

# Efficient Algorithms for Approximate Member Extraction Using Signature-based Inverted Lists

Jiaheng Lu Jialong Han Xiaofeng Meng

School of Information, Renmin University of China Beijing 100872, China

Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China

jiahenglu@ruc.edu.cn jialonghan@gmail.com xfmeng@ruc.edu.cn

## ABSTRACT

We study the problem of approximate membership extraction (AME), i.e., how to efficiently extract substrings in a text document that approximately match some strings in a given dictionary. This problem is important in a variety of applications such as named entity recognition and data cleaning. We solve this problem in two steps. In the first step, for each substring in the text, we filter away the strings in the dictionary that are very different from the substring. In the second step, each candidate string is verified to decide whether the substring should be extracted. We develop an incremental algorithm using signature-based inverted lists to minimize the duplicate list-scan operations of overlapping windows in the text. Our experimental study of the proposed algorithms on real and synthetic datasets showed that our solutions significantly outperform existing methods in the literature.

## Categories and Subject Descriptors

H.2.4 [Database Management]: Systems – Textual Databases

## General Terms

Algorithms

## Keywords

Approximate Member Extraction, Filtration-verification, Incremental Computation.

## 1. INTRODUCTION

In this paper we study the problem of finding substrings in a text document  $M$  that approximately match (e.g. having similarity scores above a given threshold  $\delta$ ) some strings in a given dictionary  $R$  of strings. This problem, called AME (short for *Approximate Member Extraction*), arises in many applications, as illustrated by the following examples.

*Named Entity Recognition:* With a given document, we want to locate pre-defined entities such as person names, conference

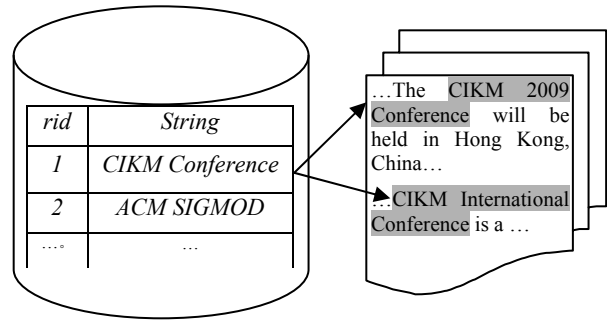


Figure 1. Approximate member extraction

names, and company names. We want this extraction to be *approximate*, i.e., we allow slight mismatches in the substrings. For instance, as shown in Figure 1, suppose we have a collection of conference names, such as “ACM SIGMOD” and “CIKM Conference.” We want to extract all conference names from a given document. We want to find matches such as “CIKM 2009 Conference” and “CIKM International Conference”, even though they do not match the string “CIKM Conference” in the dictionary exactly.

*Data Cleaning:* Documents in many applications could be “dirty” when it contains inconsistencies. Often we need to clean the inconsistencies. We need to perform data cleaning and integration by identifying the dirty words based on an existing dictionary.

One naive way to solve the AME problem is to enumerate each substring  $m$  of  $M$  and check if  $m$  matches strings in  $R$  approximately. Several algorithms have been proposed for doing the checking efficiently using two steps. In the first step, we filter the dictionary strings that are very different from  $m$ . In the second step, we compute the similarity between the remaining candidate strings and the string  $m$  to verify its approximate membership.

There are two methods to do the filtration in the first step. One is based on an *inverted index* on the dictionary  $R$ . In this method, strings are regarded as collections of *tokens*. For each token, the index stores the ids of the strings that include this token. For a given string  $m$ , we can find candidate similar strings in the dictionary by accessing the lists of the tokens in  $m$  and finding those string ids that have enough occurrences on the lists. The second method is based on *signatures generation* [1, 4]. This method focuses on exploring signature schemes that convert a string to a set of hash codes and filtering irrelevant strings using their signatures.

Recently, researchers have been trying to combine these two methods. For instance, Wang et al. propose a method called NGPP [13] to solve the AME problem by assuming the edit-distance similarity function. They shift and extend the partitions of dictionary strings to obtain an inverted index on all the *partition*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'09, November 2-6, 2009, Hong Kong, China.

Copyright 2009 ACM 978-1-60558-512-3/09/11...\$10.00.

variations (an implicit signature). Then by generating the neighborhood of partitions of document substrings and probing the index, the algorithm filters most irrelevant strings and performs verifications for the remaining strings. Chakrabarti et al. [2] studied how to solve the AME problem with other similarity score functions such as Jaccard coefficient using a method called ISH (Inverted Signature-based Hashtable), which encodes a string and its signatures into a 0-1 matrix. After generating a 0-1 matrix by computing the “bitwise-or” of all small matrices encoded with dictionary strings, ISH converts the problem of searching possible evidence into finding a certain 0-1 submatrix in the big matrix.

Our work in this paper is motivated by the following observations: (1) The ISH method generates signatures for strings and prune strings that share signatures with a total weight under a certain lower bound (see Para 8, Page 4 of [2]). Unfortunately, the lower bound may be too high, causing false negatives (see Example 1 for a counter-example). (2) The filtration processing of ISH involves an NP-Complete problem requiring finding *solid submatrix* (containing only 1) from a given 0-1 matrix under certain constraints. We give the reduction proof in this paper from an NP-Complete problem: *Balanced Complete Bipartite Subgraph* (see Appendix). Due to the intrinsic complexity, [2] solves this problem by a simple heuristics, which significantly increases the number of false positives, and thus incurs numerous I/Os in the verification phase and deteriorates overall performance. But our research in this paper shows that we can avoid the NP-complete complexity to efficiently solve AME problem by adopting a novel index structure, which effectively controls the number of false positives and consequently improves the overall performance. (3) NGPP[13] and ISH[2] both require that  $\delta$  is provided when preprocessing the data and generating an index for the filter. Once the threshold is fixed, these filters no longer support queries with thresholds other than  $\delta$  unless the index is re-generated.

EXAMPLE 1. (Example to illustrate the false negative in [2]) Under Jaccard similarity measurement, with weight assignment  $\{(a, 6), (b, 3.52), (c, 3.51), (d, 3.49), (e, 3.48), (f, 1)\}$ , string  $r = "bcde"$  matches  $m = "abef"$  under the similarity threshold  $\delta = 1/3$ . But from the following table we see that [2] incorrectly determines that  $m$  and  $r$  are impossible to match under the threshold  $1/3$ , though their real similarity is  $1/3$ .

String	Signatures ( $k=3$ )	$\tau$ value	Shared Signatures	Lower Bound Required by ISH Filter
$r = "bcde"$	$\{b, c, d\}$	$\tau(r) = 1.187$	$\{(b, 3.52)\}$	$\geq 3.67$
$m = "abef"$	$\{a, b, e\}$	$\tau(m) = 3.67$		

We summarize our contributions as follows:

- We adopt the prefix filtering technique [4], and propose new theorems which are all strictly proved and well applied as filtering conditions in our method. These theorems convert approximate matching to prefix signatures sharing, and give a tighter bound. More importantly, this filtering technique brings no false negatives.
- We utilize an inverted-list-based filtering index SIL and propose corresponding algorithm called EvSCAN. By performing inverted list scanning instead of introducing matrix-based combinatorial problem, EvSCAN naturally avoids solving NP-Complete problem. We also apply incremental optimizations on EvSCAN and propose EvITER,

which effectively reduces the duplicate list-scanning operations when the substring window shifts over a large document  $M$ . Compared with prior solutions, our method produces far less false evidences, thus achieves better filtration-verification balance and consequently improves the overall performance significantly.

- We modify our SIL to answer queries with dynamic similarity thresholds. Specifically, once initialized with a lower bound  $\delta_0$ , our filter works for any query with a threshold  $\delta \geq \delta_0$ . However previous filters only allow static similarity threshold and need to be re-initialized (i.e., the indices in filter has to be generated again) once the query threshold is changed.
- We provide detailed and accurate experimental results to support our argument. We show that SIL is significantly efficient than ISH, both in filtering power and overall running time, and explain the intrinsic reason by analyzing their runtime statistics in details. We also compare EvSCAN and EvITER under various situations, and show that our incremental optimization is effective.

The following sections are organized as follows:

Section 2 formally defines AME and gives some preliminaries. Section 3 complements the theory of prefix filtering and applies them to build the SIL structure and EvSCAN algorithm. Section 4 introduces the EvITER incremental algorithm. Section 5 shows how to support dynamic thresholds. Section 6 reports our experimental results. Section 7 discusses related work. Section 8 is the conclusion of our study.

## 2. PRELIMINARIES AND PROBLEM STATEMENT

### 2.1 Some Notations and Problem Statement

In this paper, we use the letter “t” to denote a token, and the other lower-case letters are used for strings. We regard strings as sets of tokens. For any token t, we denote  $wt(t)$  as the weight of t (e.g. IDF weight), and for any string s, we define  $wt(s)$  as  $\sum_{t \in s} wt(t)$ , i.e. the sum of weights of all tokens in s. Based on the above notations we define Jaccard similarity of any strings  $s_1$  and  $s_2$  as:

$$J(s_1, s_2) = \frac{wt(s_1 \cap s_2)}{wt(s_1 \cup s_2)}.$$

EXAMPLE 2. (Reusing the string and weight configuration of Example 1) Under weighted Jaccard similarity, the two strings  $m = "abef"$ ,  $r = "bcde"$  have a similarity of  $wt(\{b, e\})/wt(\{a, b, c, d, e, f\}) = (3.52 + 3.48)/(6 + 3.52 + 3.51 + 3.49 + 3.48 + 1) = 1/3$ .

With all notations introduced, we present the formal description of AME as follow:

#### Problem Statement

Given a dictionary  $R$  of strings and a similarity threshold  $\delta \in [0, 1]$ , then a query  $M$  is submitted. Here  $M$  represents a relatively long string (e.g. a text file). The task of AME is to extract all  $M$ 's substrings  $m$ , such that there exists some  $r \in R$  satisfying  $Sim(m, r) \geq \delta$ .

Sometimes we are only interested in substrings whose length is up to a length threshold  $L$ , so we may as well require that  $|m| \leq L$ . Here any extracted  $m$  is called approximate member of  $R$ , and

corresponding evidence for  $r$  in  $R$ . Note that our algorithms we present later can handle any similarity function that satisfies the following properties:

- $Sim(m,r)$  is symmetric, i.e.,  $Sim(m,r) = Sim(r,m)$ .
- $Sim(m,r) \leq \frac{wt(m \cap r)}{wt(m)}$ .

$$(\text{Symmetrically we have } Sim(m,r) \leq \frac{wt(m \cap r)}{wt(r)}.)$$

The first property is natural, and the second is also shared by many similarity functions [2]. For example, for any  $m$  and  $r$ , we have  $J(m,r) = \frac{wt(m \cap r)}{wt(m \cup r)} \leq \frac{wt(m \cap r)}{wt(m)}$  because  $wt(m \cup r) \geq wt(m)$ , so the Jaccard similarity is capable of serving as a similarity function in our discussion.

## 2.2 The Filtration-Verification Framework

To efficiently solve AME and other related problems, researchers have been designing methods following two phases of filtration and verification [1, 2, 4, 7, 13]. In the AME problem, employing this framework usually requires building an indexing structure for the dictionary  $R$ . Recall that for each approximate member  $m$  extracted, we define the string  $r$  in  $R$  that is similar enough with  $m$  to be  $m$ 's evidence. Thus, the task of extracting all approximate members from  $M$  can be simply reduced to determining whether there exists any evidence for each substring of  $M$ , and filtration-verification is actually referred to as evidence filtration and evidence verification.

Generally, our foundation of filtration is based on some necessary condition (denoted as NC) of our matching criterion  $Sim \geq \delta$ , that is, if some candidate evidence is real evidence, it must satisfy NC. With the dictionary  $R$  given offline, we build an index that quickly recommends for a query  $m$  ALL potential evidence that meets NC, so that true evidence is never missed. Then the evidence is verified against the actual matching criterion to determine whether the string  $m$  is a true approximate member.

Note that NC plays a key role in our whole framework. It ensures the correctness of the whole algorithm. Moreover, it determines how balanced our framework is. We can evaluate it through:

- How powerful is it? That is, does it eliminate as much false evidence as possible?
- Is it easy-going? That is, can we build a quick index to test it at low cost?

There is a tradeoff in the cost between two phases of filtration-verification. For example, with all dictionary strings being potentially possible evidence, the most easy-going filtration approach for them is obviously "no filtration", which leads to a brute-force method of scanning the whole dictionary. On the other hand, if we try to make our filtration the most powerful, i.e., it produces no false positive and achieves the smallest verification time; it will be expensive to perform such filtering. As a matter of fact, we need to obtain a beneficial compromise between two phases. In the following sections, we will intentionally highlight this issue through our theoretical and experimental analysis.

## 2.3 The K-Signature Scheme

The  $k$ -signature scheme is first demonstrated by Chakrabarti et al. in [2]. It is an extension of the prefix signature idea [1]. Here we briefly introduce some key definitions as follows:

**DEFINITION 1.** For a given string  $s$  and similarity threshold  $\delta$ , we sort all its tokens by their weight in descending order (if two tokens appear with the same weight, we sort them lexicographically), and choose the first few tokens to get a subset  $Sig(s)$ , such that  $\tau(s) = wt(Sig(s)) - (1 - \delta)wt(s) \geq 0$ . We call  $Sig(s)$  a prefix signature set of string  $s$ . For convenience we call it signature set from now on.

**EXAMPLE 3.** (Reusing the string and weight configuration of Example 1) Let  $\delta = 0.6$ , then  $\{a,b\}$  is a signature set of  $m$  because  $\tau(m) = wt(\{a,b\}) - (1 - 0.6) * wt(\{a,b,e,f\}) = 9.52 - 5.6 = 3.92 \geq 0$ .

Based on its definition, the following is some facts about prefix signature set:

- For any string  $s$ ,  $Sig(s)$  always exists because we can let  $Sig(s) = s$ , thus making  $\tau(s) = wt(s) - (1 - \delta)wt(s) = \delta * wt(s) \geq 0$ , i.e., we choose itself to be its signature set.
- A string may have more than one signature set with different sizes. For instance in Example 3, we see that  $m$  has another different signature set  $\{a,b\}$  besides itself.

One way to ensure the uniqueness of signature set is to use a parameter  $k$  to determine which set we choose for a string  $s$ , where  $k$  is a positive integer. We call this unique signature set  $k$ -signature set, denoted as  $Sig_k(s)$ . When  $k$  is fixed, we select  $Sig_k(s)$  among all available signature sets as follows:

- If all signature sets' sizes are bigger than  $k$ , choose the smallest one.
- Else if there is any signature set whose size is exactly  $k$ , choose it.
- Else, choose the largest one, i.e. the string itself.

In some special case, when we set  $k=1$ , we call the derived signature scheme as *min-signature* scheme. When  $k$  is set to be  $\infty$ , we will get the string itself to be its signature set.

**EXAMPLE 4.** (Following the configurations of Example 3) We list the different  $Sig(m)$  under different  $k$  settings as below:

$K$	1	2	3	4	5
$Sig_k(m)$	$\{a\}$	$\{a,b\}$	$\{a,b,e\}$	$\{a,b,e,f\}$	$\{a,b,e,f\}$

**Table 1. Signature sets controlled by  $K$**

Note that parameter  $k$  has nothing to do with the requirement of signature set, so we may randomly set  $k$  for our filter, and even choose different  $k$  for different strings. As a matter of fact, if we regard signature set as a compression of information in strings, then  $k$  is the parameter for global compression rate tuning. That is,  $k$  only influences the performance of our filter. We will further discuss its role in Section 3.3.

With the above description, we find that for any string  $s$ , the signature set is actually controlled by  $\delta$  and  $k$ , thus should be written as  $Sig_k(s, \delta)$ . In fact, the parameter  $k$  is fixed to SIL and before Section 5 we also consider that  $\delta$  is static, so when the context is clear, we still use  $Sig(s)$  to denote the signature set of  $s$ . In Section 5, we use  $Sig(s, \delta)$  because we start discussing how to support dynamic  $\delta$  during query processing.

### 3. FILTRATION VIA SIGNATURE-BASED INVERTED LISTS

In this subsection, we show some nice properties about the signature set which will be used in our algorithms. For instance, if String  $m$  and  $r$  meet the matching condition  $\text{Sim}(m,r) \geq \delta$ ,  $m$  must contain at least one of  $r$ 's signatures. This is apparently a necessary condition for matching and can be utilized to build a filter. In the following discussion, we'll further explore the property of signature sets and show that there are better filtering conditions.

#### 3.1 The Property of Signature Sets

LEMMA 1. For any string  $s$  and its selected signatures, we use  $\text{minsigwt}(s) = \min_{t \in \text{sig}(s)} \{wt(t)\}$  to denote the smallest weight of all  $s$ 's signature tokens. Then for any string  $s$ :

*A token  $t \in \text{Sig}(s)$  if  $t \in s$  and  $wt(t) \geq \text{minsigwt}(s)$ .*

This is apparent because the selected signature tokens must have larger weight than unselected tokens.

LEMMA 2. (PROPERTY OF PREFIX SIGNATURES). *For any string  $m$  and  $r$ , if  $\text{minsigwt}(m) \geq \text{minsigwt}(r)$ , then  $wt(\text{Sig}(m) \cap \text{Sig}(r)) \geq wt(\text{Sig}(m)) - wt(m-r)$ . Here  $m-r$  refers to the minus set of  $m$  and  $r$ .*

**Proof:** *We transform it to an equivalent form as  $wt(m-r) \geq wt(\text{Sig}(m) - \text{Sig}(r))$ , and prove this by showing that  $\text{Sig}(m) - \text{Sig}(r)$  is a subset of  $m-r$ . For any  $t \in \text{Sig}(m) - \text{Sig}(r)$ , we have  $t \in \text{Sig}(m)$ , so  $t \in m$ .*

*Now we prove that  $t \notin r$ .*

*Suppose that  $t \in r$ . because  $t \in \text{Sig}(m)$ , we know that  $wt(t) \geq \text{minsigwt}(m) \geq \text{minsigwt}(r)$ . From  $t \in r$ ,  $wt(t) \geq \text{minsigwt}(r)$  and Lemma 1, we conclude that  $t \in \text{Sig}(r)$ . This is inconsistent with the fact that  $t \in \text{Sig}(m) - \text{Sig}(r)$ . So  $t \notin r$ .*

*From  $t \notin r$  and  $t \in m$ , it's obvious  $t \in m-r$ . So  $\text{Sig}(m) - \text{Sig}(r)$  is a subset of  $m-r$ , easily leading to the result that  $wt(m-r) \geq wt(\text{Sig}(m) - \text{Sig}(r))$ . ■*

In Lemma 2, we illustrate the signature set overlapping relationship between matching strings. Intuitively this inequality condition is tighter than other conditions proposed in [1], and we believe this property is also useful in other researches involving prefix signatures. However, the set minus operator seems costly to handle, so we need to make this condition more easy-going. We solve this by introducing *Theorem 1* as follow:

THEOREM 1 (FILTERING CONDITION). *For any  $m$  and  $r$  that satisfy  $\text{Sim}(m,r) \geq \delta$ ,  $wt(\text{Sig}(m) \cap \text{Sig}(r)) \geq \min\{\tau(m), \tau(r)\}$ .*

**Proof:** *If  $\text{minsigwt}(m) \geq \text{minsigwt}(r)$ , according to Lemma 2, we have  $wt(\text{Sig}(m) \cap \text{Sig}(r)) \geq wt(\text{Sig}(m)) - wt(m-m \cap r) = wt(\text{Sig}(m)) - wt(m) + wt(m \cap r) \geq wt(\text{Sig}(m)) - wt(m) + \delta * wt(m \cup r) \geq wt(\text{Sig}(m)) - wt(m) + \delta * wt(m) = wt(\text{Sig}(m)) - (1 - \delta)wt(m) = \tau(m) \geq \min\{\tau(m), \tau(r)\}$ .*

*If  $\text{minsigwt}(m) \leq \text{minsigwt}(r)$ , based on the symmetry of  $\text{Sim}(m,r)$  we have the same result. So in conclusion we have  $wt(\text{Sig}(m) \cap \text{Sig}(r)) \geq \min\{\tau(m), \tau(r)\}$ . ■*

#### ALGORITHM 1: BuildSIL( $R, \delta, k$ )

```

1  for each  $r \in R$  do
2       $\text{Sig} \leftarrow \text{GenSig}(r, \delta, k)$ ;
      /*The function  $\text{GenSig}(r, \delta, k)$  generates signature
      for  $r$  under  $k$ -signature scheme. */
3      for each  $t \in \text{Sig}$  do
4           $\text{list}[t] = \text{list}[t] \cup \{rid(r)\}$ ; //insert  $rid$  of  $r$  into list
5  return list;
```

EXAMPLE 5. *(Following the configurations of Example 3) Suppose  $k=2$ , we have  $\text{Sig}(m) = \{a,b\}$ ,  $\tau(m) = 3.92$  and  $\text{Sig}(r) = \{b,c\}$ ,  $\tau(r) = (3.52 + 3.51) - 0.4 * 14 = 1.43$ . So  $wt(\text{Sig}(m) \cap \text{Sig}(r)) = wt(\{b\}) = 3.52 \geq \min\{\tau(m), \tau(r)\} = \min\{3.92, 1.43\} = 1.43$ .*

We obtain the foundation of filtration phase of SIL so far. It's easy to discover that when the threshold  $\tau(r)$  is computed offline, this filtering condition only involves the signature set of all strings, indicating the fact that the time and space requirement of our filter is tightly related to the average signature set size of all strings in the dictionary  $R$ , which is controlled by the parameter  $k$ . Moreover, different  $k$  provides different filtering conditions. Among them we need to decide which one to choose.

#### 3.2 Filtration via SIL

Since for any matched  $m$  and  $r$ , their signature sets overlaps, it's easy to come up with the idea of building an inverted index structure for the dictionary  $R$ , and filtering by merging inverted lists and accumulating weights. After this index is built up offline, by visiting  $\text{list}[t]$ , we can quickly retrieve for any token  $t$  a list of  $rid$  of all  $r$  in  $R$ , who contains token  $t$  as a signature. Due to the fact that these lists only involves signatures of all strings in  $R$ , we call this index SIL in short for *signature-based inverted lists*, and Algorithm 1 below shows the method to generate an SIL index.

EXAMPLE 6. *Suppose we have a dictionary  $R = \{r[1] = \text{"SIL's filtering power"}, r[2] = \text{"the power of filtering by SIL"}\}$ , the weight of each tokens are  $\{<\text{SIL's}, 5>, <\text{filtering}, 4>, <\text{power}, 3.5>, <\text{SIL}, 3>, <\text{by}, 2>, <\text{the}, 1>, <\text{of}, 1>\}$ . We set  $k=1$  and  $\delta = 0.55$ , then we have each strings' signature set in Table 2 and the SIL built as Figure 2.*

rid	String	Signature Set
1	"SIL's filtering power"	{ "SIL's", "filtering" }
2	"the power of filtering by SIL"	{ "filtering", "power" }

Table 2. Signature sets of  $R$ 's strings

Signature	String rids	rid	wt(r)	$\tau(r)$
"SIL's"	(1)	1	12.5	3.375
"filtering"	(1), (2)	2	14.5	0.975
"power"	(2)			

Figure 2. SIL and additional information for dictionary  $R$

When a string  $m$ 's membership needs to be checked, we simply compute the signature set of  $m$ , denoted as  $\{t_1, t_2, \dots, t_n\}$ . Then we scan all  $n$  lists that is indexed by  $\text{list}[t_1], \text{list}[t_2], \dots, \text{list}[t_n]$ , while aggregating the weight of  $t_i$  to all  $rid$  whose record contains  $t_i$  as one of its signature. To record the aggregated weight, we may

---

**ALGORITHM 2: EvSCAN**(  $M, \delta, k, L$  )

---

```

1  ResultSet ←  $\Phi$ ; // for storing approximate members
2  for each  $m$  of  $M$ 's substrings ( $|m| \leq L$ ) do
3      Sig ← GenSig( $m, \delta, k$ );
4      Initialize Sum[]; // for weight aggregating
5      CandSet ←  $\Phi$ ; // for storing candidate evidence
6      for each  $t \in \text{Sig}$  do
7          for each  $rid \in \text{list}[t]$  do
8              Sum[ $rid$ ] += wt( $rid$ ); // aggregating weight
9              if Sum[ $rid$ ]  $\geq \min\{\tau(m), \tau(rid)\}$  then
10                 CandSet ← CandSet  $\cup \{rid\}$ ;
11         for each  $rid \in \text{CandSet}$  do // verification
12             if Sim( $m, r(rid)$ )  $\geq \delta$  then // true evidence found
13                 ResultSet ← ResultSet  $\cup \{m\}$ ;
14         break;
15 return ResultSet;

```

---

use an array Sum[] for convenience or a hash table to save memory space. With all lists scanned, the aggregated weight of any  $rid$  is exactly the value of  $\text{wt}(\text{Sig}(m) \cap \text{Sig}(r))$ . In fact, if any  $rid$  appears satisfying the filtering condition of *Theorem 1*, i.e. with an aggregated weight larger than  $\min\{\tau(m), \tau(r)\}$ , we can store it for later verification to determine whether it is the one that makes  $m$  a true member. Note that  $\tau(r)$  and  $\text{wt}(r)$  for any dictionary string  $r$  is computed in the signature generating step of Algorithm 1 and they are stored in the main memory for the later use of our algorithms (See Figure 2).

### 3.3 Additional Discussion

In the above discussion, one may notice that we didn't involve the parameter  $k$ . This again proves the fact that with any assigned  $k$ , our algorithm will run correctly. Since the signature set is a compression of information in a string, we will certainly get more information if we choose a relatively large  $k$ , through which we can target potential matching evidences to a smaller scope, thus reducing the cost of verifying these evidences.

However, larger  $k$  causes longer inverted list length, i.e. more cost on targeting possible evidences. For instance, if we set  $k = \infty$ , that is, for all strings  $s$ , we set  $\text{Sig}(s)$  to be  $s$  itself, and  $\tau(s) = \delta * \text{wt}(s)$ , we interestingly find that our method degrades into a common inverted-list based solution. Chakrabarti [2] first analyzed this problem and showed that  $k=3$  is good on average situation, which is also proved in our experimental study.

## 4. OPTIMIZATION BY PROGRESSIVE COMPUTATION

### 4.1 Reducing Duplicate Computations

Although EvSCAN algorithm efficiently checks the approximate membership of each single substring  $m$  in a document  $M$ , it ignores the overlapping between shifting substring windows and consequently takes a lot of time on duplicate computations. Another way to solve AME is to reduce this

problem to set similarity join, which is already well studied by researchers [1, 4, 6]. In set similarity join, we are given SA and SB - two columns of sets, a similarity function Sim, and a threshold  $\delta$ . The task of set similarity join is to join the two columns, where the joining condition is  $\text{Sim}(SA, SB) \geq \delta$ . In AME, if we set  $SA=R$ ,  $SB=\{\text{all substrings of } M\}$ , run set similarity join between SA and SB and project the result set along SB, we will get the result of AME. Though the problem of set similarity join is explored and optimized in many papers, this method still doesn't notice the fact that the records in SB are quite similar with each other - they are substrings of a long text.

In this section, based on the above observations, we believe that the unique property of AME should be exploited separately, and optimized method could be designed accordingly. So we study the incremental property of Theorem 1, and demonstrate Theorem 2, in purpose of decreasing the duplicate list-scanning when examining all substrings of  $M$ .

### 4.2 Optimization by Progressive Computation

Assume  $m \oplus t$  denoting the string which we get by concatenating token  $t$  to the tail of string  $m$ . Consider the process of checking  $m$  and  $m \oplus t$ : we compute the signature set of  $m$  and verify the condition in Theorem 1, then we do the same job for  $m \oplus t$ . Intuitively the signature set of  $m$  and  $m \oplus t$  are much alike. We observe that: if some  $r$  cannot match  $m$  and does not contain  $t$ , it is not likely to match  $m \oplus t$ . Before we formalize our intuition into new theorem and algorithm, we introduce *Evidence Superset* by the lemma and definition below:

LEMMA 3. For any  $m$  and  $r$  that satisfy  $\text{Sim}(m, r) \geq \delta$ ,  $\text{wt}(m \cap \text{Sig}(r)) \geq \min\{\delta * \text{wt}(m), \tau(r)\}$ .

This lemma is an inference of Theorem 1. Recall that in previous sections we mention that Theorem 1 remains correct even if we set different  $k$  for different strings, Lemma 3 is in fact obtained by setting  $k = \infty$  for  $m$  in Theorem 1 (so  $\text{Sig}(m)$  is replaced by  $m$  and  $\tau(m)$  by  $\delta * \text{wt}(m)$ ). With Lemma 3 we define *Evidence Superset* for any query substring  $m$  as follow:

DEFINITION 2. Suppose  $\delta$  and  $k$  are fixed, for any string  $m$ , let  $ES(m) = \{r \in R \mid \text{wt}(m \cap \text{Sig}(r)) \geq \min\{\delta * \text{wt}(m), \tau(r)\}\}$ , we call  $ES(m)$  an Evidence Superset of  $m$ . Based on Lemma 3 it's obvious that any true evidence for  $m$  must be contained in  $ES(m)$ .

From its definition, we see that  $ES(m)$  is useful since any evidence matching  $m$  will be included in  $ES(m)$ . If we can efficiently compute  $ES(m)$  for any substring  $m$ , we can further filter elements in  $ES(m)$  to pick out all true evidences and check  $m$ 's approximate membership. Our intuition is formalized below.

LEMMA 4. For any string  $m$  and token  $t$ , if a dictionary  $r \notin ES(m)$  and  $t \notin \text{Sig}(r)$ , then  $r \notin ES(m \oplus t)$ .

**Proof:** We prove  $r \notin ES(m \oplus t)$  by showing that  $\text{wt}((m \oplus t) \cap \text{Sig}(r)) < \min\{\delta * \text{wt}(m \oplus t), \tau(r)\}$ :  $\text{wt}((m \oplus t) \cap \text{Sig}(r)) = \text{wt}(m \cap \text{Sig}(r))$  (Because  $t \notin \text{Sig}(r)$ )  $< \min\{\delta * \text{wt}(m), \tau(r)\}$  (because  $r \notin ES(m)$ )  $< \min\{\delta * \text{wt}(m \oplus t), \tau(r)\}$ . ■

This lemma states a fact that if a dictionary string is far from being evidence of current substring  $m$  and it is not a signature of the coming token  $t$ , then it cannot be evidence when the substring window moves to  $m \oplus t$ . However this lemma still cannot serve as a method for efficiently computing  $ES(m)$ , we further generalize

---

ALGORITHM 3: **EvITER**(  $M=\{t[1],t[2],\dots,t[n]\}$ ,  $\delta, k, L$  )

---

```

1  ResultSet  $\leftarrow$   $\Phi$ ; // for storing approximate members
2  for i=1 to n do
3    Initialize Sum[];
      /*Sum[rid].s1 records wt(Sig(m) $\cap$ Sig(r(rid))) and
      Sum[rid].s2 for wt(m $\cap$ Sig(r(rid))) */
4  LastES  $\leftarrow$   $\Phi$ ; // for iterating ES
5  for j=i to min{n,i+L-1} do //current m= t[i]...t[j]
6    ES  $\leftarrow$   $\Phi$ ;
7    CandSet  $\leftarrow$   $\Phi$ ; // for storing candidate evidence
8    update Sig(m) and maintain sum[];
      /*because m and Sig(m) changes */
9    for each rid  $\in$  list[t]  $\cup$  LastES do
10   if Sum[rid].s2  $\geq$  min{  $\delta$  * wt(m),  $\tau$  (r(rid)) } then
11     ES  $\leftarrow$  ES  $\cup$  {rid};
12   if Sum[rid].s1  $\geq$  min{  $\tau$  (m),  $\tau$  (rid) } then
13     CandSet  $\leftarrow$  CandSet  $\cup$  {rid};
14   VerifyAllCandidate(); //verification
15   LastES  $\leftarrow$  ES; //iteration for the next window
16 return ResultSet;
```

---

Lemma 4 to obtain Theorem 2 to demonstrate the incremental property of ES(m).

**THEOREM 2. (INCREMENTAL PROPERTY).** *Suppose  $\delta, k$  are fixed, it holds for any string  $m$  and token  $t$  that  $ES(m \oplus t) \subseteq ES(m) \cup list[t]$ .*

Notice that the approach we build SIL determines that  $t \in Sig(r)$  means  $r \in list[t]$ , so Theorem 2 is obviously based on Lemma 4. This theorem indicates an efficient iterative approach of maintaining ES() for the varying substring when the right boundary of the substring windows moves by a token. That is, we check all elements in  $ES(m) \cup list[t]$  and pick out proper ones into  $ES(m \oplus t)$ . Note that this process requires maintaining another field recording  $wt(m \cap Sig(r))$  in the summing table sum[], so it can be combined with the process of filtering by SIL.

Now we get a new algorithm of incrementally checking all substrings, that is, we fix the left boundary of the substring window and shift the other boundary to the right. While the substring varies we iteratively maintain corresponding ES() to filter and verify all evidence in it. Figure 3 shows this iteration process, with an instance of  $M=t_1 \oplus t_2$ . For convenience, we denote this incremental filtering algorithm as EvITER (Evidence Iterating), while EcSCAN in Section 3.2 is short for Evidence Scanning.

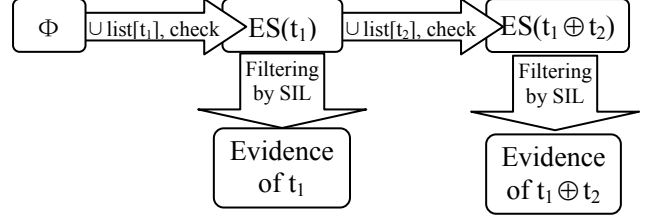


Figure 3. Flow of dictionary strings in EvITER

### 4.3 Combining Other Filtering Conditions

In Algorithm 1 and 2, we use “VerifyAllCandidate()” to denote the process of verification, by which one determine if  $m$  is really a true member. Since not all  $r$ 's in CandSet are the ones that make  $m$  a true member, we may as well make “VerifyAllCandidate()” a small filter-verification process, via introducing some other simple yet effective filtering conditions. The following is exactly one of such conditions we want:

$$\text{If } Sim(m,r) \geq \delta, \text{ then } wt(m) * \delta \leq wt(r) \leq wt(m) / \delta.$$

Intuitively, under the non-weighted situation, this condition states that if any two strings have too much difference in length, they are not likely to match each other. To apply this filtering condition, we need only store in memory the weight value of all strings in  $R$ . By checking this condition for all  $r$  in CandSet, we can quickly narrow our scope to fewer possible  $r$ , thus avoiding more disk accessing and making computation more efficient.

### 4.4 Algorithm Analysis

In this section we analyze the time and memory cost of our algorithms, and demonstrate the advantages of EvITER over EvSCAN. The notations to be used are listed in Table 3.

$ M $	The length of text used as the input of AME
$ R $	Dictionary size
$L_r$	Average length of dictionary strings
$L_m$	Average substring length
$L_{list}$	Average length of inverted lists
$\delta$	Similarity threshold
$E$	Total number of evidence that passes the filter
$C_v$	Time cost of verifying an evidence (including disk accessing and similarity score computing)

Table 3. Some notations in cost analysis

In addition to the above notations, we make two assumptions in order to simplify our discussion: (1) the length of all strings is longer than  $k$ . (2) all tokens have the same weight (e.g. 1). Therefore, we obtain an upper bound for the signature set size of any string.

**LEMMA 5.** *Suppose  $k$  and  $\delta$  are fixed, then for any string  $m$  with length  $L$ ,  $|Sig(m)| \leq \max\{k, (1-\delta)L\}$ .*

**Proof:** *Because for any string  $m$ ,  $|Sig(m)| = wt(Sig(m)) \geq (1-\delta) * wt(m) = (1-\delta)L$ , the smallest signature set size is  $(1-\delta)L$ .*

*If  $(1-\delta)L \leq k$ , from the definition of  $k$ -signature set we choose the first  $k$  tokens as signatures, so  $|Sig(m)| = k$ , else we choose the smallest signature set of size  $(1-\delta)L$ . Therefore we have  $|Sig(m)| \leq \max\{k, (1-\delta)L\}$ . ■*

With above assumptions and lemmas, the memory cost of our SIL can be expressed as

$$MEMCOST=|R|\max\{k, (1-\delta)L_r\}.$$

This equation is correct since every string in R produces  $\max\{k, (1-\delta)L_r\}$  nodes at most in the inverted lists. To estimate the time cost of our algorithms, we introduce another lemma as below:

LEMMA 6. For each substring  $m$ , the number of inverted lists that EvSCAN and EvITER scan are respectively  $|\text{Sig}(m)|$  and 2.

**Proof (Outline):** It's obvious for EvSCAN and we only give proof for EvITER. Consider the moment we finish checking  $m$  and prepare for  $m \oplus t$ , we have to scan  $\text{list}[t]$  once to iterate  $ES(m)$  into  $ES(m \oplus t)$ . Moreover, it's possible that  $t$  replace some signature token  $t'$  to be a new signature, so we must scan  $\text{list}[t']$  to maintain the summing table  $\text{sum}[]$  for next iteration. ■

Therefore, the filtration cost of EvSCAN and EvITER are respectively  $|M|L_m L_{\text{list}} \max\{k, (1-\delta)L_m\}$  and  $2|M|L_m L_{\text{list}}$ . The verification cost of EvSCAN can be estimated as  $EC_v$ , while that of EvITER is  $E(1+C_v)$  because of the  $O(1)$  evidence iteration cost for every evidence. In summary we have

$$TIMECOST(\text{EvSCAN}) = |M|L_m L_{\text{list}} \max\{k, (1-\delta)L_m\} + EC_v,$$

$$TIMECOST(\text{EvITER}) = 2|M|L_m L_{\text{list}} + E(1+C_v).$$

Here we see that EvITER avoids scanning some lists by introducing the cost of evidence iteration, so it may have some advantage when  $k$  is large or  $E$  is small, which will be demonstrated by our experimental results later.

## 5. SUPPORTING DYNAMIC SIMILARITY THRESHOLDS

### 5.1 The Static Threshold Problem

In the above discussion we talk about how to perform AME with a static similarity threshold, where the filter can be denoted as  $F(R, \delta_0)$ , given a dictionary  $R$  and a fixed threshold  $\delta_0$ , meaning that the threshold  $\delta_0$  is undesirably static. If users want to submit a query with other thresholds, the filter has to be re-initialized. This apparently leads to much inconvenience in practice. In this section we will focus on this issue and show that with a little modification, our SIL can handle this problem well. Note that in this section, the notation  $\text{Sig}(s)$  is replaced by  $\text{Sig}(s, \delta)$  to add a dynamic threshold  $\delta$ .

### 5.2 Solution and Analysis

In our SIL algorithm, we observe that the problem of static threshold is caused by the definition of prefix signatures. Recall that the prefix signatures  $\text{Sig}(s)$  for string  $s$  is a prefix subset of  $s$  that satisfies  $\text{wt}(\text{Sig}(s)) \geq (1-\delta)\text{wt}(s)$ . Therefore, with different  $\delta$  we need different number of signatures to build various filters.

Another observation is that, under min-signature schema, for any string  $s$ , if a token  $t$  is selected as a signature under some threshold, it will also be in the signature set of  $s$  when the threshold gets lower. That is, if we initialize the filter at a relatively low threshold  $\delta_0$ , when a query comes with a higher threshold  $\delta \geq \delta_0$ , those rids whose string contains  $t$  as a signature should be included in some nodes on  $\text{list}[t]$  of the current filter. For simplicity we call these nodes *active nodes*. All we need is to discriminate *active nodes*, and use them to perform filtration.

We propose Theorem 3 to provide a way to discriminate active nodes as follow:

THEOREM 3 (SUFFICIENT AND NECESSARY CONDITION OF MIN-SIGNATURE). Under min-signature schema, for any string  $s=\{t_1, t_2, \dots, t_n\}$ , where  $\text{wt}(t_1) \geq \text{wt}(t_2) \geq \dots \geq \text{wt}(t_n)$ , let  $U_i=1-(\text{wt}(t_1)+\text{wt}(t_2)+\dots+\text{wt}(t_i))/\text{wt}(s)$  for any  $i \geq 1$  and  $U_0=0$ . We have the following conclusion:

$$t_i \in \text{Sig}(s, \delta) \text{ if and only if } \delta \in [0, U_{i-1}).$$

Because  $U_{i-1}$  can be computed in the filter-constructing phase, in every node of all inverted lists, we add a field to record  $U_{i-1}$  in order to test the condition  $\delta \in [0, U_{i-1})$  to decide whether this is an active node. Moreover, we can sort all nodes in a list in descending order of corresponding  $U_{i-1}$ . In this way, for any threshold  $\delta$ , all active node in a list must form a prefix of the list. Therefore, we may stop our scan once an inactive node is found, by which we avoid scanning the whole list and enhance the performance.

Note that this modification should only be applied under the min-signature scheme. In fact, in Section 6 we will show that under most cases, min-signature schema is enough to serve as a good choice.

EXAMPLE 8. (adopting the configuration in Example 6) Suppose we initialize the modified filter with min-signature schema ( $k=1$ ) and  $\delta_0=0.55$  as Figure 4. We have a query with  $\delta=0.7$ , then all nodes in Figure 4 that is circled out become active nodes and should be scanned.

Signature	→	String rids and U
“SIL’s”	→	(1, 1.0)
“filtering”	→	(2, 1.0), (1, 0.6)
“power”	→	(2, 0.725)

Figure 4. Active nodes when  $\delta=0.7$

Comparing with the origin SIL, we see that applying this modification only requires a little more space and additional sorting in the filter-building phase. For queries with various similarity thresholds, the modified SIL successfully solves the static threshold problem, without visiting any additional list nodes or trading query performance.

## 6. EXPERIMENTAL STUDY

### 6.1 Experimental Settings

The following filters and algorithms are evaluated in this section:

**ISH** (abbreviated from *Inverted Signature-based Hashtable*) is a filter proposed in [2], whose idea is to optimize the query range of length filtering. In our experiment, we set the inverted hashtable length  $b$  to be 11 (8 is enough according to [2]).

**EvSCAN on SIL** is our proposed algorithm, which filters by scanning the Signature-based Inverted Lists. **EvITER** is an optimized version of EvSCAN, which aims at reducing unnecessary list scanning.



Figure 5. Comparison between ISH and SIL ( $k=3, |R|=1000, L=10$ )

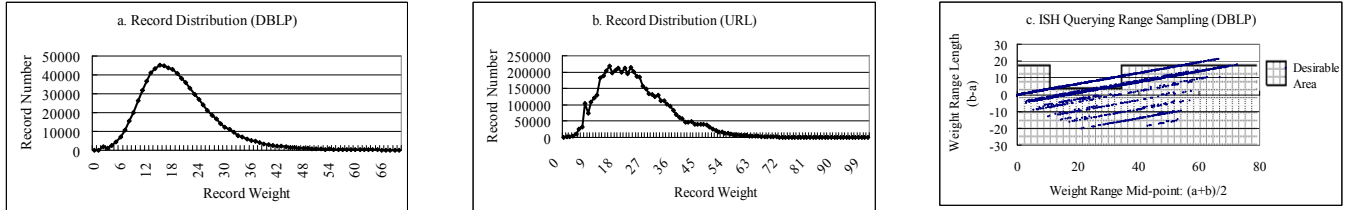


Figure 6. Some Statistics for ISH ( $k=3, L=10, \delta=0.85$ )

We ran our experiments on the following two datasets:

**DBLP:** It includes paper titles downloaded from the DBLP site. We extracted 274,788 paper titles with a total size 17.8MB as the dictionary. The query text to this dictionary is 40 web pages from CiteSeer, each containing the title, abstract, citation, etc. of a random paper. Tokens are separated by spaces and punctuations.

**URL:** The dictionary includes the first 1,838,973 URLs from an URL dataset. The query text is 40 text files, each containing 50 random URLs from the rest of the dataset. Tokens are separated by slashes.

On both datasets, standard IDF weight [14] is applied, and all tests were conducted under the weighted Jaccard similarity measurement. We mainly judge the performance of all filters via analyzing the filtering power and overall running time of them. We evaluate the power of filters by the candidate evidence they produce since the size of candidate evidences has a great influence on overall performance.

## 6.2 Comparing with ISH

We compared our approach EvSCAN on SIL with ISH in this section. We first performed experiments on DBLP data (dictionary size: 274,788 records), and our results show that ISH produced a large amount of candidate evidences and disk-accessing, thus spending much time on verification and could not terminate in one hour. Therefore, we had to reduce the size of dictionary using the first 1000 records in DBLP.

We explain this result through reviewing the filtering approach of ISH: for every query substring  $m$ , ISH optimizes the existing length filtering condition mentioned in Section 4.3, and uses a new range (denoted as  $[a, b]$ ,  $a = \delta * wt(m)$ ,  $b \leq wt(m) / \delta$ ) as the SQL querying condition at evidence record retrieving phase.

In Figure 6, we show the record distribution of two datasets across the weight axis (see sub-figure (a) and (b)), where all records distribute densely, with at most 50k and on average 10k in a unit length of weight range (DBLP dataset). This implies that it's unwise to retrieve all evidence whose weight is in certain range, unless we can make our query range desirably small or far from those regions with crowded records.

In Figure 6(c), we sampled the weight range  $[a, b]$  (each represented by a *characteristic point* (mid-point, length) or

$((a+b)/2, b-a)$ ) from 2,608 SQL queries ISH launches when processing a random webpage. We also flagged some areas as “desirable area”, where “desirable queries” appear (queries possessing small  $[a, b]$  ranges or avoiding the most frequent weight of all records, i.e. x-coordinate of the peak in Figure 6(a) and (b)).

We see in this figure that the mid-points of all sampled ranges vary averagely from 0 to 70, which include the most frequent weight in both datasets (15 for Figure 6(a) and 20 for (b)). Moreover, though ISH sometimes successfully confirms of no matching (denoted by ranges with negative length in Figure 6(c)), the range length in many queries is not desirably short. Therefore, there exist too many queries, whose *characteristic point* is located far from our “desirable area”. These insufficiently optimized queries lead to tons of I/Os and verification computations, thus deteriorating the overall performance of ISH.

## 6.3 Effects of parameters on SIL

Based on our analysis in the above sections, the performance of SIL is mainly influenced by the following aspects: the “compressing rate” parameter  $k$ , dictionary size, query text length, similarity threshold, and substring length threshold. We run EvITER on different parameter settings and record the results, from which we have the following observations:

- The parameter  $k$  is tightly related to every aspect of the filter. Larger  $k$  means stronger filtering power (Figure 7(a)), less verification time, and larger filter size. However, on average situation when  $\delta=0.85$  and  $L=10$ , the result shows that the most competitive  $k$  value is among 1, 2, and 3 (see Figure 7(b)), which makes the two phases of filtration and verification more balanced. This supports our discussion about compromising between the two phases.
- Though the performance of SIL depends much on the inherent property of the dataset (e.g. query texts about chemical science certainly run quickly on our URL dictionary because the number of word matching is expected to be small), it still exhibits a linear increase in running time under different dictionary size and query text length, which is expected in our cost analysis in Section 4.4.

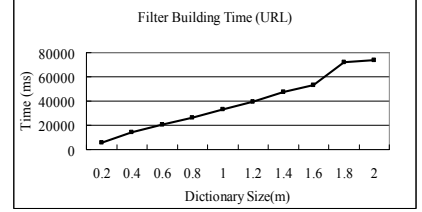
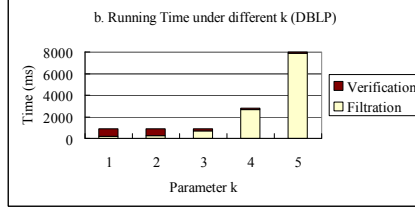
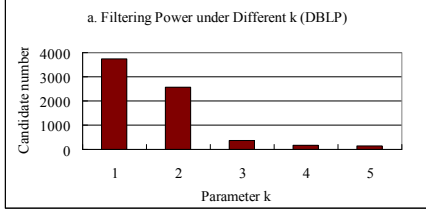


Figure 7. Performance under different k ( $L=10$ ,  $\delta=0.85$ )

Figure 8. Filter building ( $k=3$ ,  $\delta=0.85$ )

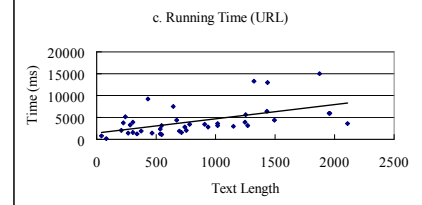
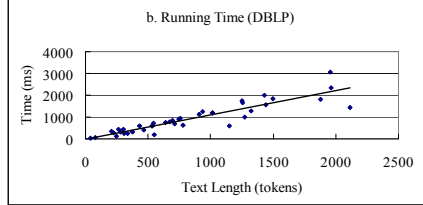
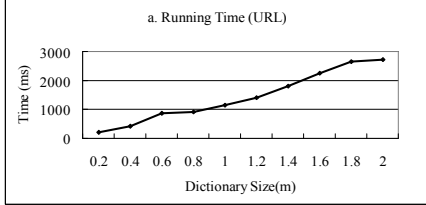


Figure 9. Performance under different data size ( $k=3$ ,  $L=10$ ,  $\delta=0.85$ )

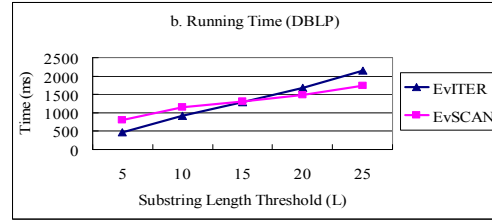
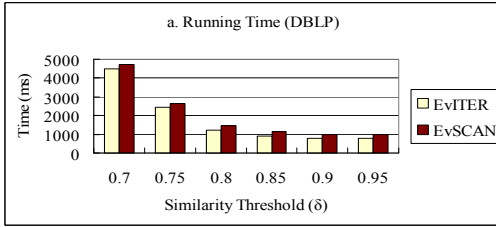


Figure 10. Performance under different thresholds ( $k=3$ )

## 6.4 Comparison between EvSCAN & EvITER

Besides the experimental analysis on ISH and SIL, we also performed a comparison between the two algorithms we propose: EvSCAN and EvITER. Instead of dictionary size and query length, in this subsection we mainly focus on the two varying threshold parameters: similarity threshold and substring length threshold.

Through the comparison we found that:

- The similarity threshold significantly affects the time consumed by our two algorithms. When  $\delta$  decreases from 0.95 to 0.7, both algorithms requires 4 times more running time. This is easily explained by our filtering condition: when  $\delta$  decreases, the signature sets of most strings get larger to make the chance of signature overlapping increase, while  $\tau(\cdot)$  for most strings leaves almost unchanged. Thus, it's easier for evidence to pass the filter.
- When  $L=10$ ,  $k=3$  and  $\delta=0.85$ , EvITER shows a performance increase of about 25% over EvSCAN, this is because EvITER reduces the operation of scanning an inverted list, by iterating from one evidence set to another. When the similarity threshold is high and the candidate evidence set is small, the advantage of EvITER will be more obvious.
- To our surprise, when  $L$  is above 15, EvITER is gradually outperformed by EvSCAN. This result is not expected by us. We carefully studied this issue and find the reason: because the candidate set  $ES(m)$  we maintain (recall in Section 4.2) tends to get bigger when  $m$  is longer, therefore EvITER will spends more time iterating it as larger  $L$  allows longer  $m$  to be checked.

## 7. RELATED WORK

In the literature "approximate string matching" refers to the problem of finding a pattern string approximately in a text. There have been many studies on this problem. See [9] for an excellent survey. The problem of AME is different: searching in a long text to approximately match a string from a dictionary. In addition, AME is also different to the problem of text document indexing (finding dictionary documents approximately containing a query string) and string similarity joins (identifying approximate matching string pairs, each from one of two columns of strings).

To measure the similarity of a pair of strings, generally all similarity functions can be categorized as token-based and character-based, depending on what they regard strings as: sets of tokens, or sequences of characters.

The token-based AME problem, as discussed in this paper, can be straightforwardly reduced to set similarity join [1, 4, 6, 10]. Paper [4] discussed the framework and implements of a primitive operator SSJoin for performing similarity joins, on which a variety of similarity functions can be applied. Paper [1] solved the similarity join problem by converting set-based similarity distance into hamming distance between binary vectors, and studying the number of shared segments of two divided vectors. In [2], Chakrabarti et al. proposed a 0-1 matrix-based AME filter. In this paper we showed that their approach touches upon a NPC decision problem, whose intractability we briefly prove in the Appendix.

As a complement to the token-based approach, the character-based approximate string-matching problem has been well studied by researchers [9]. Early methods handling the edit distance constraints mostly work on the relationship of edit distance and gram sharing [12]. Due to the dilemma in choosing gram length, [8] proposes VGRAM, namely variable-length gram

to address the problem. For non-gram-based approaches, Wang et al. uses inverted lists to index the neighborhood of dictionary strings, and enhances previous neighborhood generation methods by reducing the upper bound of the neighborhood size [13].

Another line of related work is on inverted list merging, because the filtration phase needs inverted list processing. In [7], this problem is formalized into T-occurrence problem, and three efficient algorithms are proposed. T-occurrence problem requires that the threshold T should be independent from any list nodes, which is not satisfied by our method (our threshold  $\min\{\tau(m), \tau(r)\}$  varies with the rid information in list nodes), the list processing technique in [7] is orthogonal to our solution here, and can be used (by some modification) on SIL index in a complementary manner.

## 8. CONCLUSION

In this paper, we studied the AME problem (Approximate Member Extraction). Under the framework of filtration-verification, we analyzed the issue of trading between the two phases, and proposed a new filtering condition and corresponding filter called SIL. Then we designed two algorithms for SIL: EvSCAN and its incrementally optimized version EvITER, which saves the cost of scanning some inverted lists by progressively maintaining a candidate evidence set of the current substrings. We also addressed the static threshold problem of previous filters, and gave a solution for it on our SIL. Finally we reported the performance of our filtering algorithms through theoretical and experimental analysis.

## 9. APPENDIX

Theorem 1 in [2] provides a method of filtering by converting it to a decision problem about 0-1 matrices. Here, we give the proof about its intractability. For convenience we call it *Constrained Solid Submatrix* problem and describe it as below:

*(Constrained Solid Submatrix problem)* Given a 0-1 matrix  $A$ , whose size is  $p \times q$  and two weight functions  $w_1(i)$  ( $1 \leq i \leq p$ ) and  $w_2(j)$  ( $1 \leq j \leq q$ ), we need to determine if there exists a subset  $I = \{i_1, i_2, \dots, i_r\}$  from the rows and a subset  $J = \{j_1, j_2, \dots, j_c\}$  from the columns such that for any  $i' \in I$  and  $j' \in J$ ,  $A[i'][j'] = 1$ , and  $w_1(i_1) + w_1(i_2) + \dots + w_1(i_r) \geq \delta$ ,  $w_2(j_1) + w_2(j_2) + \dots + w_2(j_c) \geq \tau$  ( $\delta$  and  $\tau$  are two given thresholds)

**THEOREM 4** (*intractability of Constrained Solid Submatrix problem*) *Constrained Solid Submatrix problem is NP-Complete.*

**Proof Outline:** We prove by reducing to *Balanced Complete Bipartite Subgraph problem*, which requires finding a  $K \times K$  complete bipartite subgraph in a given bipartite  $B = \langle V_1 \cup V_2, E \rangle$ . It is already proven to be NP-Complete (see page 196 in [5]).

For any instance of the *Balanced Complete Bipartite Subgraph problem*, let  $w_1(i) = 1, w_2(j) = 1, \delta = \tau = K$ , and construct a  $|V_1| \times |V_2|$  0-1 matrix, whose elements are assigned as follow:

$$A[i][j] = \begin{cases} 1 & \text{If the } i\text{-th vertex in } V_1 \text{ and } j\text{-th} \\ & \text{in } V_2 \text{ are adjacent.} \\ 0 & \text{Otherwise.} \end{cases}$$

Then we can show that  $A$  has a constrained solid submatrix if and only if the corresponding *Balanced Complete Bipartite Subgraph problem* has a solution. This concludes the reduction.

## ACKNOWLEDGEMENT

The authors are grateful to Prof. Chen Li for his suggestions to motivate this work and thanks anonymous reviewers for their constructive comments. This research was partially supported by the grants from 863 National High-Tech Research and Development Plan of China (No: 2009AA01Z133, 2007AA01Z155, 2009AA011904), National Science Foundation of China (NSFC) under the number (No.60833005) and Key Project in Ministry of Education (No: 109004).

## REFERENCES

- [1] A. Arasu, V. Ganti, R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918-929, 2006.
- [2] K. Chakrabarti, S. Chaudhuri, V. Ganti, D. Xin. An efficient filter for approximate membership checking. In *SIGMOD Conference*, 2008.
- [3] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.
- [4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.
- [5] M.R.Garey and D.S.Johnson. *Computers and Intractability: Guidance to the Theory of NP-Completeness*.
- [6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491-500, 2001.
- [7] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257-266, 2008.
- [8] C. Li, B. Wang, X. Yang, VGRAM: Improving performance of approximate queries on string collections using variable length grams. In *VLDB* 2007.
- [9] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31-88, 2001.
- [10] S. Sarawagi, A. Kirpal, Efficient set joins on similarity predicates. In *SIGMOD Conference*, 2004.
- [11] A. Singhal. Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35-43, 2001.
- [12] E. Sutinen and J. Tarhio. On using q-grams locations in approximate string matching. In *ESA*, pages 327-340, 1995.
- [13] W. Wang, C. Xiao, X. Lin, C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD Conference*, 2009.
- [14] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.
- [15] A. C. Yao and F. F. Yao. Dictionary loop-up with small errors. In *CPM*, pages 387-394, 1995.