{\min} - \tau$ to $l_p$ do
 4     $V \leftarrow$ GenVariants$(s, \tau)$;
       /* gen $\tau$-variant family                                  */
 5     for each $v \in V$ do
 6         for each $e \in I_v^{\text{short}}$ do
 7             if $l < l_p$ then
 8                 if $|e| - l \in [-\tau, \tau]$ then
 9                     $C \leftarrow C \cup \{e\}$;
10             else
11                 $C \leftarrow C \cup \{e\}$;
12     for each $e \in C$ do
13         if $l < l_p$ then
14             if Verify$(D[s \,..\, s + l - 1], e) =$ true then
15                 $R \leftarrow R \cup \langle D[s \,..\, s + l - 1], e \rangle$;
16         else
17             for $m = |e| - \tau$ to $|e| + \tau$ do
18                 if Verify$(D[s \,..\, s + m - 1], e) =$ true then
19                     $R \leftarrow R \cup \langle D[s \,..\, s + m - 1], e \rangle$;
20 return $R$
```

4.2.3 *Cost Analysis*

Now we can analyze the run-time complexity of our algorithm on a document D.

Let C_{enum} and C_{probe} be the average cost of enumerating a variant and probing the inverted index, respectively; let C_{verify} be the average cost of verifying a candidate pair; The size of length l string's τ-variant family is $v(l, \tau) = \sum_{i=1}^{\tau} \binom{l}{i} = O(l^\tau)$.

The total cost of the algorithm is

$$C = (|D| - L_{\min} + \tau) \cdot (C_1 + C_2) \qquad (1)$$

where

$$C_1 = (C_{\text{enum}} + C_{\text{probe}}) \cdot \sum_{i=L_{\min}}^{l_p - 1} v(i, \tau) + cand_{\text{tiny}} \cdot C_{\text{verify}}$$

$$C_2 = (C_{\text{enum}} + C_{\text{probe}}) \cdot v(l_p, \tau) + cand_{\text{other}} \cdot (2\tau + 1) \cdot C_{\text{verify}}$$

In the above formula, we distinguish the cost made by matching "tiny" entities (whose length is smaller than l_p) with the cost of matching "other" entities. The main difference is that for the latter, query segment instantiation is required and $2\tau + 1$ verification needs to be performed for each candidate entity. $cand_{\text{tiny}}$ and $cand_{\text{other}}$ represents the average number of candidates in each case.

C_{probe} is typically $O(1)$ with an appropriate value of l_p. This is empirically verified (See Table 1).

Table 1: Postings List Length ($\tau = 1$, $l_p = 10$)

Dataset	Average Length
DBLP	1.39
GENE	2.22
CONLL	1.18

We shall make the following simplifying assumptions in order to derive an asymptotic bound of the overall cost: (a) We assume $C_1 \ll C$, and thus focus on the asymptotic bound on C_2. (b) The length l_p prefixes of the query segment and the partition variations are randomly generated strings. (c) We assume $l_p \leq c \cdot |\Sigma|$ for some constant c. The important observation is that all the candidates that have 1-variant match with a string s must have edit distance no more than 2 from s. The probability of a prefix from an entity having 1-variant match with the prefix of a query segment is

$$Pr = \frac{O(l_p^2 |\Sigma|^2)}{\sum_{i=-1}^{1} |\Sigma|^{l_p + i}} = O\left(\frac{l_p^2}{|\Sigma|^{l_p - 1}}\right) = O\left(|\Sigma|^{-(l_p - 3)}\right)$$

Then $cand_{\text{other}}$ can be estimated as $Pr \cdot O(1) \cdot N$, where N is the number of entities in the dictionary.

Therefore, the overall cost of the algorithm can be estimated as

$$C \approx |D| \cdot O\left(l_p^\tau + \frac{N}{|\Sigma|^{-(l_p - 3)}} \cdot \tau\right) \qquad (2)$$

Hence we can see the cost increases with τ. The trend with respect to l_p is more complicated, as a large l_p value will increase the enumeration cost yet reduce the verification cost. Therefore, we might expect to see an optimal l_p value that minimizes the overall cost. These are indeed what we observed in the experiment (See Section 6.2).

4.3 Reduce the Amount of Enumeration

A frequent operation during the matching process is to enumerate the 1-variant family of a length l_p string, and probe each variant against the inverted index. A problem with this approach is that some of the enumeration is unnecessary. For example, consider enumerating the 1-variants of the string [`abcdef`] from left to right. Suppose we know that no variant starts with `abc` in the index. The naïve enumeration algorithm will still enumerate other three 1-variants that contain `abc`.

We design the following data structure and algorithm to reduce the amount of enumeration by leveraging the above observation on both the prefixes and suffixes. We employ a parameter l_{pp} set to $\left\lceil \frac{l_p}{2} \right\rceil$. We record in a data structure M_p (alternatively, M_s) all the l_{pp}-prefixes (alternatively, $(l_p - l_{pp})$-suffixes) that appeared in the 1-variants in the inverted index. We can probe M_p and M_s to determine if the prefix and suffix of the current query segment appears or not. There are four possible cases and they are handled as shown in the following table.

Prefix Match	Suffix Match	Action
true	**true**	enumerate all 1-variants of $q[1 \,..\, l_p]$
false	**false**	discard q as there is no match
false	**true**	enumerate all 1-variants of $q[1 \,..\, l_{pp}]$
true	**false**	enumerate all 1-variants of $q[(l_{pp} + 1) \,..\, l_p]$

We use a simplified version of Bloom Filter to implement M_p and M_s. This is a trade-off between space and time. Note that a Bloom Filter guarantees that there is no false positives (meaning, in our problem context, if a prefix/suffix appears in a 1-variant, it will always be reported as existent). This only affects the efficiency but not the correctness of the algorithm.

5. VERIFICATION

Verifying if a candidate pair satisfies τ edit distance constrain can easily be a bottleneck for the whole system. This is mainly due to the large amount of candidates and the costly $O(nm)$ running cost of the edit distance calculation.

We briefly outline several improvements for our problem below and defer the details to the full version of the paper.

- We extend the content-based filtering in [39] to the entire strings. We also minimize its filtering cost by exploiting the temporal locality of the query segments.
- We use an efficient thresholded edit distance computation procedure due to Ukkonen [36], which has a $O(\tau \cdot \min(n, m))$ space and time complexity. A heuristic is to verify the two strings in reverse order.

6. EXPERIMENTS

In this section, we report experimental results and our analysis.

6.1 Experiment Setup

The following algorithms are used in the experiment.

FastSS is a neighborhood generation based algorithm [33]. We actually improve the algorithm by using our verification procedure rather than invoking the normal edit distance calculation.

QGRAM is a q-gram-based algorithm entity extraction algorithm that incorporates the state-of-the-art filtering techniques, including the count, position and length filtering [16], prefix filtering [12], and location-based and content-based filtering [39]. Therefore, it is likely that this implementation is more efficient than the one used for comparison with the FastSS algorithm in [33].

We use different q values to index entities with different lengths. We use the following values which achieve the best performance on our datasets:

$$\begin{cases} q = 2 & \text{, when } |e| \in [1, 13] \\ q = 3 & \text{, when } |e| \in [12, 20] \\ q = 4 & \text{, when } |e| \in [18, +\infty) \end{cases}$$

Note that adjacent ranges overlap each other so that we won't miss any result.

NGPP is our proposed algorithm, abbreviated from Neighborhood Generation with partitioning and prefix-based pruning.

All algorithms are implemented as in-memory algorithms, with the inputs loaded into memory before they were run.

All experiments were carried out on a PC with Intel Xeon X3220 @ 2.40GHz CPU and 4GB RAM. The operating system is Debian 4.1.1-21. All algorithms were implemented in C++ and compiled using GCC 4.2.3 with `-O3` flag.

We used three publicly available datasets, two of which were taken from existing named entity recognition tasks.

DBLP We extract author names from the first 10% of the DBLP database[6] as the dictionary. We then use the last 10% records as documents, where each record is a concatenation of a publication and its authors' names.

GENE We used the Gene/Protein lexicon generated from MEDLINE documents by [34].[7] We sampled 33% of the 1M entities as our dictionary. We use the TREC-9 Filtering Track Collections dataset[8] as documents. It contains 350K references from the MEDLINE database. We extract and concatenate author, title, and abstract fields from the first 10K references.

CONLL We use the entities from the shared task of Conference on Computational Natural Language Learning 2003.[9] It contains more than 8,000 entities including personal names, locations, and organization. We use the Reuters dataset[10] as documents. It contains a collection of 20K pieces of news from Reuters Ltd.

Some statistics of the datasets are shown in Table 2 and Figures 2(a)–2(b).

Table 2: Datasets Statistics

Dataset	N	*avg_len*	$\lvert\Sigma\rvert$	Comment
DBLP-DICT	107,810	14.5	69	author name
DBLP-DOC	87,352	104.7	93	author, title
GENE-DICT	381,417	22.4	60	gene/protein name
GENE-DOC	10,000	870.0	62	author, title, abstract
CONLL-DICT	8,215	12.6	76	person, location, organization
CONLL-DOC	19,042	819.0	89	news article

We test all the algorithms on each dataset with τ between 1 and 3, which covers many important applications [20]. In order to avoid excessive numbers of meaningless matches (esp. when $\frac{\tau}{|e|}$ is large), we enforce a local edit distance threshold τ' when finding approximate matches for entities with different lengths, as shown below:

$$\begin{cases} \tau' = \min(1, \tau) & \text{, when } |e| \in [1, 5] \\ \tau' = \min(2, \tau) & \text{, when } |e| \in [6, 11] \\ \tau' = \tau & \text{, when } |e| \in [12, L_{\max}] \end{cases}$$

We also further restrict that both ends of the query segment must be a separator. This ensures all the algorithms can finish within reasonable amount of time.

We record the following measures: (a) the number of variants enumerated by the FastSS and NGPP algorithms. (b) the numbers of the candidate pairs before entering the verification procedure and before invoking the thresholded edit distance computation by all the algorithms. We name them CAND-1 and CAND-2, respectively. For the FastSS and NGPP algorithms, the reduction from CAND-1 to CAND-2 is due to the content-based filtering (See Section 5). For the QGRAM algorithm, the reduction comes from the use of count filtering and content-based filtering. (c) the overall running time for all the algorithms.

6.2 Effect of Prefix Length

The only parameter in the NGPP algorithm is the prefix length, l_p. We ran the algorithm on all the datasets with different l_p values. Figures 2(c)–2(f) show various measures on the DBLP dataset for $\tau \in [1, 3]$. Results on other datasets are similar.

We observe that

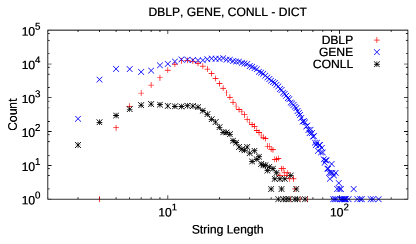
[6] `http://www.informatik.uni-trier.de/~ley/db`

[7] `ftp://ftp.ncbi.nlm.nih.gov/pub/tanabe/Gene.Lexicon.gz/`

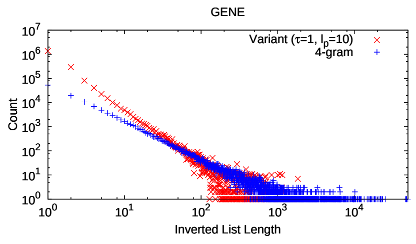
[8] `http://trec.nist.gov/data/t9_filtering.html`

[9] `http://www.cnts.ua.ac.be/conll2003/ner/`

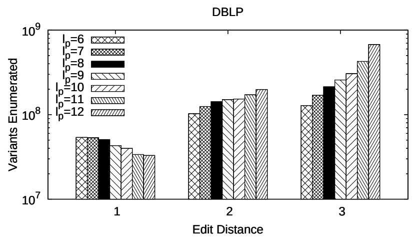
[10] `http://www.daviddlewis.com/resources/testcollections/reuters21578/`



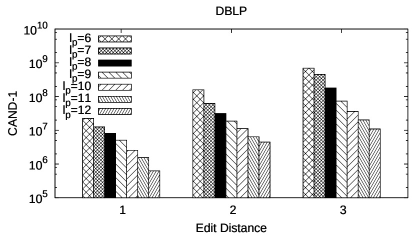
(a) Entity Length Distribution, DBLP (in red), GENE (in blue) and CONLL (in black)



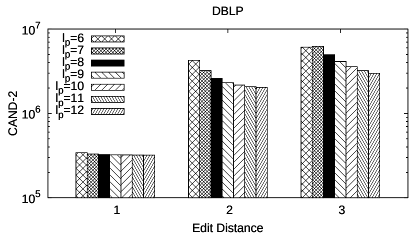
(b) Inverted List Length Distribution, GENE



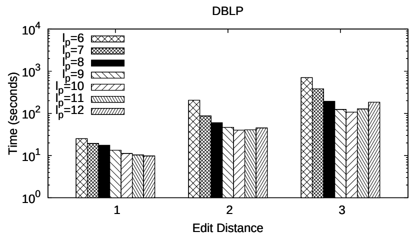
(c) DBLP, Variants Enumerated



(d) DBLP, CAND-1



(e) DBLP, CAND-2



(f) DBLP, Running Time

Figure 2: Dataset Statistics and Experiment Results - I

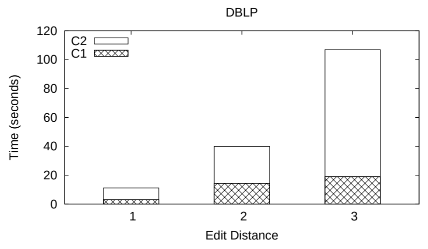


Figure 3: Validating the Cost Model (DBLP)

- Overall, the best l_p for the DBLP dataset is 10. Longer l_p (11 or 12) is competitive for $\tau = 1$, but not the best choice for larger edit distance thresholds.
- When τ is fixed, a general trend is that the running time will first decrease and then increase when we move towards a long l_p (Figure 2(f)). This is expected from our cost model in Equation (2). In fact, the number of enumeration (Figure 2(c)) directly corresponds to the $O(l_p^\tau)$ term, and it increases quickly with l_p (esp. when $\tau > 1$). In the meanwhile, a large l_p value helps to reduce the second term $O(N\tau \cdot |\Sigma|^{-(l_p-3)})$, and this is reflected in decrease of CAND-1 size (Figure 2(d)). The only exception is the number of enumeration when $\tau = 1$ decreases with l_p (and this is also observed on other datasets). The main reason is due to the optimization in Section 4.3. It however cannot contain the super-linear growth of number of enumerations when $\tau > 1$.
- Figure 2(e) shows that the content-based filtering before the final edit distance verification is very effective. It can reduce the CAND-2 size up to an order of magnitude smaller than the corresponding CAND-1 size. The reduction effect is especially substantial for large τ.

Recall that an assumption underlying the cost model when deriving Equation (2) from Equation 1 is that C_2 cost dominates the total cost. We plot both C_1 and C_2 costs for DBLP dataset with different τ in Figure 3. It can be observed that C_2 indeed accounts for 70%–80% of the total running time.

In the rest of the experiments, we use the best l_p values for the NGPP algorithm, i.e., $l_p = 10$ for DBLP and GENE, and $l_p = 7$ for CONLL.

6.3 Comparison with FastSS

Table 3: Comparison with FastSS on DBLP

Algorithm	Variants Enumerated ($\times 10^7$)	CAND-1 ($\times 10^5$)	CAND-2 ($\times 10^5$)	Time (secs)
FastSS ($\tau = 1$)	34	3	3	97.9
FastSS ($\tau = 2$)	750	21	20	2642.8
NGPP ($\tau = 1$)	4	25	3	11.1
NGPP ($\tau = 2$)	15	110	22	40.0
NGPP ($\tau = 3$)	31	361	36	106.9

We run the NGPP and FastSS algorithms on the DBLP dataset and report the number of variants enumerated, CAND-1 size, and total running time in Table 3.

We can see that the number of variants enumerated grows rapidly for the FastSS algorithm. The growth ratio between $\tau = 2$ and $\tau = 1$ is 22.1. On the contrary, NGPP has a much smaller growth ratio of 3.8. The number of enumerated variants for NGPP is only 2% of that of FastSS when $\tau = 2$; this contributes substantially to the huge difference in running time between the two algorithms. Although FastSS produces fewer CAND-1 candidates than NGPP does, the content-based filtering effectively reduces the false positive candidates such the CAND-2 sizes for both algorithms are similar to each other.

We didn't show the results for $\tau = 3$ as FastSS didn't stop after 12 hours. The reason why FastSS performs significantly worse when τ increases is because its query cost is proportional to the variants it enumerates, which is $O(m^\tau)$. Due to the existence of long entities in the dictionary, the query processing cost is dominated by attempts to match those long entities, and hence the observed exponential growth in running time.

To conclude, FastSS is only competitive when $\tau = 1$ (FastSS is actually faster than QGRAM for $\tau = 1$ for all three datasets). The partitioning and prefix-based pruning techniques employed in our NGPP algorithm drastically improve the performance by eliminating most of the enumeration.

6.4 Comparison with the q-gram-based Method

We compare the performance of the NGPP algorithm and the QGRAM algorithm on three datasets and plot the results in Figures 4(a)–4(i).

Candidate Size. The sizes of CAND-1 for both algorithms are shown in Figures 4(a)–4(c). NGPP produces fewer CAND-1s than QGRAM by more than one order of magnitude. The difference is most significant on GENE dataset, where CAND-1 size produced by QGRAM is 42 times that produced by NGPP. This is because the variants used in NGPP are longer than the q-grams used in QGRAM, and therefore more selective than q-grams.

We plot the size of CAND-2 for both algorithm in Figures 4(d)–4(f). The size of real result is also shown to lower bound the CAND-2 size. Several observations can be made.

- QGRAM produces larger CAND-2 than NGPP, with an average of 4 times on the three datasets under different settings. This is expected as the size of CAND-1 produced by NGPP is much less than QGRAM.
- content-based filtering is quite effective in removing false positives for both algorithms. About 98% candidates pairs are pruned by content-based filtering for QGRAM, and the percentage is 92% for NGPP. The reduction effect is generally more substantial with large τ.

Running Time. Running times for both algorithms on three datasets are shown in Figures 4(g)–4(i). The general trend is that the running time grows exponentially with the increase of τ. NGPP outperforms QGRAM by a large margin for all parameter settings. On GENE, the speed-up can be up to 25x.

Two main factors influence the running time:

1. There is no shared computation between query segments for QGRAM. It first enumerates all possible lengths of query segments, and then searches for the candidate entities. In contrast, NGPP searches for the candidate entities first, and then configures the query segments according to the length of the candidate entities. Hence the exhaustive iteration of query segments can be avoided.
2. The number of candidates generated by NGPP is far less than that generated by QGRAM. The total verification cost for NGPP is therefore less than QGRAM, and this contributes substantially to the difference in the running time.

6.5 Scalability against Dictionary Sizes

Since our algorithm processes a document at a time, it scales linearly with the number of documents to be matched. Below we consider its scalability with respect to the size of the dictionary. We sample the dictionary of the DBLP dataset from 20% to 100% and run the NGPP algorithm.

Figures 4(j)–4(l) show the results with respect to different τ. We observe that both candidate size and running time change approximately linearly with the varying dictionary size for all τ values. Note also that the increase in running time is slower than the increase in real result size.

6.6 Index Size

Figure 5 shows the total number of index entries in the inverted index (for q-grams or variants) by the three algorithms on the CONLL dataset.

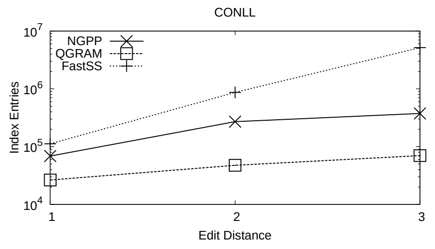


Figure 5: Index Size (CONLL)

The general trend is that the index sizes of all three algorithms increase with the increase of τ. According to the theoretic analysis, FastSS shall have exponential growth with the increase of τ, NGPP is quadratic, and QGRAM is linear.[11] This is verified empirically in the figure.

The index size of FastSS is the largest, followed by NGPP, and QGRAM is the most space-efficient algorithm. When $\tau = 3$, the indexed entries for FastSS is 15 times as large as that of NGPP, the latter being 5 times as large as that of QGRAM. Neighborhood generation-based algorithms need more space than q-gram-based method since they enumerate edit operations on the entities, while q-gram-based algorithms only index the q-grams that are located in the prefixes.

7. RELATED WORK

Approximate string matching is a well-studied area. We refer readers to survey papers [26, 28] and we will focus on recent work in related fields.

Entity Matching and Approximate Dictionary Matching. [15] proposes to use dictionary to aid name entity recognition tasks and [10] proposed efficient batch top-k matching algorithms under this setting. Subsequent work includes [8, 1]. These work only considers token-based similarity functions.

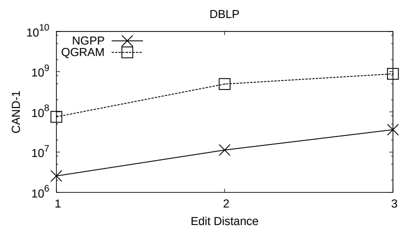
[27] is a closely related work aiming at matching a document against a dictionary of personal names. However, its query processing method is mainly based on evaluating edit distance function on all the entities simultaneously, and thus the query processing time grows linearly with the number of entities.

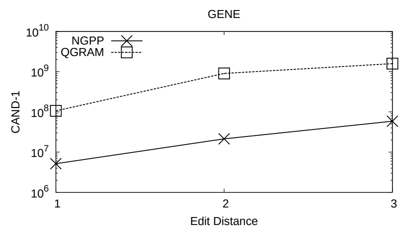
[10] proposed to build inverted index on tokens and focuses on computational sharing among subsequent queries.

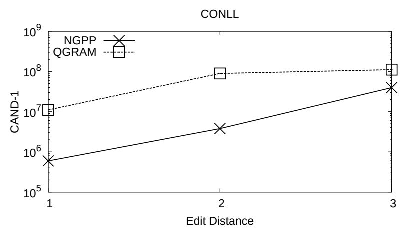
All existing work on approximate string match all uses fixed-length q-grams except the VGRAM in [22, 41]. The variants we generated and indexed can also be deemed as variable-length q-grams. However, the two methods are fundamentally different, with different similarity search methods. VGRAM-based approaches also need to access both the dictionary and the documents to select an appropriate set of variable length q-grams, thus they cannot deal with applications where documents are streaming in.

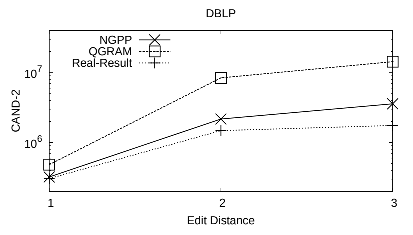
Neighborhood Generation Methods. Neighborhood generation method was used for approximate string matching in [25]. The size of the neighborhood is $O(m^\tau|\Sigma|^\tau)$ and this makes it only applicable to small alphabet sizes and small error threshold [37]. [33] proposes to generate the *deletion neighborhood* for both query and text, thus improv-

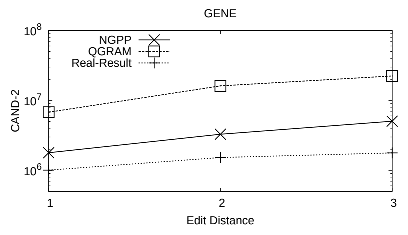
[11]Because prefix filtering on q-grams requires at most $q\tau + 1$ q-grams to be indexed.

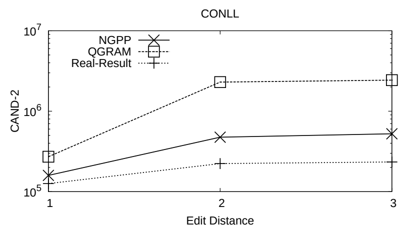

(a) DBLP, CAND-1

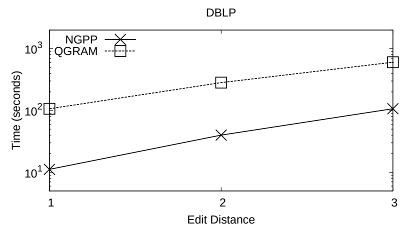

(b) GENE, CAND-1

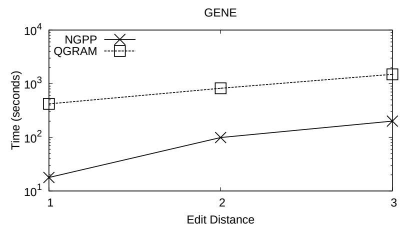

(c) CONLL, CAND-1

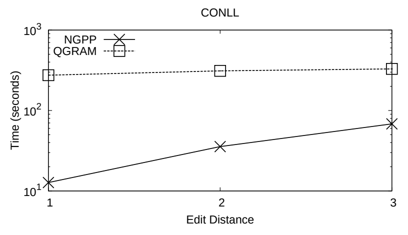

(d) DBLP, CAND-2

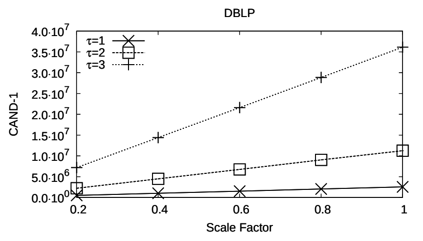

(e) GENE, CAND-2

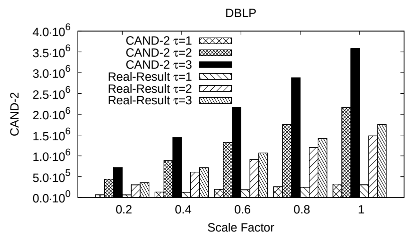

(f) CONLL, CAND-2


(g) DBLP, Running Time


(h) GENE, Running Time


(i) CONLL, Running Time


(j) DBLP, CAND-1


(k) DBLP, CAND-2

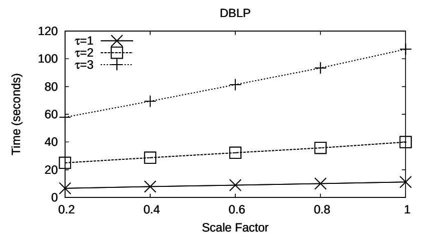

(l) DBLP, Running Time

Figure 4: Experiment Results - II

ing the neighborhood size to $O(m^\tau)$. In this work, we reduce the bound further down to $O(l_p\tau^2)$, thus making this neighborhood generation approach practical for typical entity extraction tasks.

Another approach to circumvent the large neighborhood size is to generate only a subset of all possible variants, hence some true matches may be missed. A notable example is the gene and protein named entity recognition problem, where domain knowledge is used and variants are generated according to a set of hand-crafted rules [35, 14]. Another example is the permuted lexicon method in [42].

Similarity Selection, Joins, and Record Linkage. If we extract all the possible query segments in a relation R, then our problem can be formulated by a near duplicate detection task and can be solved by a similarity join between R and the dictionary E. There has been progress in similarity joins [32, 12, 5, 40, 39, 22, 41, 23], similarity selection [21, 17], and selectivity estimation [19, 20, 24, 18].

Different similarity or distance measures were proposed in the area of record linkage. A recent experimental study of their relative effectiveness is presented in [9]. Some new types of record recently studied include utilizing group information [30], combining multiple similarity functions [11], leveraging aggregate constraints [13], and considering string transformation rules [3].

8. CONCLUSIONS AND FUTURE WORK

We study the problem of dictionary-based entity extraction with edit distance constraints in this paper. It can increase the recall of the system by capturing small errors that are likely to be missed by existing methods. Our matching problem is technically challenging as existing approaches based on q-gram filtering have poor performance due to the existence of many short entities in the dictionary. Our proposed solution is based on an improvement neighborhood generation filtering technique. We have successfully reduce the size of the neighborhood that needs to be generated and indexed from $O(m^\tau)$ to $O(l_p\tau^2)$. In addition, we propose an efficient query processing algorithm that avoids examining query segment and entity pairs that are not in the final matching results. We have also optimized the algorithm to share computation and avoid unnecessary variant enumeration. Extensive experiments have been conducted on several named entity recognition datasets. The proposed algorithm outperforms alternative approaches by up to an order of magnitude.

A future work is to study the trade-offs between the precision and recall of the approximate entity matching. We will work on more "dirty" data (e.g., blogs or forum posts) and consider leveraging existing machine learning-based NER techniques.

Acknowledgement. The authors would like to thank the anonymous reviewers for their insightful comments.
Wei Wang is supported by ARC Discovery Grants DP0987273 and DP0881779. Xuemin Lin is supported by Google Research Award and ARC Discovery Grants DP0987557, DP0881035 and DP0666428. Chengqi Zhang is supported by ARC Discovery Grant DP0667060.

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] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *SIGIR*, pages 273–280, 2004.
[3] A. Arasu, S. Chaudhuri, K. Ganjam, and R. Kaushik. Incorporating string transformations in record matching. In *SIGMOD Conference*, pages 1231–1234, 2008.
[4] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.
[5] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.
[6] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[7] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Sys.*, 18(5):16–23, 2003.
[8] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, pages 805–818, 2008.
[9] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In *SIGMOD Conference*, pages 353–364, 2007.
[10] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.
[11] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, pages 327–338, 2007.
[12] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.
[13] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik. Leveraging aggregate constraints for deduplication. In *SIGMOD Conference*, pages 437–448, 2007.
[14] A. M. Cohen. Unsupervised gene/protein entity normalization using automatically extracted dictionaries. In *Proceedings of the BioLINK2005 Workshop*, 2005.
[15] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *KDD*, pages 89–98, 2004.
[16] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.
[17] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, pages 267–276, 2008.
[18] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *PVLDB*, 1(1):201–212, 2008.
[19] L. Jin and C. Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.
[20] H. Lee, R. T. Ng, and K. Shim. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB*, pages 195–206, 2007.
[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.
[22] C. Li, B. Wang, and X. Yang. VGRAM: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, 2007.
[23] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, pages 1111–1120, 2008.
[24] A. Mazeika, M. H. Böhlen, N. Koudas, and D. Srivastava. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.*, 32(2):12, 2007.
[25] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.
[26] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.
[27] G. Navarro, R. A. Baeza-Yates, and J. M. A. Arcoverde. Matchsimile: a flexible approximate matching tool for searching proper name. *JASIST*, 54(1):3–15, 2003.
[28] G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.
[29] R. Navigli. Word sense disambiguation: A survey. *ACM Comput. Surv.*, 41(2), 2009.
[30] B.-W. On, N. Koudas, D. Lee, and D. Srivastava. Group linkage. In *ICDE*, pages 496–505, 2007.
[31] M. Pasca. Acquisition of categorized named entities for web search. In *CIKM*, pages 137–145, 2004.
[32] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.
[33] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007.
[34] L. Tanabe and W. J. Wilbur. Generation of a large gene/protein lexicon by morphological pattern analysis. *Journal of Bioinformatics and Computational Biology*, 1(4):1–16, 2004.
[35] Y. Tsuruoka and J. ichi Tsujii. Improving the performance of dictionary-based approaches in protein name recognition. *Journal of Biomedical Informatics*, 37(6):461–470, 2004.
[36] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.
[37] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.
[38] J. Wang, Z. Li, C. Cai, and Y. Chen. Assessment of approximate string matching in a biomedical text retrieval problem. *Computers in Biology and Medicine*, 35(8):717–724, 2005.
[39] 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.
[40] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.
[41] 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.
[42] J. Zobel and P. W. Dart. Finding approximate matches in large lexicons. *Softw., Pract. Exper.*, 25(3):331–345, 1995.