The Token Distribution Filter for Approximate String Membership Checking

Chong Sun, Jeffrey Naughton
University of Wisconsin, Madison
{sunchong, naughton}@cs.wisc.edu

ABSTRACT

A common application over web data is to find all the strings in a collection of pages that match strings in a given dictionary. We consider the problem of extracting all the strings or substrings in a document (or a page) that approximately match some string in a given dictionary. The current state-of-art approach for this problem involves first applying an approximate, fast filter, then applying a more expensive exact verification algorithm to the strings that survive the filter. Many string filters, such as the length filter and prefix filter, have been proposed. However, we find many string filters are ineffective or inefficient in some problem scenarios. In this paper, we propose a new filter, the TDF (token distribution filter). We conduct experiments on both synthetic and real data sets, and show that for a wide class of problems it performs better than previously proposed filters.

1. INTRODUCTION

We consider a common and ubiquitous string membership checking problem: given a dictionary of strings, and a collection of documents, find all occurrences of strings or substrings in the documents that approximately match some string in the dictionary. String membership checking is used in many applications, including collecting customer feedback on products by scanning thousands of emails, aggregating evaluations of movies from many reviews and identifying mentions of entities to extract information from Web pages. For example in DBLife [1], the mentions of entities, e.g., researchers, publications and conferences, are located in many Web pages in the database community, and further structured information is collected to build a web portal. There has been a great deal of research on how to efficiently conduct both exact string membership checking [3, 7] and approximate string membership checking [6, 8, 10, 12, 17]. In this paper, we focus on approximate membership checking.

A naive solution to the string membership problem is to iteratively consider each document string (or substring), checking with the dictionary for matches. There is a growing consensus that a better way to do this is to adopt a filter-verification approach [17], in which some quick approximate filter is constructed based upon the dictionary strings, and then candidate strings in the documents are first passed through the filter and only the strings that pass the filter are subjected to a more expensive verification. Many filters, including the length filter, counter filter, position filter [12], prefix filter ([10, 17]), LSH filter [11], ISH filter [8], and others, have been proposed. These filters use only partial information about the dictionary strings (e.g., the length filter [12] uses string lengths while the prefix filter [10] uses prefixes of string tokens.) Since different filters use different information, different filters are good at eliminating strings with different properties. The question is what information we should use to build an effective filter and how can we efficiently conduct filter checking.

In this paper, we propose a new filter, which we call the *TDF* (Token Distribution Filter) to cover as much of the string information, e.g., string length and string tokens, as possible. The *TDF* partitions strings based on a token partitioning scheme and uses the number of tokens in each partition as the string signature. The key motivation for the *TDF* is that there is a chance that two strings s_i and s_j approximately match if the string tokens roughly fall into the same partitions for s_i and s_j; otherwise, s_i does not match s_j. In addition, we design a grid data structure to index the signatures of the dictionary strings to accelerate the checking of candidate document strings against these signatures. We show in experiments that the *TDF* performs well on both synthetic and real data sets.

2. RELATED WORK

Approximate string matching for a pair of strings is a classical computer science problem and many algorithms have been proposed for its solution [15]. As an extension to many-many comparisons, the approximate string join (or string similarity join) assumes that we have two string collections and identifies all the approximately matching string pairs, with one string from each collection. Most work [5, 10, 12, 16, 17] on the approximate string join problem considers each string to be a set of tokens (e.g., q-Grams, words), and the similarity of two strings is measured based on the number of overlapping tokens in the two strings.

Gravano et al. [12] exploit the existing facilities in commercial databases to support approximate string joins based on tokenizing each string into a set of q-Grams. In [10], a database primitive query operator $ssjoin$ (set similarity join) is implemented to handle string joins. To efficiently evaluate approximate string joins, most recent work adopts the filter-verification framework [17], in which we first apply a rough, fast filter, and only do a detailed check for similarity if a string in one collection passes the filter built on the signatures of strings in the other collection.

Many signature schemes have been proposed. The counter filter, length filter and position filter, presented in [12], use the edit distance similarity. The prefix filter was proposed in [10] and later extended in [17]. A novel signature scheme PartEnum, based on *partitioning* and *enumeration* was presented in [5], in which by con-

 Fourteenth International Workshop on the Web and Databases (WebDB 2011), June 12, 2011 - Athens, Greece.

trolling the number of the string partitions, we guarantee that any two approximately matching strings must be the same in at least one or more partitions. LSH (locality sensitivity hashing) [11], exploits the property that similar strings have similar hash values, but it may not return all the approximately matching strings (the complete result). We do not consider filters that return an incomplete result in this work.

Approximate string membership checking ([8, 9, 14]) is another variation on the approximate string matching problem, in which we view one string collection as the dictionary and process a second collection of strings (or documents) to check if each string in the second collection approximately matches some dictionary string. In [14], efficient exact algorithms are proposed to conduct approximate string checking based on merging token inverted lists. In [8], the ISH (Inverted Signature-based Hashtable) structure is presented with the focus on reducing the filter checking time. In contrast to the inverted list, the ISH filter stores a list of string signatures, rather than the string ids, for each string token. The work in [4, 13] focuses on using cosine similarity metrics based on TF/IDF to approximately match one string against the strings in a large database.

3. PRELIMINARIES

3.1 Approximate String Membership Checking (ASMC)

Assume we have a string dictionary R and that each string $s_i \in R$ consists of a sequence of tokens (e.g., words, phrases or q-Grams) $\langle t_1, t_2, \ldots, t_l \rangle$. An input string s' is a member of dictionary R if and only if there exists a string $s_r \in R$ such that $sim(s', s_r) \geq \tau$, where $sim(s', s_r)$ is a string similarity function and τ is a similarity threshold. Formally, we define the approximate string membership checking (ASMC) problem as in [8]:

DEFINITION 3.1. *Given a string dictionary R, a string similarity function $sim()$ and a similarity threshold τ, find every string or substring s' in a document S such that $\exists\ s_r \in R$ and $sim(s', s_r) \geq \tau$, in which case we say s' approximately matches s_r.*

Several similarity metrics have been proposed, with *edit similarity (es)* and *Jaccard similarity (js)* perhaps the most broadly used. In this paper, we focus on the ASMC problem using js as the similarity metric. Given two strings s_i and s_j, $sim(s_i, s_j)$ is defined according to *es* and *js* as follows.

DEFINITION 3.2. *The* edit similarity *of two strings s_i and s_j,* es(s_i, s_j)*, is defined as $1 - \frac{ed(s_i,s_j)}{max(|s_i|,|s_j|)}$, in which $ed(s_i, s_j)$, the edit distance of s_i and s_j, refers to the minimum number of edit operations (i.e., deletions, insertions, and substitutions) over tokens or characters to transform s_i into s_j.*

DEFINITION 3.3. *The* Jaccard similarity *of strings s_i and s_j,* js(s_i, s_j)*, is defined as $|s_i \cap s_j|/|s_i \cup s_j|$, where $|s_i \cap s_j|$ refers to the number of common tokens in s_i and s_j, and $|s_i \cup s_j|$ refers to the total number of distinct tokens in s_i and s_j.*

In the following sections, unless explicitly stated otherwise, we say that two strings s_1 and s_2 approximately match using js if $sim(s_1, s_2) \geq 0.8$. According to [8], if $ed(s_i, s_j) \leq \epsilon$, then $|s_i \cap s_j| \geq max(|s_i|, |s_j|) - \epsilon$. Therefore, we have

$$es(s_i, s_j) \leq 1 - \frac{max(|s_i|, |s_j|) - |s_i \cap s_j|}{max(|s_i|, |s_j|)} \leq \frac{|s_i \cap s_j|}{|s_j|},$$

based on which, the *es* of strings s_i and s_j is bounded by an expression built on the number of the string tokens, similar to *js*. It is straightforward to extend the string filters used to solve the ASMC problem in Section 4 based on *js* to string filters based on es.

3.2 Filter-verification Framework

A naive approach to the ASMC problem is to iteratively take each string or substring s' in an input document and compute the similarity between s' and every dictionary string, outputting s' as a dictionary member if s' approximately matches some dictionary string. Generally this approach is expensive.

The filter-verification approach eliminates many candidate strings using filters before computing the string similarity function. As the name suggests, the approach includes two phases: filtering and verification. The filtering phase itself consists of two steps: building a filter over the dictionary strings, and checking the document strings against the filter. To conduct membership checking for a string document S and a dictionary R, we first build a filter f with R and then check each string in S with f. We say a string s passes a filter f if and only if f cannot guarantee that no string in R matches s. In the verification step, we compute $sim(s', s_r)$ for each string s' passing the filter f and each string $s_r \in R$. If $sim(s', s_r) \geq \tau$, we say s' approximately matches s_r and s' is a member of R. With an effective filter, many fewer candidate strings are checked than would be checked with the naive (no filter) approach.

Most string filters are built based on different string signature schemes, such as using the string length or the prefix [10] as the signature. A filter f built over a dictionary of strings consists of the signatures of all the dictionary strings, and typically also uses some index structure (e.g., ISH structure [8]) to speed the application of a filter to the strings.

4. FILTERING WITH THE TDF

We propose a new filter *TDF* based on string token distributions, and we design an index structure for the *TDF* to check candidate strings against a string dictionary.

4.1 The TDF

The *TDF* imposes a universal partitioning scheme ϕ on a string token space that splits all the tokens in the space into u partitions ($P = \langle P_1, P_2, \ldots, P_u \rangle$). This is reminiscent of but different from the approach in the PartEnum filter [5], which splits each string uniquely based on the similarity threshold. Correspondingly, a string s is partitioned as $P^s = \langle P_1^s, P_2^s, \ldots, P_u^s \rangle$. Each string partition $P_i^s \in P^s$ contains $|P_i^s|$ tokens from string s in the token partition P_i.

EXAMPLE 4.1. *Suppose we have an ordered string token space $P = [t_1, t_2, \ldots, t_{20}]$, and a partitioning scheme ϕ which splits P as $P_1 = [t_1, \ldots, t_5]$, $P_2 = [t_6, \ldots, t_{10}]$, $P_3 = [t_{11}, \ldots, t_{15}]$ and $P_4 = [t_{16}, \ldots, t_{20}]$. Unless explicitly stated otherwise, we use ϕ and P in the following examples. Suppose we have a string $s_1 = \langle t_2, t_3, t_7, t_8, t_9 \rangle$. Then s_1 is partitioned by ϕ as $P_1^{s_1} = [t_2, t_3]$ and $P_2^{s_1} = [t_7, t_8, t_9]$.*

In the following, before discussing the *TDF* in detail, we first describe the prefix signature [10] and the length signature [12], which influenced our design of the *TDF*.

4.1.1 Length Signature and Prefix Signature

Assume we have a v-token string $s_i = \langle t_1^{s_i}, t_2^{s_i}, \ldots, t_v^{s_i} \rangle$ and an m-token string $s_j = \langle t_1^{s_j}, t_2^{s_j}, \ldots, t_m^{s_j} \rangle$ ($m \geq v$). If s_i approximately matches s_j according to js with the similarity threshold τ, we have

$$|s_i \cap s_j|/|s_i \cup s_j| \geq \tau. \quad (1)$$

Based on Formula 1, we have the following lemma on the length of two strings that approximately match.

LEMMA 4.1. *Given two strings s_i and s_j as presented above, s_i matches s_j only if $\tau \cdot |s_i| \leq |s_j| \leq |s_i|/\tau$.*

A drawback of a length filter built on the length signature is that it can not differentiate strings with the same length. The prefix signature, on the other hand, considers only the tokens in the string prefix. Suppose s_i and s_j have l_c common tokens, s_i matches s_j only if $l_c/(m + v - l_c) \geq \tau$ based on Formula 1. Then s_i and s_j must have at least $l_c = (m + v) \cdot \tau/(1 + \tau)$ tokens in common to approximately match each other. Following that, we have Lemma 4.2.

LEMMA 4.2. *Suppose string s_i approximately matches s_j and the string prefix length is l_p. Then we have at least $(l_c + l_p - v)$ common tokens in the prefixes of s_i and s_j.*

The disadvantage of a prefix filter built on the prefix signature is that it only checks string prefixes, so it cannot differentiate between two strings with common prefix tokens.

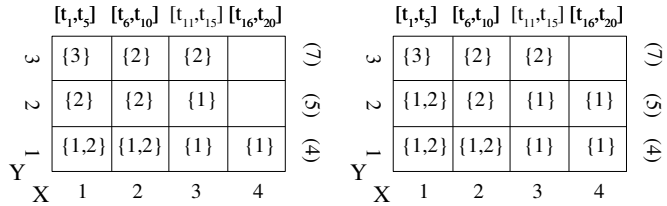
EXAMPLE 4.2. *Suppose we have two strings represented as $s_1 = \langle t_1, t_2, t_3, t_4, t_5, t_6, t_7\rangle$ and $s_2 = \langle t_1, t_2, t_3, t_8, t_9, t_{10}\rangle$. We note that s_1 will pass the length filter based on s_2, as $\tau \cdot |s_1| \leq |s_2| \leq |s_1|/\tau$; we also note that s_1 will pass the prefix filter based on s_2 if we check the 3-token prefix signatures of s_1 and s_2, in which $l_c = 6$ and $l_p = 3$, given $\tau = 0.8$. However, they do not approximately match.*

4.1.2 Token Distribution Filter

Both the length signature and the prefix signature only describe limited string information, and using the limited information in the length filter or prefix filter to check matching candidates can lead to many false positives. For example, the length filter is useless if every string to check has the same number of string tokens as some dictionary string. Similarly, the prefix string filter is not helpful if for most strings to check, there are some dictionary strings having similar prefixes.

Motivated by the drawbacks of the length filter and the prefix filter, we propose the *TDF* based on a new string signature scheme *TDS* (Token Distribution Signature), which is designed to contain more string information. Assume we have strings s_i and s_j, which are split into a set of u partitions as $P^{s_i} = \langle P_1^{s_i}, P_2^{s_i}, \ldots, P_u^{s_i}\rangle$ and $P^{s_j} = \langle P_1^{s_j}, P_2^{s_j}, \ldots, P_u^{s_j}\rangle$ based on the partitioning scheme ϕ. Intuitively, if s_i approximately matches s_j, then s_i should have roughly the same number of tokens in each space partition $P_i \in P$ as s_j. We refer to the number of string tokens in each partition of a string as the string token distribution. The String token distribution implicitly covers the string length information, as two strings with quite different lengths can not have similar token distributions over the whole token space. Similarly, with tokens following a same universal order, two strings having quite different prefixes or suffixes can not have similar token distributions either. Therefore, we use the string token distribution information as the string signature, which we call the *TDS*. Formally, we represent the signature of a string s as $\langle P_1(n_1), P_2(n_2), \ldots, P_u(n_u)\rangle$, in which $P_i (i \in [1, u])$ refers to the i_{th} partition and n_i refers to the token number of string s in P_i. Considering that string s may have no tokens in many partitions, we only keep $P_i(n_i)$ in the string signature if $n_i > 0$. Note that the space partitioning scheme affects the performance of the *TDF*. Suppose that all string tokens fall into one token partition. Then the *TDF* filter is similar to the length filter. Also, the *TDF* is similar to the prefix filter if we only check the first few string token partitions.

Next we discuss how we use the TDS in the *TDF* to check candidate strings. Suppose string s_i has $|P_x^{s_i}|$ tokens in $P_x \in P$. For any string s_j, the maximum number of common tokens of s_i and s_j outside the partition P_x is $M^{out} = min((|s_i| - |P_x^{s_i}|), |s_j|)$.



(a) Before update (b) After update

Figure 1: Signature Grid

THEOREM 4.3. *Suppose we have a partitioning scheme ϕ to split a token space into u partitions ($P = \langle P_1, P_2, \ldots, P_u\rangle$) and a string s_i. A string s_j approximately matches s_i based on js with similarity threshold τ only if s_j has at least $(\frac{(|s_i|+|s_j|)\cdot\tau}{1+\tau} - M^{out})$ tokens and at most $(\frac{(1+\tau)\cdot|P_x^{s_i}|+|s_j|-|s_i|\cdot\tau}{1+\tau})$ tokens in partition P_x.*

EXAMPLE 4.3. *Suppose we take the partitioning scheme ϕ that splits space $P = [t_1, t_2, \ldots, t_{20}]$ as $P_1 = [t_1, \ldots, t_5]$, $P_2 = [t_6, \ldots, t_{10}]$, $P_3 = [t_{11}, \ldots, t_{15}]$ and $P_4 = [t_{16}, \ldots, t_{20}]$. Furthermore, suppose we have a string $s_1 = \langle t_2, t_6, t_7, t_8, t_9\rangle$ represented as $\langle P_1(1), P_2(4)\rangle$. Any string s_2 of length 4 must have at most 1 token in P_1 and at least 3 tokens in P_2 to approximately match s_1 with similarity threshold $\tau = 0.8$ based on Theorem 4.3. Accordingly, string $s_2 = \langle t_2, t_3, t_4, t_6\rangle$ can not approximately match s_1.*

4.2 Using the TDF for Dictionary Strings

In this section, we address the problem of how to efficiently conduct approximate string membership checking with the *TDF*. Briefly, a *TDF* over a collection of dictionary strings is an index structure built over the *TDS*s of the dictionary strings, such that we can efficiently check each string against the *TDS*s of all the dictionary strings. A naive approach is to build a *TDF* f by collecting a set of *TDS*s for every dictionary string. Then, we compare the *TDS* of a candidate string s with each *TDS* in f. This approach is expensive if we have many string signatures in the filter. In the following, we design a structure we call the signature grid to index the *TDS*s of all the dictionary string in the filter. Then we discuss how we build the signature grid for the *TDF* and use the *TDF* to efficiently eliminate the no-matching candidate strings.

4.2.1 The Signature Grid

We design the signature grid consisting of a two-dimensional array (or a grid) to index the *TDS*s of the dictionary strings. We use the two one-dimensional arrays to describe how the grid is partitioned along each dimension, the string token dimension, X; and the string length dimension, Y. If the X dimension is split into n segments and the Y dimension is split into m segments, we have a total of $m \cdot n$ grid cells. We represent a grid cell as $c(x_i, y_j, N)$, in which x_i and y_j separately refer to the i_{th} segment in the X dimension and the j_{th} segment in the Y dimension; N refers to a set of integers. Each integer $v \in N$ represents that there is some dictionary string s, such that the length of s corresponds to some value in the j_{th} Y segment and s has v tokens in the i_{th} X segment.

Given a partitioning scheme on a token space, we incrementally build a signature grid for a string dictionary in the following two steps: (I) initializing the two one-dimension arrays and the grid and (II) updating the grid with the *TDS* of each dictionary string. In step (I), the token dimension array follows directly from the space partitioning scheme and each token segment corresponds to a token space partition as in Figure 1. We initialize the length dimension

input : A string s to check and a *TDF* f
output: A boolean variable pass

low =$|s| \cdot \tau$, high =$|s|/\tau$
foreach low $\leq$ l & $l \leq$ high **do**
  Compute Y segment value y according to l
  Set pass =true
  foreach *partition* P_i *in the TDS of* s **do**
    Compute X segment value x according to P_i
    Fetch the grid cell $c(x, y, N)$ from f
    lowB =$\frac{(l+|s|)\cdot\tau}{1+\tau} - min((|s| - |P_x^s|), l)$
    upB =$\frac{(1+\tau)\cdot|P_x^s|+l-|s|\cdot\tau}{1+\tau}$
    Set isIn =false
    foreach $v \in N$ **do**
      if $v \geq$ lowB & $v \leq$ upB **then**
        Set isIn =true
      end
    end
    if !isIn **then**
      Set pass =false; break
    end
    if pass **then**
      Return pass =true
    end
  end
  Return pass =false
end

Algorithm 1: $checkTDF(s, f)$

array with only one segment containing all the dictionary strings. We partition the grid according to the two one-dimension arrays.

In step (II), for each dictionary string s, we search and update the related grid cells. First, we use the string length $|s|$ to get the Y segment value y of the grid cells. If the current length dimension does not contain a segment for strings of length $|s|$, we add a segment for string length $|s|$ and partition the grid accordingly. Then, we compute the *TDS* for s according to the token partitioning scheme. For each partition P_x in the *TDS* with $|P_x^s| > 0$ tokens, we find the X segment value x corresponding to partition P_x and update the integer set N in grid cell $c(x, y, N)$ as follows: If $|P_x^s| \in N$, we do not change $c(x, y, N)$; otherwise, we add $|P_x^s|$ to N.

Assume the maximum length of the dictionary string is l_{max}, and a d-token domain is horizontally split into u partitions. Then, we can have $l_{max} \cdot u$ grid cells. The number of different integers in each cell is determined by the possible number of tokens falling into each cell, which is at most $(\frac{d}{u})$. We represent each integer with 1 byte and the space cost of the *TDF* is bounded by $l_{max} \cdot d$ bytes. The time cost of building a signature grid is linear in the number of strings in the dictionary.

EXAMPLE 4.4. *According to the partitioning scheme ϕ over the token space P, we build the signature grid in Figure 1 for the TDSs of 4 strings ($s_1 = \langle t_1, t_7, t_8, t_{13}\rangle$, $s_2 = \langle t_2, t_3, t_6, t_{17}\rangle$, s_3=$\langle t_1, t_2, t_3, t_8, t_9, t_{11}, t_{12}\rangle$, s_4=$\langle t_2, t_3, t_6, t_8, t_{14}\rangle$). Take grid cell $c(1, 1, \{1, 2\})$ in Figure 1a for example. We see that string s_1 and s_2 separately have 1 token and 2 tokens in the partition $[t_1, t_5]$. Suppose we incrementally build the grid with another string $s_5 = \langle t_1, t_7, t_9, t_{14}, t_{18}\rangle$. The TDS of s_5 is $\langle P_1(1), P_2(2), P_3(1), P_4(1)\rangle$, based on which we get Figure 1b by updating two grid cells: from $c(1, 2, \{2\})$ to $c(1, 2, \{1, 2\})$ and from $c(4, 2, \{\})$ to $c(4, 2, \{1\})$.*

4.2.2 Checking Candidate Strings with the TDF

After we build a *TDF* f with the signature grid on the dictionary strings, we use f to check each candidate string s, and eliminate s before verification if we can guarantee that s can not approximately match any dictionary string as described in Algorithm 1. Algorithm 1 takes a candidate string s and a *TDF* f as input and outputs a boolean value to indicate whether s can pass f. In Algorithm 1, we first compute the possible lengths of dictionary strings which may approximately match s, and for each partition in the *TDS* of s we fetch the corresponding grid cells and we check whether the *TDS* of s matches the *TDS* of some dictionary string in this partition. If for every partition in the *TDS* of s, the number of tokens of s and some dictionary string s satisfy the constraints specified in Theorem 4.3, we say s passes the filter; otherwise, s does not. The time complexity of the checking algorithm is $O((|s|/\tau - |s| \cdot \tau) \cdot |s|)$.

EXAMPLE 4.5. *Suppose we use the TDF f with the signature grid in Figure 1 to check two strings s_6=$\langle t_2, t_{11}, t_{16}, t_{17}, t_{19}\rangle$ and s_7=$\langle t_2, t_3, t_{11}, t_{19}\rangle$. Considering s_6, the TDS of which is $\langle P_1(1)$, $P_3(1)$, $P_4(3)\rangle$, we only need to check dictionary strings of length 4 or 5 (no string of length 6). For either length 4 or 5, in token partition P_4, the lower bound of token number for a dictionary string is 2 to match s_6. From the corresponding grid cells, we find no dictionary string of length 4 or 5 has more than 1 token in P_4. Therefore, s_6 does not match any dictionary string and can not pass the filter. By similar checking, we see that s_7 passes the filter f.*

4.2.3 Enhancing the TDF

The *TDF* considers the token distributions. However, it is possible that a string s may match the token distributions of the combination of several strings but none of them individually. To address this issue, we extend the *TDF* by adding a bit-array for each distinct token number in each token partition. We use the bit-array to keep the hash values of the ids of strings which have t tokens in partition P_i. If s approximately matches some string with t_1 tokens in P_i and t_2 tokens in P_j, there is chance that s approximately matches some dictionary string only if the bit-array corresponding to t_1 and the bit-array corresponding to t_2 contain at least a same string id hash value. By not only checking the grid, but also matching the bit-arrays, we increase the filtering ratio of the *TDF* . Through adjusting the bit-array sizes, we take the tradeoff between the memory cost and filtering ratio of the *TDF*.

EXAMPLE 4.6. *Suppose we have two dictionary strings $s_1 = \langle t_1, t_2, t_3, t_{12}, t_{13}, t_{14}\rangle$ represented as $\langle P_1(3), P_3(3)\rangle$ and $s_2 = \langle t_6, t_7, t_8, t_9, t_{13}, t_{14}\rangle$ represented as $\langle P_2(4), P_3(2)\rangle$. Then a string $s_3 = \langle t_1, t_2, t_3, t_6, t_7, t_8, t_9\rangle$ can pass the TDF built on s_1 and s_2. Suppose we build a bit-array b_1 for 3 in P_1 and b_2 for 4 in P_2. Assume the hash values of s_1 and s_2 are 1 and 2. Then b_1 contains 1 and b_2 contains 2. To check string s_3, which matches strings with 3 tokens in P_1 and 4 tokens in P_2, we get null by intersecting b_1 and b_2, which indicates s_3 can not match any dictionary string.*

4.3 Analysis of the TDF

The filtering ratio of the *TDF* is determined by the token distributions of both the dictionary strings and document strings to check, and also the token space partitioning scheme. Suppose we have a string $s = \langle P_1(n_1), P_2(n_2), \ldots, P_u(n_u)\rangle$. Assuming that the tokens of a n-string dictionary are uniformly distributed over the space partitions, the probability (filtering ratio) that s is filtered out by a *TDF* f built on this dictionary is

$$f^r(s) = \prod_{l=\lceil |s|\cdot\tau \rceil}^{\lfloor |s|/\tau \rfloor} (1 - \prod_{k=1}^{m} p(l, k, s)), \quad (2)$$

in which we use m to represent the number of non-empty token partitions of s, and use $p(l,k,s)$ to represent the probability of s passing f if we only check the k_{th} non-empty token partition of s with dictionary strings of length l. We get

$$p(l,k,s) = 1 - \prod_{t \in R(l)} (1 - \sum_{n=low(k)}^{high(k)} (1-\frac{1}{u})^{(|t|-n)} \cdot (\frac{1}{u})^n),$$

in which $R(l)$ refers to the set of strings of length l in dictionary R, and $low(k)$ and $high(k)$ represent the lower bound and upper bound on the number of tokens of a dictionary string t in the k_{th} non-empty partition of s, such that s can approximately match t.

To maximize $f^r(s)$, we minimize $p(l,k,s)$, which is affected by the number of tokens in each non-empty partitions of string s and the number of token space partitions u. We select the optimal u to minimize $p(l,k,s)$ as

$$u = argmin_i((1-\frac{1}{i})^{(|t|-n)} \cdot (\frac{1}{i})^n),$$

given a dictionary string t and the partitioning of string s.

By extending the *TDF* with bit-arrays, the filtering ratio is

$$\begin{aligned} f^r(s) &= \prod_{l=\lceil |s| \cdot \tau \rceil}^{\lfloor |s|/\tau \rfloor} (1 - \prod_{k=1}^{m} p(l,k,s) + \prod_{k=1}^{m} p(l,k,s) \cdot (1 - \check{p}(s))) \\ &= \prod_{l=\lceil |s| \cdot \tau \rceil}^{\lfloor |s|/\tau \rfloor} (1 - \prod_{k=1}^{m} p(l,k,s) \cdot \tilde{p}(s)) \end{aligned} \quad (3)$$

in which $\tilde{p}(s)$ refers to the probability that s can pass the *TDF* by checking the bit-arrays. Assuming the dictionary string ids are uniformly distributed in the b-bit bit-arrays, then we have

$$\tilde{p}(s) = \binom{b}{1} \cdot \prod_{k=1}^{m} \sum_{n=low(k)}^{high(k)} ((1-(1-(1-\frac{1}{b})^{c(n)})^{high(k)-low(k)}) \cdot \gamma),$$

in which $c(n)$ refers to the number of dictionary strings having n tokens in the k_{th} non-empty partition of s, and γ is represented as

$$\gamma = \begin{cases} 1 & low(k) > 0 \\ 0 & low(k) = 0. \end{cases}$$

We can either increase the bit array size b or the number of token space partitions m to decrease $\check{p}(s)$ to achieve a higher filtering ratio. The tradeoff is the time cost, which is $O(b \cdot \sum_{k=1}^{m}(high(k) - low(k)))$ to check the bit-arrays.

5. PERFORMANCE STUDY

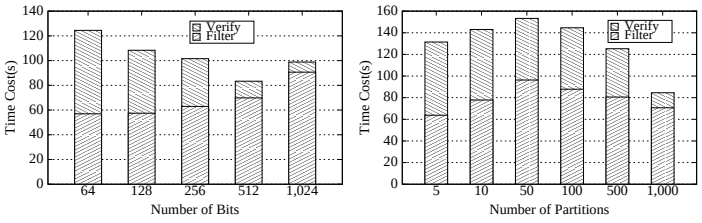
5.1 Experiment Setup

We ran all experiments on a 2.4 GHZ Intel Duo Core PC with 3GB memory. All the strings filters and the membership checking code was implemented in Java.

5.1.1 Data Sets

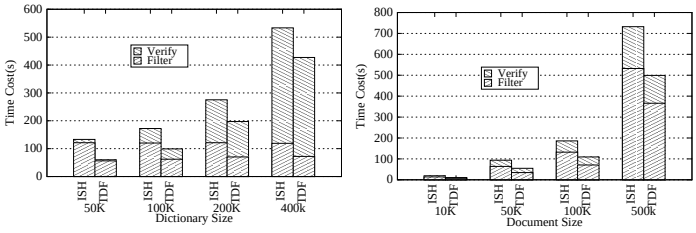
Synthetic Data: We designed a data generator consisting of a dictionary generator and a document generator, configured with the *shared* and *exclusive* parameters. We use the *shared* parameters to specify correlations between the generated dictionary and documents, and the *exclusive* parameters to specify the individual properties of the dictionary generator and document generator.

Exclusive parameters	Shared parameters
Token Space Coverage	Overall Token Space
Token Distribution	Token Space Overlap
Number of Strings	Overlap Strings
Average (Max, Min) Length	



(a) 1000 token partitions (b) 512 bits for each bit-array

Figure 2: Performance of the TDF on Synthetic Data Set



(a) 100K document strings (b) 100K dictionary strings

Figure 3: Performance of the TDF and ISH on Synthetic Data Set

Token Space Coverage refers to the set of tokens that a dictionary or document generator can use from the Overall Token Space, the universal token space used in the data generator. Token Space Overlap refers to the overlap ratio of the dictionary and document token spaces. We refer to the dictionary strings appearing in the documents as Overlap Strings.

We set default values for the synthetic data generator parameters as follows. For the dictionary generator: Token Space Coverage (5K); Token Distribution (*Normal*); Number of Strings (100K); Average, Min and Max Length (15, 10, 20). This configuration generates a dictionary with 100K strings composed of 5K distinct tokens with *Normal* distribution and the average, minimum and maximum string lengths are 15, 10 and 20. Similarly, we specify the default values for the document generator as: Token Space Coverage (5K); Token Distribution (*Normal*); Number of Strings (100K); Average, Min and Max Length (20, 10, 25). Furthermore, we set the Overall Token Space to contain 5K tokens, the Token Space Overlap to be 0.8 and the number of Overlap Strings to be 50 by default.

Real Data: We used a snapshot of the crawled DBLife [1] data (web pages) as the documents and a list of paper titles extracted from the DBLP Bibliography [2] as the dictionary. The DBLife data consists of 21607 web pages and the total data size is approximately 106MB. Many web pages, e.g., researchers' homepages, contain some publication mentions, which include paper titles. The dictionary consists of 200K paper titles with a total size 11.9MB and the number of distinct string tokens in the dictionary is approximately 85K.

5.1.2 Filters and Verification

We compared the performance of the TDF with that of the ISH filter [8], using string prefix signatures. The ISH filter is the state of the art filter used for approximate string membership checking. We used inverted lists to verify whether a string passing a filter can approximately match some dictionary string or not. We implemented the DivideSkip algorithm [14] for merging the inverted lists of tokens for the candidate strings. We used the time cost of the string filtering and verification to measure the ASMC performance.

5.2 Experimental Results

We show the TDF performance over synthetic data generated using the default values for the parameters in Figure 2. In Figure 2a, we vary the number of bits in each bit-array from 64 to 1024, while

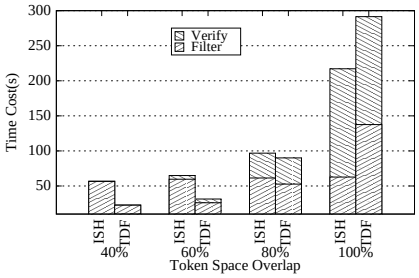
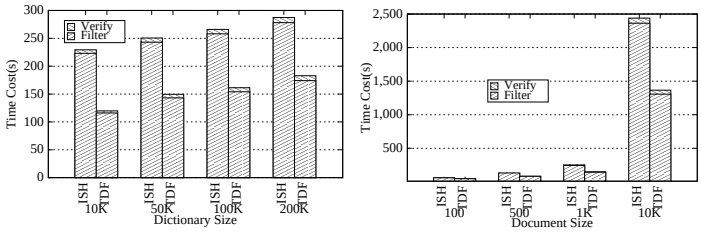


Figure 4: Performance of the TDF and ISH on Synthetic Data Set



(a) 1K Documents (b) 50K Dictionary Strings

Figure 5: Performance of the TDF and ISH on DBLife Data Set

keeping the number of token partitions fixed at 1000. We see that while increasing the number of bits in the bit-arrays, more strings are filtered out and the verification time is reduced. However, more time is spent on the string filtering. For this data set, we achieve the best performance when there are 512 bits for each bit-array. In Figure 2b, we vary the number of token partitions from 5 to 1000, while keeping the number of bits in each bit-array fixed at 512. We find that while increasing the number of token partitions, more strings are eliminated in the filtering and the verification time is reduced. However, the filtering time first increases and then decreases. This is because more bit-arrays need to be processed when there are more partitions. However, the number of "1"s in each bit-array decreases when there are more partitions and the processing time is reduced.

In Figure 3, we compare the performance of the TDF and ISH on synthetic data sets. We use 1000 partitions in the TDF and set the number of bits in each bit-array to be 256. Correspondingly, we set the bits in the ISH, such that the ISH takes the same amount of memory as the TDF. In Figure 3a, we use 100K document strings and vary the dictionary size from 50K to 400K strings. We have achieved consistently better performance in both the filtering and verification by using the TDF. In Figure 3b, we use 100K dictionary strings and vary the document size from 10K to 100K strings. We also find that using the TDF leads to better performance in both the string filtering and verification on this synthetic data set.

However, TDF is not always superior to ISH in all problem scenarios. We generated a set of synthetic data sets varying the token space overlap between the document strings and the dictionary strings from 0.4 to 1.0. We used default values for other parameters, except that we set the Average, Min and Max Length of the dictionary strings as (10, 7, 13). In Figure 4, we compare the performance of the TDF and the ISH. We find that when the token space overlap is 1.0, the ASMC using the ISH filter performs much better than using the TDF. The reason is that when there is a very large token space overlap between the dictionary and document strings, there is less chance that a string could be filtered out without checking the bit-arrays when using the TDF.

We also evaluated the performance the TDF on the DBLife data set, and we compare the performance of the TDF and ISH over that data set in Figure 5. We cut each dictionary or document string into a sequence of q-grams (we set q=4), and used q-grams as tokens to build filters and check strings. In Figure 5a, we use 1K documents, and vary the dictionary size from 10K to 200K strings. We see that with increasing the dictionary size, the time of the ASMC increases. Overall, using the TDF takes less time than using the ISH filter. In Figure 5b, we use a dictionary of 50K strings, and vary the number of documents from 100 to 10K. We also find that we get better performance by using the TDF.

6. CONCLUSION

The filter-verification approach is broadly used to solve the ASMC problem, and many string filters have been proposed. In this paper, we proposed a new filter TDF, and we showed that the TDF performed well on both synthetic and real data sets. However, we note the TDF is not always superior to other filters. The performance of a string filter may depend on many characteristics of the dictionary and document strings. Different string filters may be good at processing different dictionary and document strings. An interesting question for future work is to explore the possibility of using a set of filters, rather than a single filter, to achieve good performance over a wide range of data set characteristics.

7. REFERENCES

[1] *DBLife*. `"http://dblife.cs.wisc.edu/"`.

[2] *DBLP*. `"http://www.informatik.uni-trier.de/ ley/db"`.

[3] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. In *Commun. ACM*, 1975.

[4] N. K. Amit, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, 2004.

[5] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[6] A. N. Arslan and O. Egecioglu. Dictionary look-up within small edit distance. In *COCOON*, 2002.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.

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

[9] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, 2006.

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

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[12] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[13] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[14] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[15] G. Navarro. A guided tour to approximate string matching. In *ACM Computing Surveys*, 2001.

[16] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[17] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In *PVLDB*, 2008.