Simple and Efficient Algorithm
for Approximate Dictionary Matching

Naoaki Okazaki
University of Tokyo
okazaki@is.s.u-tokyo.ac.jp

Jun'ichi Tsujii
University of Tokyo
University of Manchester
National Centre for Text Mining
tsujii@is.s.u-tokyo.ac.jp

Abstract

This paper presents a simple and efficient algorithm for approximate dictionary matching designed for similarity measures such as cosine, Dice, Jaccard, and overlap coefficients. We propose this algorithm, called *CPMerge*, for the τ-overlap join of inverted lists. First we show that this task is solvable exactly by a τ-overlap join. Given inverted lists retrieved for a query, the algorithm collects fewer candidate strings and prunes unlikely candidates to efficiently find strings that satisfy the constraint of the τ-overlap join. We conducted experiments of approximate dictionary matching on three large-scale datasets that include person names, biomedical names, and general English words. The algorithm exhibited scalable performance on the datasets. For example, it retrieved strings in 1.1 ms from the string collection of Google Web1T unigrams (with cosine similarity and threshold 0.7).

1 Introduction

Languages are sufficiently flexible to be able to express the same meaning through different diction. At the same time, inconsistency of surface expressions has persisted as a serious problem in natural language processing. For example, in the biomedical domain, *cardiovascular disorder* can be described using various expressions: *cardiovascular diseases*, *cardiovascular system disorder*, and *disorder of the cardiovascular system*. It

is a nontrivial task to find the entry from these surface expressions appearing in text.

This paper addresses *approximate dictionary matching*, which consists of finding all strings in a string collection V such that they have similarity that is no smaller than a threshold α with a query string x. This task has a broad range of applications, including spelling correction, flexible dictionary look-up, record linkage, and duplicate detection (Henzinger, 2006; Manku et al., 2007).

Formally, the task obtains a subset $\mathcal{Y}_{x,\alpha} \subseteq V$,

$$\mathcal{Y}_{x,\alpha} = \{y \in V \mid \text{sim}(x, y) \geq \alpha\}, \qquad (1)$$

where $\text{sim}(x, y)$ presents the similarity between x and y. A naïve solution to this task is to compute similarity values $|V|$ times, i.e., between x and every string $y \in V$. However, this solution is impractical when the number of strings $|V|$ is huge (e.g., more than one million).

In this paper, we present a simple and efficient algorithm for approximate dictionary matching designed for similarity measures such as cosine, Dice, Jaccard, and overlap coefficients. Our main contributions are twofold.

1. We show that the problem of approximate dictionary matching is solved exactly by a *τ-overlap join* (Sarawagi and Kirpal, 2004) of inverted lists. Then we present *CPMerge*, which is a simple and efficient algorithm for the τ-overlap join. In addition, the algorithm is easily implemented.

2. We demonstrate the efficiency of the algorithm on three large-scale datasets with person names, biomedical concept names,

and general English words. We compare the algorithm with state-of-the-art algorithms, including Locality Sensitive Hashing (Ravichandran et al., 2005; Andoni and Indyk, 2008) and DivideSkip (Li et al., 2008). The proposed algorithm retrieves strings the most rapidly, e.g., in 1.1 ms from Google Web1T unigrams (with cosine similarity and threshold 0.7).

2 Proposed Method

2.1 Necessary and sufficient conditions

In this paper, we assume that the *features* of a string are represented arbitrarily by a set. Although it is important to design a string representation for an accurate similarity measure, we do not address this problem: our emphasis is *not on designing a better representation for string similarity* but *on establishing an efficient algorithm.*

The most popular representation is given by *n-grams*: all substrings of size n in a string. We use trigrams throughout this paper as an example of string representation. For example, the string "methyl sulphone" is expressed by 17 elements of letter trigrams[1], {`$$m`, `$me`, `met`, `eth`, `thy`, `hyl`, `yl_`, `l_s`, `_su`, `sul`, `ulp`, `lph`, `pho`, `hon`, `one`, `ne$`, `e$$`}. We insert two $s before and after the string to denote the start or end of the string. In general, a string x consisting of $|X|$ letters yields $(|x| + n - 1)$ elements of n-grams. We call $|x|$ and $|X|$ the *length* and *size*, respectively, of the string x.

Let X and Y denote the feature sets of the strings x and y, respectively. The cosine similarity between the two strings x and y is,

$$\text{cosine}(X, Y) = \frac{|X \cap Y|}{\sqrt{|X||Y|}}. \quad (2)$$

By integrating this definition with Equation 1, we obtain the necessary and sufficient condition for

---

[1] In practice, we attach ordinal numbers to n-grams to represent multiple occurrences of n-grams in a string (Chaudhuri et al., 2006). For example, the string "prepress", which contains two occurrences of the trigram `pre`, yields the set {`$$p`#1, `$pr`#1, `pre`#1, `rep`#1, `epr`#1, `pre`#2, `res`#1, `ess`#1, `ss$`#1, `s$$`#1}.

Table 1: Conditions for each similarity measure

Measure	min $|Y|$	max $|Y|$	$\tau (= \min |X \cap Y|)$
Dice	$\frac{\alpha}{2-\alpha}|X|$	$\frac{2-\alpha}{\alpha}|X|$	$\frac{1}{2}\alpha(|X| + |Y|)$
Jaccard	$\alpha|X|$	$|X|/\alpha$	$\frac{\alpha(|X|+|Y|)}{1+\alpha}$
Cosine	$\alpha^2|X|$	$|X|/\alpha^2$	$\alpha\sqrt{|X||Y|}$
Overlap	—	—	$\alpha \min\{|X|, |Y|\}$

approximate dictionary matching,

$$\left\lceil \alpha\sqrt{|X||Y|} \right\rceil \leq |X \cap Y| \leq \min\{|X|, |Y|\}. \quad (3)$$

This inequality states that two strings x and y must have at least $\tau = \left\lceil \alpha\sqrt{|X||Y|} \right\rceil$ features in common. When ignoring $|X \cap Y|$ in the inequality, we have an inequality about $|X|$ and $|Y|$,

$$\left\lceil \alpha^2|X| \right\rceil \leq |Y| \leq \left\lfloor \frac{|X|}{\alpha^2} \right\rfloor \quad (4)$$

This inequality presents the search range for retrieving similar strings; that is, we can ignore strings whose feature size is out of this range. Other derivations are also applicable to similarity measures, including Dice, Jaccard, and overlap coefficients. Table 1 summarizes the conditions for these similarity measures.

We explain one usage of these conditions. Let query string $x =$ "methyl sulphone" and threshold for approximate dictionary matching $\alpha = 0.7$ with cosine similarity. Representing the strings with letter trigrams, we have the size of x, $|X| = 17$. The inequality 4 gives the search range of $|Y|$ of the retrieved strings, $9 \leq |Y| \leq 34$. Presuming that we are searching for strings of $|Y| = 16$, we obtain the necessary and sufficient condition for the approximate dictionary matching from the inequality 3, $\tau = 12 \leq |X \cap Y|$. Thus, we need to search for strings that have at least 12 letter trigrams that overlap with X. When considering a string $y =$ "methyl sulfone", which is a spelling variant of y ($ph \rightarrow f$), we confirm that the string is a solution for approximate dictionary matching because $|X \cap Y| = 13 \, (\geq \tau)$. Here, the actual similarity is $\text{cosine}(X, Y) = 13/\sqrt{17 \times 16} = 0.788$ $(\geq \alpha)$.

2.2 Data structure and algorithm

Algorithm 1 presents the pseudocode of the approximate dictionary matching based on Table 1.

```
Input: V: collection of strings
Input: x: query string
Input: α: threshold for the similarity
Output: Y: list of strings similar to the query
1  X ← string_to_feature(x);
2  Y ←[];
3  for l ← min_y(|X|, α) to max_y(|X|, α) do
4  |    τ ← min_overlap(|X|, l, α);
5  |    R ← overlapjoin(X, τ, V, l);
6  |    foreach r ∈ R do  append r to Y;
7  end
8  return Y;
```

Algorithm 1: Approximate dictionary matching.

```
Input: X: array of features of the query string
Input: τ: minimum number of overlaps
Input: V: collection of strings
Input: l: size of target strings
Output: R: list of strings similar to the query
1  M ← {};
2  R ← [];
3  foreach q ∈ X do
4  |    foreach i ∈ get(V, l, q) do
5  |    |    M[i] ← M[i] + 1;
6  |    |    if τ ≤ M[i] then
7  |    |    |    append i to R;
8  |    |    end
9  |    end
10 end
11 return R;
```

Algorithm 2: AllScan algorithm.

Given a query string x, a collection of strings V, and a similarity threshold α, the algorithm computes the size range (line 3) given by Table 1. For each size l in the range, the algorithm computes the minimum number of overlaps τ (line 4). The function overlapjoin (line 5) finds similar strings by solving the following problem (τ-*overlap join*): given a list of features of the query string X and the minimum number of overlaps τ, enumerate strings of size l in the collection V such that they have at least τ feature overlaps with X.

To solve this problem efficiently, we build an inverted index that stores a mapping from the features to their originating strings. Then, we can perform the τ-overlap join by finding strings that appear at least τ times in the inverted lists retrieved for the query features X.

Algorithm 2 portrays a naïve solution for the τ-overlap join (AllScan algorithm). In this algorithm, function $get(V, l, q)$ returns the inverted list of strings (of size l) for the feature q. In short, this algorithm scans strings in the inverted lists retrieved for the query features X, counts the frequency of occurrences of every string in the inverted lists, and returns the strings whose frequency of occurrences is no smaller than τ.

This algorithm is inefficient in that it scans all strings in the inverted lists. The number of scanned strings is large, especially when some query features appear frequently in the strings, e.g., 's$$' (words ending with 's') and 'pre' (words with substring 'pre'). To make matters worse, such features are too common for characterizing string similarity. The AllScan algorithm

is able to maintain numerous candidate strings in M, but most candidates are not likely to qualified because they have few overlaps with X.

To reduce the number of the candidate strings, we refer to *signature-based algorithms* (Arasu et al., 2006; Chaudhuri et al., 2006):

Property 1 *Let there be a set (of size h) X and a set (of any size) Y. Consider any subset $Z \subseteq X$ of size $(h - \tau + 1)$. If $|X \cap Y| \geq \tau$, then $Z \cap Y \neq \phi$.*

We explain one usage of this property. Let query string $x =$ "methyl sulphone" and its trigram set X be features (therefore, $|X| = h = 17$). Presuming that we seek strings whose trigrams are size 16 and have 12 overlaps with X, then string y must have at least one overlap with any subset of size 6 ($= 17 - 12 + 1$) of X. We call the subset *signatures*. The property leads to an algorithmic design by which we obtain a small set of candidate strings from the inverted lists for signatures, ($|X| - \tau + 1$) features in X, and verify whether each candidate string satisfies the τ overlap with the remaining ($\tau - 1$) n-grams.

Algorithm 3 presents the pseudocode employing this idea. In line 1, we arrange the features in X in ascending order of the number of strings in their inverted lists. We denote the k-th element in the ordered features as X_k ($k \in \{0, ..., |X| - 1\}$), where the index number begins with 0. Based on this notation, X_0 and $X_{|X|-1}$ are the most uncommon and the most common features in X, respectively.

In lines 2–7, we use ($|X| - \tau + 1$) features

```
Input: X: array of features of the query string
Input: τ: minimum number of overlaps
Input: V: collection of strings
Input: l: size of target strings
Output: R: list of strings similar to the query
 1  sort elements in X by order of |get (V, l, X_k) |;
 2  M ← {};
 3  for k ← 0 to (|X| − τ) do
 4  |   foreach s ∈ get (V, l, X_k) do
 5  |   |   M[s] ← M[s] + 1;
 6  |   end
 7  end
 8  R ← [];
 9  for k ← (|X| − τ + 1) to (|X| − 1) do
10  |   foreach s ∈ M do
11  |   |   if bsearch(get (V, l, X_k), s) then
12  |   |   |   M[s] ← M[s] + 1;
13  |   |   end
14  |   |   if τ ≤ M[s] then
15  |   |   |   append s to R;
16  |   |   |   remove s from M;
17  |   |   else if M[s] + (|X| − k − 1) < τ then
18  |   |   |   remove s from M;
19  |   |   end
20  |   end
21  end
22  return R;
```

Algorithm 3: CPMerge algorithm.

$X_0, ..., X_{|X|-\tau}$ to generate a compact set of candidate strings. The algorithm stores the occurrence count of each string s in $M[s]$. In lines 9–21, we increment the occurrence counts if each of $X_{|X|-\tau+1}, ..., X_{|X|-1}$ inverted lists contain the candidate strings. For each string s in the candidates (line 10), we perform a binary search on the inverted list (line 11), and increment the overlap count if the string s exists (line 12). If the overlap counter of the string reaches τ (line 14), then we append the string s to the result list R and remove s from the candidate list (lines 15–16). We prune a candidate string (lines 17–18) if the candidate is found to be unreachable for τ overlaps even if it appears in all of the unexamined inverted lists.

3 Experiments

We report the experimental results of approximate dictionary matching on large-scale datasets with person names, biomedical names, and general English words. We implemented various systems of approximate dictionary matching.

- **Proposed**: CPMerge algorithm.

- **Naive**: Naïve algorithm that computes the cosine similarity $|V|$ times for every query.
- **AllScan**: AllScan algorithm.
- **Signature**: CPMerge algorithm without pruning; this is equivalent to Algorithm 3 without lines 17–18.
- **DivideSkip**: our implementation of the algorithm (Li et al., 2008)[2].
- **Locality Sensitive Hashing (LSH)** (Andoni and Indyk, 2008): This baseline system follows the design of previous work (Ravichandran et al., 2005). This system *approximately* solves Equation 1 by finding dictionary entries whose LSH values are within the (bit-wise) hamming distance of θ from the LSH value of a query string. To adapt the method to approximate dictionary matching, we used a 64-bit LSH function computed with letter trigrams. By design, this method does not find an exact solution to Equation 1; in other words, the method can miss dictionary entries that are actually similar to the query strings. This system has three parameters, θ, q (number of bit permutations), and B (search width), to control the tradeoff between retrieval speed and recall[3]. Generally speaking, increasing these parameters improves the recall, but slows down the speed. We determined $\theta = 24$ and $q = 24$ experimentally[4], and measured the performance when $B \in \{16, 32, 64\}$.

The systems, excluding LSH, share the same implementation of Algorithm 1 so that we can specifically examine the differences of the algorithms for τ-overlap join. The C++ source code of the system used for this experiment is available[5]. We ran all experiments on an application server running Debian GNU/Linux 4.0 with Intel Xeon 5140 CPU (2.33 GHz) and 8 GB main memory.

---

[2]We tuned parameter values $\mu \in \{0.01, 0.02, 0.04, 0.1, 0.2, 0.4, 1, 2, 4, 10, 20, 40, 100\}$ for each dataset. We selected the parameter with the fastest response.

[3]We followed the notation of the original paper (Ravichandran et al., 2005) here. Refer to the original paper for definitions of the parameters θ, q, and B.

[4]q was set to 24 so that the arrays of shuffled hash values are stored in memory. We chose $\theta = 24$ from $\{8, 16, 24\}$ because it showed a good balance between accuracy and speed.

[5]http://www.chokkan.org/software/simstring/

3.1 Datasets

We used three large datasets with person names (IMDB actors), general English words (Google Web1T), and biomedical names (UMLS).

- **IMDB actors**: This dataset comprises actor names extracted from the IMDB database[6]. We used all actor names (1,098,022 strings; 18 MB) from the file `actors.list.gz`. The average number of letter trigrams in the strings is 17.2. The total number of trigrams is 42,180. The system generated index files of 83 MB in 56.6 s.
- **Google Web1T unigrams**: This dataset consists of English word unigrams included in the Google Web1T corpus (LDC2006T13). We used all word unigrams (13,588,391 strings; 121 MB) in the corpus after removing the frequency information. The average number of letter trigrams in the strings is 10.3. The total number of trigrams is 301,459. The system generated index files of 601 MB in 551.7 s.
- **UMLS**: This dataset consists of English names and descriptions of biomedical concepts included in the Unified Medical Language System (UMLS). We extracted all English concept names (5,216,323 strings; 212 MB) from `MRCONSO.RRF.aa.gz` and `MRCONSO.RRF.ab.gz` in UMLS Release 2009AA. The average number of letter trigrams in the strings is 43.6. The total number of trigrams is 171,596. The system generated index files of 1.1 GB in 1216.8 s.

For each dataset, we prepared 1,000 query strings by sampling strings randomly from the dataset. To simulate the situation where query strings are not only identical but also similar to dictionary entries, we introduced random noise to the strings. In this experiment, one-third of the query strings are unchanged from the original (sampled) strings, one-third of the query strings have one letter changed, and one-third of the query strings have two letters changed. When changing a letter, we randomly chose a letter position from a uniform distribution, and replaced the letter at the position with an ASCII letter randomly chosen from a uniform distribution.

3.2 Results

To examine the scalability of each system, we controlled the number of strings to be indexed from 10%–100%, and issued 1,000 queries. Figure 1 portrays the average response time for retrieving strings whose cosine similarity values are no smaller than 0.7. Although LSH (B=16) seems to be the fastest in the graph, this system missed many true positives[7]; the recall scores of approximate dictionary matching were 15.4% (IMDB), 13.7% (Web1T), and 1.5% (UMLS). Increasing the parameter B improves the recall at the expense of the response time. LSH (B=64)[8]. It not only ran slower than the proposed method, but also suffered from low recall scores, 25.8% (IMDB), 18.7% (Web1T), and 7.1% (UMLS). LSH was useful only when we required a quick response much more than recall.

The other systems were guaranteed to find the exact solution (100% recall). The proposed algorithm was the fastest of all exact systems on all datasets: the response times per query (100% index size) were 1.07 ms (IMDB), 1.10 ms (Web1T), and 20.37 ms (UMLS). The response times of the Naïve algorithm were too slow, 32.8 s (IMDB), 236.5 s (Web1T), and 416.3 s (UMLS).

The proposed algorithm achieved substantial improvements over the AllScan algorithm: the proposed method was 65.3 times (IMDB), 227.5 times (Web1T), and 13.7 times (UMLS) faster than the Naïve algorithm. We observed that the Signature algorithm, which is Algorithm 3 without lines 17–18, did not perform well: The Signature algorithm was 1.8 times slower (IMDB), 2.1 times faster (Web1T), and 135.0 times slower (UMLS) than the AllScan algorithm. These results indicate that it is imperative to minimize the number of candidates to reduce the number of binary-search operations. The proposed algorithm was 11.1–13.4 times faster than DivideSkip.

Figure 2 presents the average response time

---

[6] `ftp://ftp.fu-berlin.de/misc/movies/database/`

[7] Solving Equation 1, all systems are expected to retrieve the exact set of strings retrieved by the Naïve algorithm.

[8] The response time of LSH (B=64) on the IMDB dataset was 29.72 ms (100% index size).
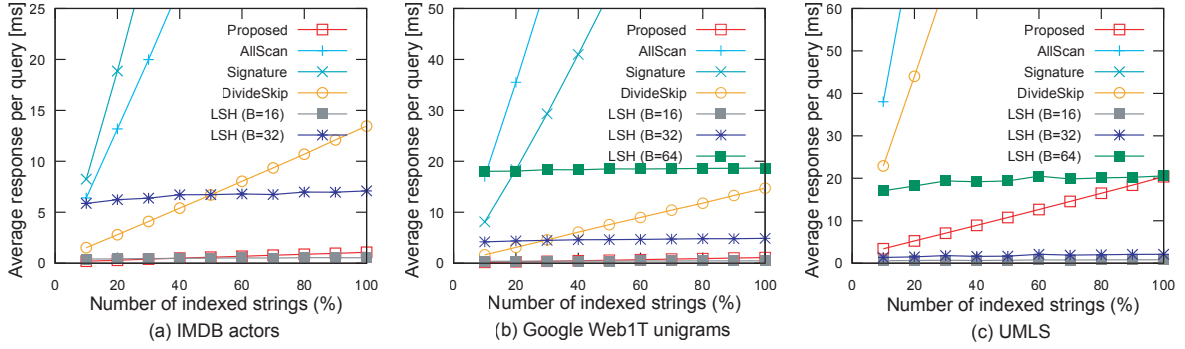
Figure 1: Average response time for processing a query (cosine similarity; $\alpha = 0.7$).
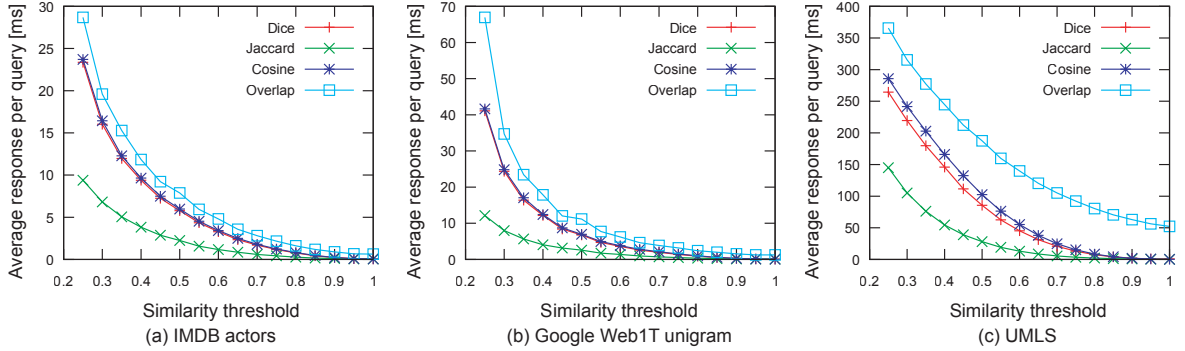

Figure 2: Average response time for processing a query.

of the proposed algorithm for different similarity measures and threshold values. When the similarity threshold is lowered, the algorithm runs slower because the number of retrieved strings $|\mathcal{Y}|$ increases exponentially. The Dice coefficient and cosine similarity produced similar curves.

Table 2 summarizes the run-time statistics of the proposed method for each dataset (with cosine similarity and threshold 0.7). Using the IMDB dataset, the proposed method searched for strings whose size was between 8.74 and 34.06; it retrieved 4.63 strings per query string. The proposed algorithm scanned 279.7 strings in 4.6 inverted lists to obtain 232.5 candidate strings. The algorithm performed a binary search on 4.3 inverted lists containing 7,561.8 strings in all. In contrast, the AllScan algorithm had to scan 16,155.1 strings in 17.7 inverted lists and considered 9,788.7 candidate strings, and found only 4.63 similar strings.

This table clearly demonstrates three key contributions of the proposed algorithm for efficient

approximate dictionary matching. First, the proposed algorithm scanned far fewer strings than did the AllScan algorithm. For example, to obtain candidate strings in the IMDB dataset, the proposed algorithm scanned 279.7 strings, whereas the AllScan algorithm scanned 16,155.1 strings. Therefore, the algorithm examined only 1.1%–3.5% of the strings in the entire inverted lists in the three datasets. Second, the proposed algorithm considered far fewer candidates than did the AllScan algorithm: the number of candidate strings considered by the algorithm was 1.2%–6.6% of those considered by the AllScan algorithm. Finally, the proposed algorithm read fewer inverted lists than did the AllScan algorithm. The proposed algorithm actually read 8.9 (IMDB), 6.0 (Web1T), and 31.7 (UMLS) inverted lists during the experiments[9]. These values indicate that the proposed algorithm can solve τ-overlap join problems by checking only 50.3% (IMDB), 53.6% (Web1T), and 51.9% of the total inverted lists re-

---

[9]These values are 4.6 + 4.3, 3.1 + 2.9, and 14.3 + 17.4.

Table 2: Run-time statistics of the proposed algorithm for each dataset

Averaged item	IMDB	Web1T	UMLS	Description
$\min |y|$	8.74	5.35	21.87	minimum size of trigrams of target strings
$\max |y|$	34.06	20.46	88.48	maximum size of trigrams of target strings
τ	14.13	9.09	47.77	minimum number of overlaps required/sufficient per query
$|\mathcal{Y}|$	4.63	3.22	111.79	number of retrieved strings per query
Total				— averaged for each query and target size:
# inverted lists	17.7	11.2	61.1	number of inverted lists retrieved for a query
# strings	16 155.1	52 557.6	49 561.4	number of strings in the inverted list
# unique strings	9 788.7	44 834.6	17 457.5	number of unique strings in the inverted list
Candidate stage				— averaged for each query and target size:
# inverted lists	4.6	3.1	14.3	number of inverted lists scanned for generating candidates
# strings	279.7	552.7	1 756.3	number of strings scanned for generating candidates
# candidates	232.5	523.7	1 149.7	number of candidates generated for a query
Validation stage				— averaged for each query and target size:
# inverted lists	4.3	2.9	17.4	number of inverted lists examined by binary search for a query
# strings	7 561.8	19 843.6	20 443.7	number of strings targeted by binary search

trieved for queries.

4 Related Work

Numerous studies have addressed approximate dictionary matching. The most popular configuration uses n-grams as a string representation and the edit distance as a similarity measure. Gravano et al. (1998; 2001) presented various filtering strategies, e.g., count filtering, position filtering, and length filtering, to reduce the number of candidates. Kim et al. (2005) proposed two-level n-gram inverted indices (n-Gram/2L) to eliminate the redundancy of position information in n-gram indices. Li et al. (2007) explored the use of variable-length grams (VGRAMs) for improving the query performance. Lee et al. (2007) extended n-grams to include wild cards and developed algorithms based on a replacement semi-lattice. Xiao et al. (2008) proposed the Ed-Join algorithm, which utilizes mismatching n-grams.

Several studies addressed different paradigms for approximate dictionary matching. Bocek et al. (2007) presented the Fast Similarity Search (FastSS), an enhancement of the neighborhood generation algorithms, in which multiple variants of each string record are stored in a database. Wang et al. (2009) further improved the technique of neighborhood generation by introducing partitioning and prefix pruning. Huynh et al. (2006) developed a solution to the k-mismatch problem in compressed suffix arrays. Liu et al. (2008) stored string records in a trie, and proposed a framework called TITAN. These studies are spe-

cialized for the edit distance measure.

A few studies addressed approximate dictionary matching for similarity measures such as cosine and Jaccard similarities. Chaudhuri et al. (2006) proposed the SSJoin operator for similarity joins with several measures including the edit distance and Jaccard similarity. This algorithm first generates *signatures* for strings, finds all pairs of strings whose signatures overlap, and finally outputs the subset of these candidate pairs that satisfy the similarity predicate. Arasu et al. (2006) addressed *signature schemes*, i.e., methodologies for obtaining signatures from strings. They also presented an implementation of the SSJoin operator in SQL. Although we did not implement this algorithm in SQL, it is equivalent to the Signature algorithm in Section 3.

Sarawagi and Kirpal (2004) proposed the MergeOpt algorithm for the τ-overlap join to approximate string matching with overlap, Jaccard, and cosine measures. This algorithm splits inverted lists for a given query A into two groups, S and L, maintains a heap to collect candidate strings on S, and performs a binary search on L to verify the condition of the τ-overlap join for each candidate string. Their subsequent work includes an efficient algorithm for the top-k search of the overlap join (Chandel et al., 2006).

Li et al. (2008) extended this algorithm to the SkipMerge and DivideSkip algorithms. The SkipMerge algorithm uses a heap to compute the τ-overlap join on entire inverted lists A, but has an additional mechanism to increment the fron-

tier pointers of inverted lists efficiently based on the strings popped most recently from the heap. Consequently, SkipMerge can reduce the number of strings that are pushed to the heap. Similarly to the MergeOpt algorithm, DivideSkip splits inverted lists A into two groups S and L, but it applies SkipMerge to S. In Section 3, we reported the performance of DivideSkip.

Charikar (2002) presented the Locality Sensitive Hash (LSH) function (Andoni and Indyk, 2008), which preserves the property of cosine similarity. The essence of this function is to map strings into N-bit hash values where the bitwise hamming distance between the hash values of two strings approximately corresponds to the angle of the two strings. Ravichandran et al. (2005) applied LSH to the task of noun clustering. Adapting this algorithm to approximate dictionary matching, we discussed its performance in Section 3.

Several researchers have presented refined similarity measures for strings (Winkler, 1999; Cohen et al., 2003; Bergsma and Kondrak, 2007; Davis et al., 2007). Although these studies are sometimes regarded as a research topic of approximate dictionary matching, they assume that two strings for the target of similarity computation are given; in other words, it is out of their scope to find strings in a large collection that are similar to a given string. Thus, it is a reasonable approach for an approximate dictionary matching to quickly collect candidate strings with a loose similarity threshold, and for a refined similarity measure to scrutinize each candidate string for the target application.

5 Conclusions

We present a simple and efficient algorithm for approximate dictionary matching with the cosine, Dice, Jaccard, and overlap measures. We conducted experiments of approximate dictionary matching on large-scale datasets with person names, biomedical names, and general English words. Even though the algorithm is very simple, our experimental results showed that the proposed algorithm executed very quickly. We also confirmed that the proposed method drastically reduced the number of candidate strings considered during approximate dictionary matching. We believe that this study will advance practical NLP

applications for which the execution time of approximate dictionary matching is critical.

An advantage of the proposed algorithm over existing algorithms (e.g., MergeSkip) is that it does not need to read all the inverted lists retrieved by query n-grams. We observed that the proposed algorithm solved τ-overlap joins by checking approximately half of the inverted lists (with cosine similarity and threshold $\alpha = 0.7$). This characteristic is well suited to processing compressed inverted lists because the algorithm needs to decompress only half of the inverted lists. It is natural to extend this study to compressing and decompressing inverted lists for reducing disk space and for improving query performance (Behm et al., 2009).

Acknowledgments

References

Andoni, Alexandr and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122.

Arasu, Arvind, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient exact set-similarity joins. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 918–929.

Behm, Alexander, Shengyue Ji, Chen Li, and Jiaheng Lu. 2009. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 604–615.

Bergsma, Shane and Grzegorz Kondrak. 2007. Alignment-based discriminative string similarity. In *ACL '07: Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 656–663.

Bocek, Thomas, Ela Hunt, and Burkhard Stiller. 2007. Fast similarity search in large dictionaries. Technical Report ifi-2007.02, Department of Informatics (IFI), University of Zurich.

Chandel, Amit, P. C. Nagesh, and Sunita Sarawagi. 2006. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*.

Charikar, Moses S. 2002. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388.

Chaudhuri, Surajit, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*.

Cohen, William W., Pradeep Ravikumar, and Stephen E. Fienberg. 2003. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, pages 73–78.

Davis, Jason V., Brian Kulis, Prateek Jain, Suvrit Sra, and Inderjit S. Dhillon. 2007. Information-theoretic metric learning. In *ICML '07: Proceedings of the 24th International Conference on Machine Learning*, pages 209–216.

Gravano, Luis, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate string joins in a database (almost) for free. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 491–500.

Henzinger, Monika. 2006. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR '06: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 284–291.

Huynh, Trinh N. D., Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. 2006. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1-3):240–249.

Kim, Min-Soo, Kyu-Young Whang, Jae-Gil Lee, and Min-Jae Lee. 2005. n-Gram/2L: a space and time efficient two-level n-gram inverted index structure. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 325–336.

Lee, Hongrae, Raymond T. Ng, and Kyuseok Shim. 2007. Extending q-grams to estimate selectivity of string matching with low edit distance. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 195–206.

Li, Chen, Bin Wang, and Xiaochun Yang. 2007. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 303–314.

Li, Chen, Jiaheng Lu, and Yiming Lu. 2008. Efficient merging and filtering algorithms for approximate string searches. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 257–266.

Liu, Xuhui, Guoliang Li, Jianhua Feng, and Lizhu Zhou. 2008. Effective indices for efficient approximate string search and similarity join. In *WAIM '08: Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management*, pages 127–134.

Manku, Gurmeet Singh, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, pages 141–150.

Navarro, Gonzalo and Ricardo Baeza-Yates. 1998. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2).

Ravichandran, Deepak, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *ACL '05: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 622–629.

Sarawagi, Sunita and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 743–754.

Wang, Wei, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. 2009. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 759–770.

Winkler, William E. 1999. The state of record linkage and current research problems. Technical Report R99/04, Statistics of Income Division, Internal Revenue Service Publication.

Xiao, Chuan, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. In *VLDB '08: Proceedings of the 34th International Conference on Very Large Data Bases*, pages 933–944.