An Efficient Filter for Approximate Membership Checking

Kaushik Chakrabarti Surajit Chaudhuri Venkatesh Ganti Dong Xin

Microsoft Research
Redmond, WA 98052
{kaushik, surajitc, vganti, dongxin}@microsoft.com

ABSTRACT

We consider the problem of identifying sub-strings of input text strings that approximately match with some member of a potentially large dictionary. This problem arises in several important applications such as extracting named entities from text documents and identifying biological concepts from biomedical literature. In this paper, we develop a filter-verification framework, and propose a novel in-memory filter structure. That is, we first quickly filter out sub-strings that cannot match with any dictionary member, and then verify the remaining sub-strings against the dictionary. Our method does not produce false negatives. We demonstrate the efficiency and effectiveness of our filter over real datasets, and show that it significantly outperforms the previous best-known methods in terms of both filtering power and computation time.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Textual Databases*

General Terms

Algorithms

Keywords

Approximate Membership Checking, Filtering, String Match

1. INTRODUCTION

Given an input text string consisting of a sequence of tokens, the task of *approximate membership checking* (or, approximate dictionary lookup) is to identify all sub-strings that approximately match with some string from a potentially large dictionary. This problem has many applications. Most comparison shopping sites (*e.g.*, MSN shopping) have backend databases with product dictionaries. It is important for them to identify product mentions in documents (either on the web or in internal customer support databases) for mining sentiment about specific products. Here it is really crucial to match mentions which approximately match product titles in dictionaries. For instance, database entries such as "canon EOS digital rebel xti digital SLR Camera" are often concrete, while most web pages/reviews may not use the full description. Phrases such as "canon rebel xti digital SLR Camera" are commonly seen on web pages. Exact dictionary lookup will not catch all these mentions. Other applications include the dictionary-based biological concept extraction [21] and lookup-based network intrusion detection [16].

As shown in the above examples, in many cases the dictionary is known a priori, and the query strings are submitted on-the-fly. Suppose we are interested in sub-strings with length up to L. All sub-strings with l ($l \leq L$) tokens are possible candidates to match with a dictionary string. Since there are a large number of candidate sub-strings to be considered, a membership checking system generally preprocesses the dictionary and builds an efficient lookup structure such that candidates that do not match with any string in the dictionary can be quickly pruned.

We emphasize that the membership checking problem is different from the *text document indexing* and the *string similarity join* problems. In text indexing, documents are pre-processed and queries like "find all documents that contain a query string" are answered. Unlike the membership checking problem, in the text document indexing, long documents are given initially, and the incoming queries are short phrases (similar to strings in the dictionary). The string similarity join takes two collections of strings as input, and identifies all pairs of strings, one from each collection, that are similar to each other. Informally, we can consider one collection of strings as dictionary, and the other as query strings. The main task of string similarity is to find which string in the dictionary best matches with the query string. In contrast, membership checking problem identifies sub-strings which approximately match with a dictionary string.

The membership checking problem has been studied under the *exact-match* criterion [1, 4] and the *approximate-match* criterion [20, 2, 5, 11]. We focus on the latter scenario since it is very important in many real applications as described above. Previous approaches for the general approximate membership checking problem can be classified into two categories described below. The *inverted index* based approach [7, 18] builds a *rid* (*i.e.*, record id for dictionary strings) list for each distinct token in the dictionary. At query time, it retrieves and merges the *rid* lists of tokens in the query sub-string. After merging, the inverted index based approach

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

directly identifies the dictionary string that best matches with the query sub-string. However, the computational cost of repeated list-merge is very high, especially for a large dictionary. The *signature* based approach computes a set of signatures from both dictionary strings and query sub-strings. The signatures ensure that if a query sub-string is similar to a dictionary string, then their sets of signatures overlap. Examples of signatures include low frequent tokens [8, 7], subsets of tokens [3], and some proximity preserving hash codes [11]. The signature based approaches avoid merging at query time by matching signatures directly.

In this paper, we propose a new *ISH* filter based on a novel structure called *inverted signature-based hashtable*. The key insight we leverage is that *ISH* filter combines the benefit of the inverted index based approach [7, 18] and of the signature based approach [8, 7, 3]. The *ISH* filter has a structure similar to that of an inverted index, which stores a list of *rid*s per token. However, instead of a traditional inverted index structure, the *ISH* filter stores a list of signatures per token obtained by replacing each rid in the *rid*-list with the set of signatures of the string corresponding to the rid. The advantage is that we can quickly determine for a given query sub-string m whether a token's (in m) signature list contains any of the signatures generated from m. Depending on the number and weights of tokens which contain m's signatures we can quickly decide whether or not m can match with any string in the dictionary. Observe that these checks do not require us to merge the signature lists. We only lookup in each token's signature list whether m's signatures are present. This is a constant time operation per token in m. In contrast, inverted index based approaches merge *rid*-lists which is significantly more expensive and proportional to frequencies of tokens. We further compress the signature list using hashes. The resulting structure is small enough to typically fit into main memory[1].

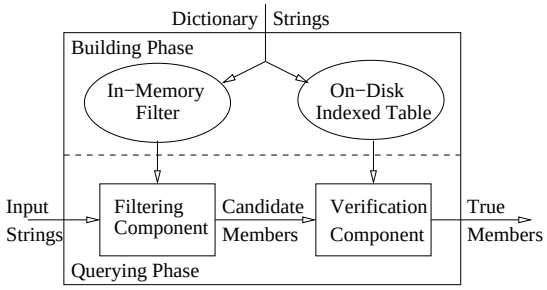

Figure 1: Overview of the framework

We refer to query sub-strings that are accepted by the *ISH* filter as *candidate members*, among which those sub-strings that really match with a string in the dictionary are *true members*, and those that do not are *false members*. To identify the true members, we further develop a verification component. The complete system framework is shown in Figure 1. At the offline building phase, we build the in memory *ISH* filter; and assume that the complete dictionary is stored and indexed in a relational database. We exploit previous methods for indexing the dictionary so as to allow approximate match queries [3]. At the online querying phase, the input text string is first submitted to the filter; and only those candidate members are further processed by the verification component. The main contributions of our paper are:

- We propose a novel filtering structure (the *inverted signature-based hashtable*), based on which we develop an effetive filter that is able to eliminate many candidate sub-strings for consideration;
- Our filter works with various similarity functions, including string edit similarity, jaccard similarity and weighted jaccard similarity;
- We develop a complete filter-verification system framework, and compare the *ISH* filter with the best known previous methods under the same framework.

The rest of this paper is organized as follows. The next section reviews the related work on membership checking. We give the formal problem statement in Section 3. Section 4 describes the inverted signature-based hashtable structure and presents the details of the *ISH* filter. Section 5 discusses options for verification. The complete algorithm is presented in Section 6. We report our experimental results in Section 7. Section 8 discusses possible extensions of our work. Finally, Section 9 concludes the study.

2. RELATED WORK

Previous work on membership checking includes *exact-match based* and *approximate-match based checking*.

2.1 Exact Match

The Aho-Corasick algorithm [1] and the Bloom Filter [4] are two well-known approaches used in exact-match based dictionary lookup. The Aho Corasick algorithm constructs a pattern matching machine (*i.e.*, a finite state machine) which simultaneously recognizes all occurrences of multiple patterns in a dictionary in a single pass through an input text. The trie-like automaton consists of two types of links: goto link and failure link. At the lookup phase, all of the matches are enumerated by moving along the input, following the links and keeping the longest match.

The bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. At the filter building phase, the bloom filter uses k different hash functions to map a dictionary string to k array positions in a bit-array where each bit is set to 0 initially. At the query time, the bloom filter uses the same set of hash functions to compute k positions for a query sub-string. If any bit on those positions is 0, the query sub-string is rejected.

2.2 Approximate Match

Similar to exact membership checking, approximate membership checking also has deterministic and probabilistic solutions. The deterministic approach has been studied in the context of d-queries: given a set of n binary strings with length m, for each binary string α with length m, find if there exists a string in the set with Hamming distance d with α. Typically, for $d = 1$, Yao *et al.* [20] and Brodal *et al.* [5] provided solutions by using trie-like data structures.

[1]For instance, for a product name dictionary which contains $10M$ dictionary strings with average length 7.3, the *ISH* filter only requires around $70MB$ memory space, which is fairly acceptable with modern computers.

Manber *et al.* [14] proposed a solution that generates all possible variations of each dictionary string, and insert all variations into a Bloom filters. This solution only works for very small d value (*e.g.*, $d = 1$). For reasonably large d, generating all variations is computational expensive.

For probabilistic solutions, Gionis *et al.* [11] proposed the *locality-sensitive hashing* (*lsh*) method for the equivalent nearest neighbor query problem. Their approach is approximate such that it may miss some true results. Roughly speaking, a locality sensitive hashing function has the property that if two points are close, then they hash to same bucket with high probability.

A closely related area of membership checking is the top-k query problem, for which a rich family of algorithms have been developed in the literature [10]. These methods generally construct an inverted index for each token in the dictionary string, and progressively merge multiple lists to compute the top-k results. Chandel *et al.* [7] studied the top-k dictionary lookup problem for all sub-strings (using moving windows) from an input text document, and presented a batch top-k algorithm.

The signature-based filtering idea was exploited in efficiently implementing string similarity joins [8, 3]. Chaudhuri *et al.* [8, 7] used low frequent tokens as signatures. Arasu *et al.* [3] improved the filtering power by using a subset of tokens as signatures.

3. APPROXIMATE MEMBERSHIP CHECKING

In this section, we first give the formal definition of the *approximate membership checking*, and then present a unified filtering condition for various similarity measures.

3.1 Problem Statement

A dictionary $\mathcal{R}$ is a set of strings r, each of which is a sequence of tokens $r = \langle t_1^r, t_2^r, \ldots, t_l^r \rangle$. In this paper, we use the term *token* to refer to the basic element in a string. A typical token is an English word. Alternative token types include characters, q-grams, or their hash values. An input string $\mathcal{S}$ (*e.g.*, a document) is a sequence of tokens $\mathcal{S} = \langle t_1^s, t_2^s, \ldots, \rangle$. Any sub-string $m = \langle t_i^s, t_{i+1}^s, \ldots, t_j^s \rangle \subseteq \mathcal{S}$ is a candidate member. m is a true member if there exists a dictionary string r such that $similarity(r, m) \geq \delta$. The approximate membership checking problem can be formally stated as follows.

DEFINITION 1. *Given a dictionary $\mathcal{R}$ and a threshold ϵ, extract all true members m ($|m| \leq L$) from input strings $\mathcal{S}$ such that there exists $r \in \mathcal{R}$, and $similarity(r, m) \geq \delta$.*

In this paper, we focus on three similarity measures: *edit similarity*, *jaccard similarity* and *weighted jaccard similarity*.

DEFINITION 2. *Given two strings r and m, the edit distance $ED(r, m)$ between them is the minimum number of edit operations (i.e., insertion, deletion, and substitution) to transform r into m. We define the edit similarity $ES(r, m) = 1 - \frac{ED(r,m)}{max(|r|,|m|)}$.*

DEFINITION 3. *Given two strings r and m, each of which is considered as a set, the jaccard similarity between them is defined as $JS(r, m) = \frac{|r \cap m|}{|r \cup m|}$, and the weighted jaccard similarity is defined as $WJS(r, m) = \frac{wt(r \cap m)}{wt(r \cup m)}$, where $wt(s) = \sum_{t \in s} wt(t)$, and $wt(t) \geq 0$ is the weight [2] of token t.*

EXAMPLE 1. *Suppose $\mathcal{R} = \{$ "canon eos 5d digital camera", "canon ef len" $\}$, and $\mathcal{S} =$ "The Canon eos 5d digital slr camera offers advanced photographers a lightweight, robust digital slr that uses Canon ef len without a conversion factor." $m_1 =$ "Canon eos 5d digital slr camera", and $m_2 =$ "Canon ef len" are two true members with jaccard similarity ($\delta = 0.8$).*

Approximate match is essential to adapt the membership checking problem in a noisy environment. However, it may also lead to redundant results. For instance, by setting $\delta = 0.7$, $m_3 =$ "uses Canon ef len" and $m_4 =$ "Canon ef len without" become true but redundant (w.r.t. m_2) results. In general, if we slightly extend a true member to the left (or to the right), the resulting sub-string may continue to be a true member, but with lower similarity score. We call this as *boundary redundancy*. To remove boundary redundancy, we require that the first and last tokens of m must be present in the corresponding dictionary string r.

3.2 A Unified Pruning Condition

For computational efficiency, a filter f avoids comparing m with every dictionary string $r \in \mathcal{R}$. Instead, f computes a upper bound of similarity between the query m and any $r \in \mathcal{R}$. In order to do this, the similarity measures need to be carefully rewritten. Interestingly, there exists a unified pruning condition for all three similarity measures used in the paper.

For *edit similarity*, Gravano *et al.* [12] concluded that if r and m are within an edit distance of ϵ, then $|r \cap m| \geq \max(|r|, |m|) - \epsilon$. According to Definition 2, we have

$$ES(r, m) \leq 1 - \frac{\max(|r|, |m|) - |r \cap m|}{\max(|r|, |m|)} \leq \frac{|r \cap m|}{|m|}$$

For (weighted) *jaccard similarity*, we have $JS(r, m) \leq \frac{|r \cap m|}{|m|}$ and $WJS \leq \frac{wt(r \cap m)}{wt(m)}$, respectively.

Setting $wt(t) = 1$ for edit similarity and unweighted jaccard similarity, we have a unified pruning condition for all three measures. That is, a candidate m is pruned if:

$$\max_{r \in R} similarity(r, m) \leq \frac{\max_{r \in R} wt(r \cap m)}{wt(m)} < \delta \qquad (1)$$

Since $wt(m)$ can be directly computed from the candidate m, it turns out that the main task of f is to compute the upper bound of $\max_{r \in R} wt(r \cap m)$. This unified pruning condition will be used in the rest of this paper, and unweighted measures are considered as weighted ones by assigning uniform weights to all tokens.

4. FILTERING BY ISH

Given an input string $\mathcal{S}$, all sub-strings with length up to L are candidate members. In this section, we present our new filtering strategy based on *ISH* (namely, Inverted Signature-based Hashtable). The *ISH* filter is able to efficiently identify the queries which cannot match with any dictionary string. Candidates that pass the filter $f(\mathcal{R}, \delta)$ will be verified (See Section 5).

[2] A typical weight configuration is the IDF [19] weight.

ISH filter is motivated by inverted indices [18, 7]. In inverted index, each token t is associated with a list of *rid*s. Given a query $m = \langle t_1, t_2, \ldots, t_l \rangle$, one can merge *rid*s from the inverted indices of t_i $(i = 1, \ldots, l)$, and aggregate weights of token t to a *rid* that appears in t's inverted index. Since the aggregated weight is exactly the value of $wt(r \cap m)$, the pruning condition (*i.e.*, Eqn. (1)) can be tested.

The *ISH* filter has a structure similar to that of an inverted index, which stores a list of *rid*s per token. The *ISH* filter instead stores a list of signatures per token obtained by replacing each *rid* in the *rid*-list with the set of signatures of the string corresponding to the *rid*. The advantage is that we can quickly determine for a given query sub-string m whether a token's signature list contains any of the signatures generated by m. Depending on the number and weights of tokens which contain m's signatures we can quickly decide whether or not m can match with any string in the dictionary. Observe that these checks do not require us to merge the signature lists. We only lookup in each token's signature list whether m's signatures are present. This is a constant time operation per token. In contrast, inverted index based approaches merge *rid*-lists which is significantly more expensive and proportional to frequencies of tokens.

Note that the number of signatures per token in the *ISH* filter is typically greater than the number of *rid*s. However, we hash the signatures (at the cost of a few more false positives) to a bit array to further compress the signature list. Thus signature lists are represented compactly.

Previous signature schemes are all *binary* such that as soon as one signature is shared by a query string and a dictionary string, they query string is considered a candidate for match. In *ISH*, we require multiple signatures to be matched simultaneously by extending the binary signature scheme to *weighted* signature scheme. That is, for each token in a candidate member, we count the number of matched signatures that occur in the token's signature list. Based on the weights associated with each signature we are now able to derive stronger pruning conditions than obtained by binary signature schemes. In the remaining of this section, we first extend the traditional signature scheme to the *weighted signature scheme*, and then present the details of the *ISH* filter. To begin with, Table 1 summarizes the notations.

Symbol	Meaning
δ	similarity threshold
$wt(t)$	the weight of a token t
m	a query string
r	a dictionary string
k	a parameter to control the number of prefix signatures
$\lambda(m)$	number of prefix signatures generated by m
$Sig(m)$	set of signatures generated by m
$\tau(m)$	derived threshold for signature merging

Table 1: Notations and Their Meanings

4.1 Weighted Signatures

We first give the general introduction on weighted signatures, and then demonstrate it by the prefix signature scheme [8, 7].

Concept: We classify the signature as *binary signature* and *weighted signature*. Informally, a binary signature scheme ensures that highly similar strings match on at least one signature. A weighted signature scheme on the other hand requires that highly similarity strings match on multiple signatures such that the "sum of weights" is greater than a threshold.

Given two string r and m, a binary signature scheme generates a set of signatures $Sig(r)$ for r and $Sig(m)$ for m. If $similarity(r, m) \geq \delta$, then $Sig(r) \cap Sig(m) \neq \phi$. *lsh*-signatures [11] is one example of binary signature. In weighted signature scheme, each signature s is associated with a weight $wt(s)$. If $similarity(r, m) \geq \delta$, then $wt(Sig(r) \cap Sig(m)) = \sum_{s \in Sig(r) \cap Sig(m)} wt(s) \geq \tau(m, \delta)$, where $\tau(m, \delta)$ is a threshold determined by the signature scheme, m and δ. As we show below, prefix-signatures can be extended to weighted signatures. When the context is clear, we use $\tau(m)$ to notate $\tau(m, \delta)$.

Example: Given a string r, prefix signature scheme sorts r in weights decreasing order, and extracts the prefix tokens whose aggregated weight is larger than $(1-\delta) \times wt(r)$. Each prefix token is a signature. The prefix signatures are generated for each candidate m in the same way. Prefix signatures are weighted signatures, and the weight of each signature is the weight of the corresponding token. The value of $\tau(m)$ is defined in the following Lemma.

LEMMA 1. *Let the prefix signatures for two string r and m be $Sig(r)$ and $Sig(m)$. If $similarity(r, m) \geq \delta$, then $\tau(m) = wt(Sig(m)) - (1 - \delta)wt(m)$* ∎

EXAMPLE 2. *Suppose $r =$ "canon eos 5d digital camera", $m =$ "Canon eos 5d digital slr camera", and the weights of tokens (digital, camera, canon, eos, 5d, slr) are (1, 1, 2, 2, 7 ,9), respectively. Assume $\delta = 0.8$, thus $WJS(r, m) = 0.909 \geq \delta$. Let $k = 3$, the prefix signatures[3] for r is $Sig(r) =$ {5d, eos, canon}, and that for m is $Sig(m) =$ {5d, slr, eos}. We have $\tau(m) = wt(Sig(m)) - (1 - \delta)wt(m) = 18 - (1 - 0.8) \times 22 = 13.6$, and $wt(Sig(r) \cap Sig(m)) = 16 \geq \tau(m)$.*

Number of Signatures: Note that for the same threshold, one can choose different number of prefix signatures. The minimal number of signatures corresponds to the shortest prefix such that $wt(Sig(r)) \geq (1 - \delta) \times wt(r)$. In the above example, the minimal number of signatures for r is $Sig(r) =$ {5d}. The maximal number of signatures is to include all tokens. Intuitively, generating more signatures requires more signature-matches at query time, and thus leads to stronger filtering power (this is contrast to earlier binary signatures). On the other hand, it needs more space. We use a parameter k to control the number of signatures. Let $\lambda(r, k)$ be the number of prefix signatures generated from r, we have:

$$\lambda(r, k) = \begin{cases} \lambda_{min}(r) & if \ \ \lambda_{min}(r) > k \\ \lambda_{max}(r) & if \ \ \lambda_{max}(r) < k \\ k & otherwise \end{cases}$$

where $\lambda_{min}(r)$ and $\lambda_{max}(r)$ are the minimal and maximal number of prefix signatures from r. We will further explore the issue of configuring k in Section 4.3. For simplicity, we use $\lambda(m)$ to notate $\lambda(m, k)$ when the context is clear.

[3]Ties are broken according to the reverse order in (digital, camera, canon, slr, eos, 5d)

4.2 Inverted Signature-based Hashtable

We are now ready to present the *ISH* filter. While *ISH* can take any signature schemes, we demonstrate our technique by the prefix signature in this paper. Extensions to other signature schemes are discussed in Section 4.4. We discuss the *building phase* and the *querying phase*.

Building Phase: In *ISH*, each token is associated with a signature hash table. The hash table is implemented as a bit-array. There is one bit-array for each distinct token. Let $BA(t)$ be the bit-array corresponding to token t (details on how to assign spaces for bit-arrays are discussed in Section 4.3.1). The *ISH* is created by unioning all the hashed signatures (*i.e.*, bits) for each token across all dictionary strings (Figure 2):

1. For each string $r = \langle t_1, t_2, \ldots, t_l \rangle \in \mathcal{R}$, compute $\lambda(r)$ signatures for r: $Sig(r) = \{s_1, s_2, \ldots, s_{\lambda(r)}\}$;
2. Compute the hash value for all signatures s_i ($i = 1, \ldots, \lambda(r)$), and get $l \times \lambda(r)$ array positions: p_{ij} corresponds to the position of s_i on bit array $BA(t_j)$ ($j = 1, \ldots, l$);
3. Set the bit at position p_{ij} on $BA(t_j)$ to 1 (all bits on $BA(t_j)$ are 0 initially).

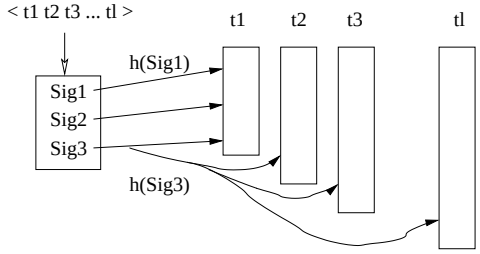

Figure 2: Inverted Signature-based Hashtable

EXAMPLE 3. *Suppose prefix signatures are used, and k is set to 3. Let* $\mathcal{R}$ = {r_1 = *"canon eos 5d digital camera"*, r_2 = *"nikon digital slr camera"*}, *and the weights of tokens (digital, camera, canon, nikon, slr, eos, 5d) be (1, 1, 2, 2, 2, 7 ,9), respectively.* $Sig(r_1)$ = {*5d, eos, canon*}, $Sig(r_2)$ = {*slr,nikon,camera*}. *The signatures and their hash values are shown in Table 2. For simplicity, we assume that all tokens are assigned the same bit-array size. After inserting* r_1 *and* r_2, *the bit-arrays of tokens are shown in Table 3 (assuming the bit-array index position starts from 0). Note the token canon appears in string* r_1, *and it co-occurs with signatures 5d, eos, canon. Thus, in Table 3, canon has value 1 on bit positions 0,3,5. The tokens canon, eos, and 5d all appear in the same set of strings (here, only* r_1*), and have the same set of signatures, and identical bit-arrays. Note that digital appears in two strings, and hence has a different (signature set and) bit-array from that of 5d.*

Querying Phase: To test a candidate $m = \langle t_1, t_2, \ldots, t_l \rangle$, we generate $\lambda(m)$ signatures for m, and apply the same hash function on signatures s_i ($i = 1, \ldots, \lambda(m)$) to get $l \times \lambda(m)$ array positions: p_{ij} corresponds to the position of s_i on bit-array $BA(t_j)$ ($j = 1, \ldots, l$). Let P be a $l \times \lambda(m)$ bit-matrix where row i ($i = 1, \ldots, \lambda(m)$) corresponds to s_i, and

Signature	Hash
5d	3
eos	5
canon	0
slr	4
nikon	3
camera	4

Table 2: Signatures and Hash Values

Token	Bit Array
canon	100101
eos	100101
5d	100101
digital	100111
camera	100111
nikon	000110
slr	000110

Table 3: Tokens and Bit-arrays

coloum j ($j = 1, \ldots, l$) corresponds to t_j. Cell $P[i, j] = 1$ if the bit array $BA(t_j)$ is set to 1 at position p_{ij}, otherwise, $P[i, j] = 0$. The bit-matrix can be seen as a small working-set corresponding to the current query. The number of columns in the bit-matrix is the number of tokens in the query. The number of rows in the bit-matrix is the number of signatures generated from the query string.

	canon	eos	5d	digital	slr	camera
s_1=5d	1	1	1	1	1	1
s_2=eos	1	1	1	1	0	1
s_3=slr	0	0	0	1	1	1

Table 4: Bit-matrix $P(Sig(m_1), m_1)$

	canon	digital	slr	camera
s_1=slr	0	1	1	1
s_2=canon	1	1	0	1
s_3=camera	0	1	1	1

Table 5: Bit-matrix $P(Sig(m_2), m_2)$

EXAMPLE 4. *Given the dictionary* $\mathcal{R}$ *and the* ISH *filter built in Example 3, Table 4 (the matrix* $P(Sig(m_1), m_1)$*) and Table 5 (the matrix* $P(Sig(m_2), m_2)$*) correspond to the query strings* m_1 = {*"canon eos 5d digital slr camera"*}, *and* m_2 = {*"canon slr digital camera"*}, *respectively. Suppose* $\delta = 0.8$. *Using the same prefix signatures,* $Sig(m_1)$ = {*5d,eos,slr*}, *and* $Sig(m_2)$ = {*slr,canon,camera*}. *Forthermore, In Table 4, the bit on column canon and row 5d is 1 because the bit corresponding to hash(5d) on the bit-array of canon is 1 (in Table 3).*

Each row in the matrix P corresponds to a set of signatures (the mapping is one-to-many due to the hash collision), and each signature maps to a set of dictionary strings where the signature is generated. Hence, every row in the matrix P represents a subset of dictionary strings that can possibly match with the query string. Suppose the subset of dictionary strings corresponding to the i^{th} *row* is R_i. Cells with value 1 indicate that the corresponding token is shared by the query string and one of the dictionary strings in R_i. Let $m' = m \cap R_i$ (*i.e.*, m' is the set of tokens whose corresponding bits are set to 1). The aggregated weight $wt(m')$ is a upper bound of $wt(m \cap r)$ for all $r \in R_i$. Thus, a necessary condition for m to match with any $r \in R_i$ is $wt(m') \geq \delta \times wt(m)$.

The weighted signature scheme requires multiple signatures to be matched simultaneously. That is, instead of looking for each individual row in the matrix P, we need to examine multiple rows at the same time. Let $Sig' \subseteq Sig(m)$ and $m' \subseteq m$. $P(Sig', m')$ is a sub-matrix of P by selecting

rows in Sig' and columns in m'. We say $P(Sig', m')$ is *solid* if all cells $P[i, j] \in P(Sig', m')$ are set to 1. Using $P(Sig', m')$, we can derive a necessary condition for m if m matches with a string r in the dictionary, as stated in Theorem 1.

THEOREM 1. *Suppose an* ISH *filter has been built based on the dictionary $\mathcal{R}$ and the similarity threshold is δ. For any candidate m, if there exists $r \in \mathcal{R}$ and similarity$(r, m) \geq \delta$, then there must exist a solid sub-matrix $P(Sig', m')$, such that:*

1. $wt(m') \geq \delta \times wt(m)$;
2. $wt(Sig') \geq \tau(m)$.

where $m' \subseteq m$ and $Sig' \subseteq Sig(m)$. ∎

EXAMPLE 5. *Continue on Example 4, we look for solid sub-matrices. From matrix $P(Sig(m_1), m_1)$, we find $Sig' =$ {5d,eos}, and $m' =$ {canon, eos, 5d, digital, camera} such that $P(Sig', m')$ is a solid sub-matrix, $wt(Sig') = 16 \geq \tau(m) = 13.6$, and $wt(m') = 20 \geq \delta \times wt(m) = 16$. Hence, m_1 is accepted as a candidate member. On the other hand, m_2 is pruned because there does not exist a sub-matrix that satisfies Theorem 1.*

4.3 Adapting to Memory Budget

Here we discuss how to determine the value of k. We first consider the case where the given memory budget M is sufficient to store the filter, and then consider the case where M is not large enough to hold the complete filter.

4.3.1 Complete Filter

We present a simplified analysis, which works well in selecting k in our experiment. Given the memory budget M, the computational factor that we consider is to achieve the best filtering power (*e.g.*, least rate of false positives). As shown in the last subsection, given a value of k, the space requirement of the *ISH* Filter is $N(k) = \sum_{t \in T} n(t, k)$, where T is the set of distinct tokens in $\mathcal{R}$, and $n(t, k)$ is the total number of signatures generated by all $r \in \mathcal{R}$ such that $t \in r$. We set the size of the bit-array $BA(t)$ to be $\frac{M \times n(t,k)}{\sum_{t \in T} n(t,k)}$.

For each signature, we assume that a hash function selects each position on a bit-array with equal probability. For each k value, let $\gamma(t, k)$ represent the expected proportion of bits in $BA(t, k)$ still set to 0 after all $r \in \mathcal{R}$ have been inserted.

$$\gamma(t, k) = (1 - \frac{N(k)}{M \times n(t, k)})^{n(t,k)} \approx 1 - \frac{N(k)}{M}$$

The rightmost term does not contain t, and we notate $\gamma(k) = 1 - \frac{N(k)}{M}$ thereafter.

Given a candidate $m = \langle t_1, t_2, \ldots, t_l \rangle$, let the set of signatures be $Sig(m) = \{s_1, s_2, \ldots, s_{\lambda(m)}\}$, and P be the $l \times \lambda(m)$ bit matrix where row i ($i = 1, \ldots, \lambda(m)$) corresponds to s_i, and column j ($j = 1, \ldots, l$) corresponds t_j. There are two cases that lead to cell $P[i, j] = 1$: *signature collision* and *hash collision*. The former happens if there exists an r such that $t_j \in r$ and $s_i \in Sig(r)$, and the latter happens if a corresponding bit on $BA(t)$ was set to 1 by other signatures. In our problem configuration, we want to control the memory requirement (*e.g.*, M is 2-3 times larger than $N(k)$). The probability of hash collision is then much larger than that of the signature collision. Thus, we consider the hash collision only, and the probability that $P[i, j] = 1$ is $1 - \gamma(k)$.

The weights of tokens and signatures of m may be chosen arbitrarily. Here, we use expected weights for tokens and signatures, which simplify our problem to the unweighted case. Consequently, the pruning conditions in Theorem 1 can be rewritten as follows. A candidate $m = \langle t_1, t_2, \ldots, t_l \rangle$ that does not approximately match with any $r \in \mathcal{R}$ will be falsely accepted if there exists a solid sub-matrix $P(Sig', m')$, such that $|m'| \geq \delta|m|$ and $|Sig'| \geq \tau(m, \delta)$. Typically, for unweighted prefix signature $\tau(m, \delta) = |Sig(m)| - (1 - \delta)|m|$.

For any signature $s_i \in Sig$, we use the notation $hit(s_i) = true$ if there are at least $\delta \times |m|$ 1s on the i^{th} of row P. For a given k, the probability of $p(hit(s) = true)$ is:

$$\beta(|m|, k) = \sum_{h=\delta|m|}^{|m|} \left(\begin{array}{c} |m| \\ h \end{array} \right) (1 - \gamma(k))^h \gamma(k)^{|m|-h}$$

To compute the probability of the presence of a solid sub-matrix with multiple signatures is rather complicated. We present a recursive method in Appendix. Let $\eta(k)$ be the probability of existing a solid sub-matrix $P(Sig', m')$, such that $|m'| \geq \delta|m|$ and $|Sig'| \geq \tau(m, \delta)$. We choose

$$k = argmin_i(\eta(i))$$

4.3.2 Partial Filter

Here we describe solutions when M is not sufficient to hold the complete filter. Our solution is to remove bit-arrays belonging to high frequent tokens. That is, we sort tokens in $n(t, k)$ decreasing order, and progressively remove $BA(t)$ until the remaining bit-arrays fit in M. Intuitively, high frequent tokens are similar to stop words. They appear in many strings in the dictionary, and are associated to a large number of signatures. Thus the probability of signature collision for high frequent tokens is relatively larger (the expected hash collision is same for all tokens). On the other hand, the high frequent tokens consume significant amount of memory space. In many applications, the frequency distribution of tokens follows the power-law distribution [7, 6]. We expect that by removing small number of tokens, the memory requirement of *ISH* filter reduces quickly. To avoid false negatives, for each t that $BA(t)$ is removed, any query against $BA(t)$ returns 1 (*e.g.*, assuming $BA(t)$ is full of 1). Hence, the reduced memory configuration may introduce additional false positives.

4.4 Other Signature Schemes

As we stated earlier, *ISH* filter is a framework which supports multiple signature schemes. We have demonstrated the filter by prefix signatures. Here we discuss how to incorporate other signature schemes in the framework. We use *locality-sensitive hashing* (*i.e.*, *lsh*) [13, 9, 15] as example. The extension to other signature schemes (*e.g.*, partenum [3]) should be similar.

The key idea in *lsh* is to hash a sequence of tokens as to ensure that for each hash function, the probability of collision is much higher for similar sequences than for dissimilar sequences. The process is probabilistic, and introduces both false positives and false negatives. In order to reduce the false negatives, l different signatures are computed. The classic implementation of *lsh* is *minhash*-based that concatenates g minhashes as a signature (details on minhashes can be found in [13, 9, 15]). To achieve false negative rate ω, l can be chosen as the minimal integer that satisfies $(1-\delta^g)^l \leq 1-\omega$, where δ is the similarity threshold.

Note that *lsh* is a binary signature scheme. By setting $wt(s) = 1$ for each *lsh* signature, and $\tau(w) = 1$ for the hit signature threshold, we can directly replace prefix signatures by *lsh* signatures in the building and querying phases.

5. VERIFICATION

For the sake of procedure completeness, we discuss verification in this section. One option is to use the batch verification that takes the complete set of candidate members and the dictionary input, and output $\langle m, r\rangle$ pairs where $similarity(r, m) \geq \delta$. This is basically a string similarity join problem [3, 8] discussed in Section 2.

Besides *batch verification*, we also want to support *one-at-a-time verification*, which is implemented as follows. Similar to the filtering module, the verification module also consists of two phases: the building phase and the querying phase. In the building phase, we create $\lambda(r)$ tuples $\langle id, r, hash\_sig, wt\rangle$ for each dictionary string r and each signature generated by r. Where $hash\_sig$ is the hash code of the signature, and wt is the weight of the string r. We store all tuples in a relational table V, and create a clustered index on $\langle hash\_sig, wt\rangle$. In the querying phase, we need to identify all matched signatures and compute the upper and lower bounds of $wt(r)$ to retrieve dictionary strings. Ideally, this should be computed from all solid sub-matrices that satisfy Theorem 1. This requires us to enumerate all sub-matrixes. An alternative solution is as follows.

For any candidate m that was accepted by the filter, let $\tilde{m}$ and $\tilde{Sig}$ be the set of conditional hit tokens and conditional hit signatures as defined below.

Definition 4. *Given a candidate m, for each token $t \in m$, t is a conditional (on m) hit token if there exists a set of signatures $\{s_1, \ldots, s_i\} \subseteq Sig(m)$, such that the cells corresponding to t and $s_1, \ldots, s_i$ are set to 1 and $wt(s_1) + \ldots + wt(s_i) \geq \tau(m, \delta)$. For each signature $s \in Sig(m)$, s is a conditional (on m) hit signatures if there exists a set of tokens $\{t_1, \ldots, t_j\} \subseteq m$, such that the cells corresponding to s and $t_1, \ldots, t_j$ are set to 1 and $wt(t_1) + \ldots + wt(t_j) \geq \delta \times wt(m)$.*

Clearly, for any solid sub-matrix $P(Sig', m')$ that satisfies Theorem 1, $Sig' \subseteq \tilde{Sig}$ and $m' \subseteq \tilde{m}$. Thus, for each $s \in \tilde{Sig}$, we retrieve the dictionary strings by:

$$Select * from\ V\ where$$
$$hash\_sig = hash(s)\ and\ \delta \times wt(m) \leq wt \leq \frac{wt(\tilde{m})}{\delta}$$

m is verified against the retrieved dictionary strings.

6. THE COMPLETE ALGORITHM

This section presents the complete algorithm for approximate membership checking. Given an input string $\mathcal{S}$, the algorithm tests all sub-strings with length up to L using the filter, and those candidate members are further submitted for verification. A high level description of the framework is illustrated in Algorithm 1.

We explain the algorithm line by line. Lines 1-2 construct the filter and index dictionary strings in the dictionary $\mathcal{R}$. This procedure is conducted offline. The compact filter resides in memory, and we assume the dictionary is stored on disk. Lines 3-5 generate query strings with length up to L. The filter f is applied on lines 6. Finally, candidate members passed f are verified in line 7. Note Algorithm 1 verifies each candidate member one-at-a-time. Alternatively, one can keep all candidate members in a *candidate_set*, and then issue a batch verification at the end of the execution.

Algorithm 1 Approximate Membership Checking

Input: $\mathcal{R}$, δ, $\mathcal{S} = \langle t_1, t_2, \ldots, \rangle$

```
1: Build the filter f(R, δ) ; //offline
2: Index R for verification; //offline
3: for (start = 1 to |S| − L + 1)
4:     for (length = 1 to L)
5:         m ← ⟨t_start, t_{start+1}, t_{start+length−1}⟩;
6:         if (f.prune(m) == true) continue; //filter
7:         if (∃r ∈ R, s.t. similarity(r, m) ≥ delta) //verify
8:             Output m;
```

7. PERFORMANCE STUDY

We now report our experimental results. We compare the performance of the *ISH* filter (referred as *ISH-Filter*) with two state-of-the-art methods: the *lsh*-signature based filter [3] (referred as *LSH-Signature*) and the segmented index merging [7] (referred as *Segmented-Merging*). All the experiments are conducted on a 2.4GHz Intel Core 2 Duo PC with 4GB RAM. We use Microsoft SQL Server 2005.

We use real data sets for the experiments. The dictionary set is a collection of $10M$ product names (*e.g.*, electronics, book titles, furniture, *etc*). Example dictionary strings are "The Food of the Western World An Encyclopedia of Food from North America and Europe" and "Microsoft Wireless Notebook Optical Mouse 400". Table 6 shows some statistics on the product name data set.

Parameter	value
Number of Dictionary Strings	10,000,000
Distinct Number of Tokens	2,421,627
Maximal Length	40
Minimal Length	1
Average Length	7.3

Table 6: Statistics on Product Name Data Set

The input string is a collection of 10,000 documents. On average, each document contains 3,689 tokens (each token is an English word).

7.1 Preliminary Filtering Techniques

Before we discuss the experimental configuration for all three methods, we first describe two basic filtering techniques. These methods may not achieve the desired filtering power of any approach used in our experiment. But they have very low computational overhead, and can be integrated with any advanced filters.

7.1.1 Filtering by Token-table

The first method maintains a token hash table of all distinct tokens appearing in $\mathcal{R}$ (notated as $TT(\mathcal{R})$). In general, even for very large $\mathcal{R}$, the number of distinct tokens may still be much smaller. We assume $TT(\mathcal{R})$ can fit in memory.

Hit Tokens: For each token t in the candidate m, we call t is a hit token if $t \in m \cap TT(\mathcal{R})$. Clearly, m can be safely pruned if $\sum_{t \in TT(\mathcal{R}) \cap m} wt(t) < \delta \times wt(m)$.

Strong Tokens: Given a string $r \in \mathcal{R}$, all tokens $t \in r$ can be sorted in decreasing order of their weights, and then divided into two parts: *strong* and *weak* (as suggested by [7, 8, 17]). The strong set consists of the shortest prefix of tokens whose aggregated weight is larger than $(1-\delta) \times wt(m)$. Intuitively, for any candidate m, if there exists a string r such that $similarity(r,m) \geq \delta$, then there is at least one token t from the strong set of r, and $t \in r \cap m$. Based on this observation, for each token $t \in TT(\mathcal{R})$, we maintain a boolean value $strong(t)$ such that $strong(t) = true$ if and only if t belongs to the strong set in at least one $r \in \mathcal{R}$. Consequently, a candidate m can be pruned if no token in m is strong.

7.1.2 Handling Short Candidates

The second method applies exact-match module to match short candidates m against a set of pre-computed variations of all $r \in \mathcal{R}$. Let l_E be the maximal candidate length for exact-match (*e.g.*, $l_E = 3$). We discuss how to handle candidate whose length is no larger than l_E for unweighted and weighted measures separately.

Unweighted Measures: Intuitively, for a short candidate m, even the smallest difference (*i.e.*, by differing one token) from a string r may lead to $similarity(r,m) < \delta$. More specifically, let $l(\delta) = \frac{\delta}{1-\delta}$. For any $m \neq r$, and $|m| < l(\delta)$, we have $similarity(r,m) \leq \frac{|m|}{|r|} < \frac{l(\delta)}{l(\delta)+1} = \delta$. Let $l_E < l(\delta)$, then for any candidates m ($|m| \leq l_E$), m is a true member if and only if there exists $r \in \mathcal{R}$, such that $r = m$. This is an exact-match problem. We can apply exact-match based membership checking methods [1, 4] to store all strings r ($|r| \leq l_E$), and extract true members whose length is no longer than l_E efficiently.

Weighted Measures: Unlike the unweighted measures, weighted similarity (*i.e.*, WJS measure) can not be bounded by the length of candidates due to different token weights, and thus all strings in $\mathcal{R}$ should be considered for approximately matching to m ($|m| \leq l_E$). For each string $r \in \mathcal{R}$, we enumerate all $r' \subseteq r$, such that $wt(r') \geq \delta \times wt(r)$, and $|r'| \leq l_E$. We then store pairs $\langle r', wt(r) \rangle$ in the exact matching structure.

For a candidate m ($|m| \leq l_E$), the matching scenario can be divided into two categories:

1. There exists an r' such that $r' = m$. Since $\frac{wt(r')}{wt(r)} \geq \delta$, m is a true member.
2. There exists an r' and a $m' \subseteq m$ such that $r' = m'$. m is true member if $\frac{wt(r')}{wt(r)+wt(m)-wt(r')} \geq \delta$.

7.2 Experimental Configuration

We apply the basic filtering techniques described above for all three methods. The maximal length l_E for exact match module is set to 3. The maximal candidate length is set to $\frac{max\_len}{\delta}$, where max_len is the maximal length of dictionary strings. Thus, the candidate queries submitted to all methods have length from 4 to $\frac{max\_len}{\delta}$.

LSH-Signature: We implement an in-memory structure by combining the *lsh*-signature scheme used in [3] and a Bloom Filter [4]. For *lsh*-signatures, we set the false negative rate ω to 0.95, which means only 95% of correct results will be output. The observed accuracy is close to the expected one in our experiments. The performance of *lsh*-signature depends on the parameters g and l. We report the results from the optimal setting (*i.e.*, the one achieves the best filtering power). For each string in the dictionary, we generate l signatures, and insert them into a bloom filter, which is configured as follows: the size (in bits) of the bloom filter is 8 times the total number of signatures, and the number of hash functions used by the bloom filter is 5, which is derived from the optimal ratio of hash-functions to bits [4].

Segmented-Merging: We implemented an improved version of the segmented index merging approach proposed by [7], which was developed for a batch top-k query problem for an input string. The segmented merging strategy was shown to be more efficient than the traditional top-k merging [10] and progressive merging that shares computation among neighboring query sub-strings. The main idea of *Segmented-Merging* is to only build inverted indices for strong tokens. Since it loses *rid* lists from weak tokens, the *Segmented-Merging* also has false positives. For fair comparison, we store all inverted indices in memory. At the query phase, only strong tokens are merged, and non-strong tokens are considered as "hit". An upper bound similarity score can be computed for a candidate. The original algorithm was developed for top-k retrieval, where the *rid* lists are maintained sorted according to the aggregated similarity score. Since the membership checking problem only needs to know whether the candidate presents in the dictionary or not, we avoid sorting by maintaining a maximal aggregate score.

ISH-Filter: We use prefix signatures in the experiments for two reasons. First, they are simple and efficient to compute; and second, they can be extended to weighted version which generally performs better than binary signatures. We also compare the performance of prefix-signature based *ISH* and *lsh*-signature based *ISH*. Let the total number of tokens in the dictionary be D. We configure the memory budget in terms of the dictionary size. By default, we allocate $b = 8$ bits for each token. Hence, M is set to $b \times D$ (bits). For our product name dictionary, $D = 73,000,000$, and $M \approx 70MB$. For *ISH-Filter*, we need to determine the parameter k for signature generation. We use the estimation method described in Section 4.3.1. $k = 3$ is the best choice for most cases (by varying δ and dictionary size). Therefore, we fix $k = 3$ for *ISH-Filter*.

We briefly discuss how to remove boundary redundancy for all three methods. First, all methods can use the token table to identify hit tokens. A candidate is directly pruned if either the first token or the last token is not a hit token. Secondly, for *Segmented-Merging*, we only consider strings that contain both the first and the last tokens (*e.g.*, *rid*s appearing in the both lists of the first and last tokens). Finally, for *ISH-Filter*, we require both first and last tokens to be *conditional hit tokens* (Definition 4).

7.3 Experimental Results

The computational factors that we consider are: *execution time, memory requirement* and *filtering power*. The filtering power is measured by the numberof candidate members. This is also directly related to verification cost. We conduct experiments on both *filter-only* and *filter-verification* configurations.

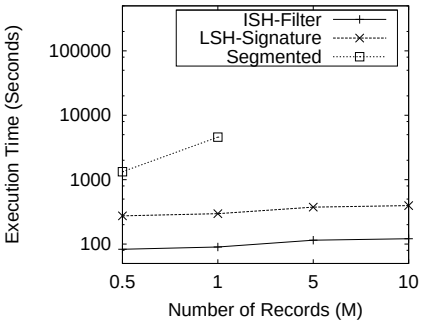

Figure 3: Execution Time w.r.t. $|\mathcal{R}|$, Weighted Measure, $\delta = 0.85$

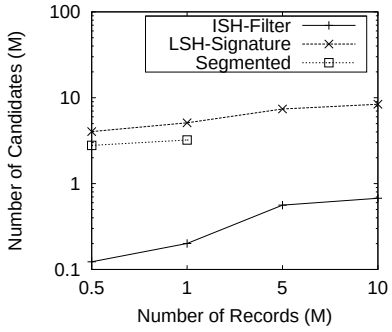

Figure 4: Filtering Power w.r.t. $|\mathcal{R}|$, Weighted Measure, $\delta = 0.85$

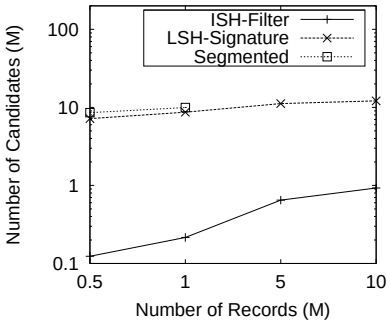

Figure 5: Filtering Power w.r.t. $|\mathcal{R}|$, Unweighted Measure, $\delta = 0.85$

We summarize the experimental results as follows. In terms of the execution time, *Segmented-Merging* is significantly slower (by two order of magnitude) than the other two alternatives, and *ISH-Filter* is slightly (around 2 times) faster than *LSH-Signature*. In terms of the filtering power, *ISH-Filter* generates much less candidate members (often, by an order of magnitude). *Segmented-Merging* uses the most memory space, and *ISH-Filter* uses 2 times more memory than *LSH-Signature* by default configuration. We also conduct experiments by further reducing the memory requirement of *ISH-Filter*, and show that even using $\frac{1}{10}$ of the default memory, the filtering power of *ISH-Filter* is still comparable to *LSH-Signature*.

7.3.1 Filter-only

This subsection reports the experimental results with filter-only approaches. Since we use a unified pruning condition, the filter-only configuration does not involve specific similarity measures. Instead, we report results for weighted and unweighted measures. For weighted measures, we assign each token the standard IDF weights [19] derived from the dictionary. To study the scalability of the method, we vary the size of dictionary by using $500k$, $1M$, $5M$ and $10M$ dictionary strings from the product name data set. Figure 3 to 9 show the execution time, filtering power and memory usage. All experiments use all $10,000$ documents as query string. The performance on weighted and unweighted filtering is similar.

Filter Building Time: Here we briefly report the costs to build filters for each method. For weighted measure, *ISH-Filter*, *LSH-Signature* and *Segmented-Merging* use 980, 820, 290 seconds (for $\delta = 0.85$), respectively, to build a filter for $10M$ dictionary strings. The corresponding building time for unweighted measure are 840, 590 and 270 seconds. All building times include the cost to build the basic filter. *ISH-Filter* uses slightly more time because it needs an additional dictionary scan to compute $N(k)$ to determine the best k value and initialize space for bit arrays. For *ISH-Filter* and *LSH-Signature*, the construction time slightly increases (decreases) when δ decreases (increases).

Execution Time: Figure 3 shows the execution time for the weighted measure. First of all, We observe that *Segmented-Merging* runs significantly slower than the other two alternatives. The computation complexities of both *LSH-Signature* and *ISH-Filter* are independent of the dictionary size. Figure 3 shows that the execution time of both *LSH-Signature* and *ISH-Filter* slightly increases. This is because when the dictionary is small, the basic filtering techniques are more effective. *Segmented-Merging*, although only retaining the strong tokens, has computational cost proportional to the dictionary size. We only report results of segmented index merging for dictionary size $500k$ and $1M$. The experiments on dictionary size $5M$ and $10M$ did not finish within our time limit.

The execution time of *LSH-Signature* and *ISH-Filter* are similar for unweighted measure, while that of *Segmented-Merging* is even worse. For instance, setting $|\mathcal{R}| = 1M$, the *Segmented-Merging* finishes in $63,233$ seconds (versus $4,592$ seconds on weighted measure). This is because there are more strong tokens for unweighted measure.

The advantage of index merging is that it keeps *rid* in inverted indices, and thus it is possible to compute the exact similarity score for a candidate member without verification. On the other hand, this verification-free configuration need to keep inverted indices for all tokens (instead of only strong tokens in *Segmented-Merging*), which is obviously more computational expensive. As we show in the next subsection, even including on-disk verification, the *LSH-Signature* and *ISH-Filter* are orders of magnitude faster than *Segmented-Merging* (without verification).

Note both *LSH-Signature* and *ISH-Filter* can be implemented more efficiently by applying progressive computation: the *LSH-Signature* may be further improved by progressively computing minhashes; and the *ISH-Filter* may be further improved by keeping the previous bit-lookup results since it is highly likely that some prefix signatures are shared by the neighboring candidates. We will further explore the progressive computation in Section 8. As shown in Figure 3, *ISH-Filter* is about two times faster than *LSH-Signature*. This is because: (1) the computation of prefix-signatures is cheaper than that of minhashes in *lsh*-signature; and (2) the bit-lookups against *ISH* are also cheaper than that against the Bloom Filter (*i.e.*, involving multiple hash computation).

The execution time is also related to the similarity threshold δ. For instance, by varying the similarity threshold from 0.8 to 0.9, the execution time of *ISH-Filter* decreases from 141 seconds to 109 seconds (with dictionary size $10M$).

Filtering Power: Figure 4 and Figure 5 compare the number of candidate members generated by all three methods. We exclude the outputs generated by the exact match module since they do not need verification. We observe that

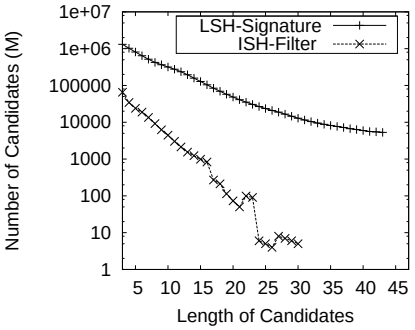

Figure 6: Filtering Power w.r.t. Length, Weighted Measure, $|\mathcal{R}| = 10M$, $\delta = 0.85$

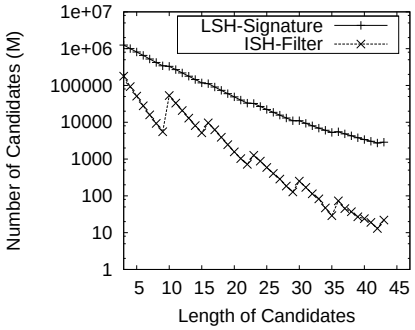

Figure 7: Filtering Power w.r.t. Length, Unweighted Measure, $|\mathcal{R}| = 10M$, $\delta = 0.85$

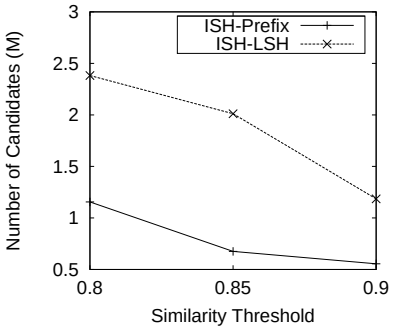

Figure 8: Filtering Power w.r.t. Signature Schemes, Weighted Measure, $|\mathcal{R}| = 10M$

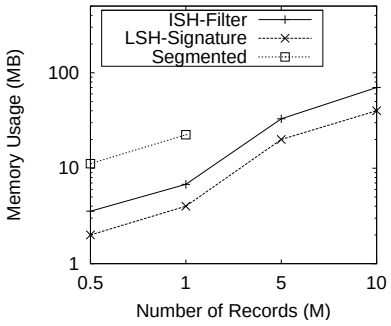

Figure 9: Memory Usage w.r.t. $|\mathcal{R}|$, Weighted Measure, $\delta = 0.85$

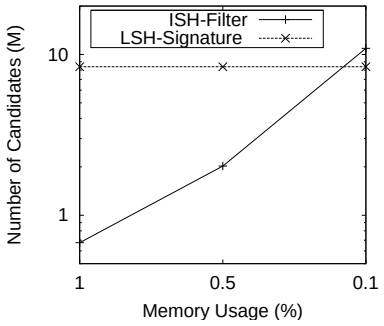

Figure 10: Filtering Power w.r.t. Memory Usage, Weighted Measure, $|\mathcal{R}| = 10M$, $\delta = 0.8$

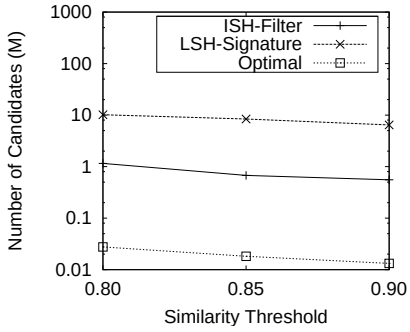

Figure 11: Filtering Power w.r.t. δ, Weighted Measure, $|\mathcal{R}| = 10M$

ISH-Filter is almost one order of magnitude better than the other two alternatives, and *Segmented-Merging* performs similarily to *LSH-Signature*. As we stated earlier, leveraging more tokens is beneficial to prune false members. Although *Segmented-Merging* does not have false positives introduced by hash collision (*i.e.*, it keeps exact *rid*s), it is only able to access the subset of strong tokens. Hence, its pruning power is weaker than *ISH-Filter*, but close to *LSH-Signature*. The false positives of *LSH-Signature* may be introduced by either signature collision or hashing collision in bloom filter. In our experiment, we observe that only a very small portion of the false positives is due to hash collisions. For instance, when $|\mathcal{R}| = 10M$, only 0.002% of total signature hits are introduced by the bloom filter.

Figure 6 and Figure 7 show the number of candidates with respect to candidate length (for $|\mathcal{R}| = 10M$). Both *LSH-Signature* and *ISH-Filter* are more effective in pruning long candidates. The zig-zag patterns are mainly because of the integer rounding of error threshold. For instance, if on un-weighted measure the similarity threshold is 0.8, only 1 token error is allowed for queries to match with a 9-token strings. 2 token errors are allowed for queries to match with 10-token strings. Consequently, we observed that the filtering power for 9-token strings is stronger than the 10-token strings.

If the filter is built solely based on signatures, [3] shows that for string similarity join, the *lsh*-signature performs much better than the prefix-signature. By introducing inverted signature-based hashtable, we show that the filtering power can be significantly improved. We further examine the filtering power by integrating the *lsh*-signature to the *ISH-Filter*. The results are shown in Figure 8, where the dictionary size is $10M$, and the similarity threshold is varied from 0.8 to 0.9. Interestingly, we observe that the performance of the *lsh*-signature based *ISH* (*i.e.*, *ISH-LSH*) is worse than the prefix-signature based *ISH* (*i.e.*, *ISH-Prefix*). The main insight is that we use the weighted extension of prefix-signatures. In the original proposal [3], the prefix signatures are used as binary signatures such that as soon as there is one signature-hit, the candidate passes the filter. While in the weighted extension, it requires multiple signature-hits. It is not clear to us how to extend the *lsh*-signature to the weighted version.

Memory Requirement: The last factor is the memory requirements which are shown in Figure 9. With the default setting, *ISH-Filter* is more compact than the *Segmented-Merging* and roughly uses two times more memory space than *LSH-Signature*. As we discussed in Section 4.3, we can further reduce the memory requirement for *ISH-Filter*. The experiment is presented in Figure 10, where we reduce the memory requirement by progressively removing the largest bit arrays, until the memory space is no larger than rM. We set the reducing rate r to 0.5 and 0.1, and run *ISH-Filter* for dictionary size $10M$ with $\delta = 0.8$. We observe that when $r = 0.5$, where the *ISH-Filter* uses the same memory space as *LSH-Signature* the filtering power of *ISH-Filter* is still more than 4 times better than that of *LSH-Signature*.

Their performance becomes close when $r = 0.1$. At that time, *ISH-Filter* uses 5 times less memory space.

An alternative solution to reduce the memory requirement of *ISH-Filter* is to simply assign less space for each bit-array, and possibly use smaller k (*i.e.*, parameter to control the number of signatures) value. For instance, we can assign $b = 4$ (*i.e.*, number of bits per token), and the memory requirement of *ISH*-Filter is same as $r = 0.5$. The number of candidate members for the same membership checking task is $1.4M$ (versus $2.0M$ by setting $r = 0.5$ in Figure 10).

7.3.2 Filter-Verification

Here we report the experimental results with verification. The *Segmented-Merging* is excluded in this set of experiments since it is not competitive to the other two approaches. We use a *hybrid* verification strategy that verifies candidate members in a batch for each document, using the one-at-a-time verification interface. Given the fact that some members are repeated in the same document, the hybrid verification can reduce the number of disk access by catching previously retrieved dictionary strings.

The verification module for *LSH-Signature* is implemented exactly same as that for *ISH-Filter*. Specifically, at the filter building phase, for each signature s that is generated by a dictionary string r, we store $\langle rid, s, r, wt(r)\rangle$ in a relational table. We do not need to hash the signature again since the *lsh* uses minhashes as signatures. A clustered index on $(s, wt(t))$ is created. At the querying phase, when the bloom filter returns hit for a signature s' (generated by a candidate m), we will retrieve dictionary strings that satisfy $s' = s$ and $\delta \times wt(m) \leq wt(t) \leq \frac{wt(m)}{\delta}$.

Number of True Members: Figure 11 and 12 show the number of true members (represented by *Optimal*) for weighted and unweighted measures, respectively. The similarity threshold is varied from 0.8 to 0.9, and the number of dictionary strings is $10M$. In Figure 12, since we use the unified pruning condition for both jaccard similarity and edit similarity, the number of candidate members (and thus the number of verification call) are exactly same for jaccard similarity and edit similarity.

Execution Time: The overall execution time including verification is shown in Figure 13 (weighted jaccard) and Figure 14 (unweighted jaccard), where we set $\delta = 0.85$ and vary $|\mathcal{R}|$ from $500k$ to $10M$. The curves demonstrate the same trend as those in Figure 4 and Figure 5, validating that the verification costs are proportional to the number of candidate members. When $|\mathcal{R}| = 10M$, *ISH-Filter* uses 408 seconds for weighted measure (versus 121 seconds in filter-only), and 788 seconds for unweighted measure (versus 119 seconds in filter-only). *LSH-Signature* uses 3001 seconds for weighted measure (versus 396 seconds in filter-only), and 6921 seconds for unweighted measure (versus 204 seconds in filter-only).

8. DISCUSSION

Here we discuss two extensions of the proposed methods: (1) leveraging progressive computation for efficient filtering; (2) integrating different tokenization scheme.

Progressive Computation: In membership checking problem, every possible sub-string from the input string is a candidate. In general, we first fix a start position of query sub-strings, and then progressively expand the query sub-string by including more tokens, until the maximal length is reached. Progressive computation refers to the possible computation share among neighboring query sub-strings. Progressive computation has been a main focus of previous proposals [1, 20, 5] in exact-match (or small error) based sub-string lookup. Those methods build an in-memory structure that directly outputs the true members. Since no verification is involved, applying progressive computation can significantly improve the algorithm efficiency. To support flexible similarity thresholds in our problem, we use a filter-verification framework, where the verification cost becomes the main component in the overall computational cost. Therefore, we did not explore the progressive computation with *ISH-Filter* throughout the paper.

For some application where verification is not required, or cheap verification methods are available, we can apply progressive computation for *ISH-Filter* as follows. First, the prefix-signature generation can be made progressive. In fact, it is very likely that the prefix-signatures (or majority of them) keep the same when more tokens are included in the query sub-string. Second, the bit-array lookup for each signature can be made progressive. Suppose the current candidate length is $|m|$, and the number of signatures is $|Sig|$. Without progressive computation, it may involve $|m| \times |Sig|$ bit-lookups. Assume we keep the lookup results of the previous candidate (with length $|m-1|$). When we move $|m|-1$ to $|m|$, we will generate at most one new prefix-signature. Hence, the number of incremental bit-lookups is at most $|m| + |Sig|$ ($|m|$ bit-lookups for the new signature, and $|Sig|$ bit-lookups for the new token).

Tokenization Scheme: We use individual English words as tokens to demonstrate our method. In fact, the tokenization module is orthogonal to the inverted signature-based hashtable structure. For instance, we can use q-gram that combines q English words as a token, and builds a bit-array for each q-gram. In order to find bit-array entries, the algorithm may need to maintain a q-gram table, which could be significantly larger than the table for distinct words. To remedy this, one can hash q-grams to a smaller range and only keep entries for the hash codes. This is equivalent to randomly group multiple q-grams. Another method is to simply create a big bit-array with size M (*e.g.*, the complete memory budget), and insert $\langle q-gram, signature\rangle$ pairs to the bit-array.

9. CONCLUSIONS

In this paper, we considered the problem of identifying all sub-strings in an (long) input string which approximately match (according to one of several popular similarity measures) with some member string in a large dictionary. The characteristic of this scenario is that most input sub-strings do not match with any member of the dictionary. We developed a compact filter which efficiently filters out a large number of sub-strings which cannot match with any dictionary member. The sub-strings which pass our filter are then verified by checking for membership. At the same time, our filter is exact in that any input sub-string which matches with a dictionary member will not be filtered out. We demonstrate using real datasets that our approach significantly outperforms both current best exact methods (often, by an order of magnitude) as well as probabilistic methods,

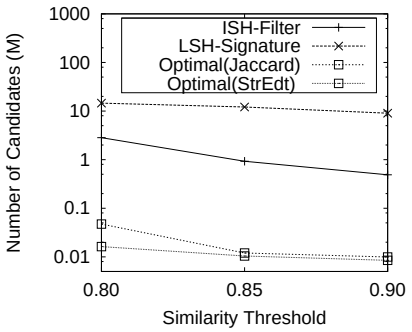

Figure 12: Filtering Power w.r.t. δ, Unweighted Measure, $|\mathcal{R}| = 10M$

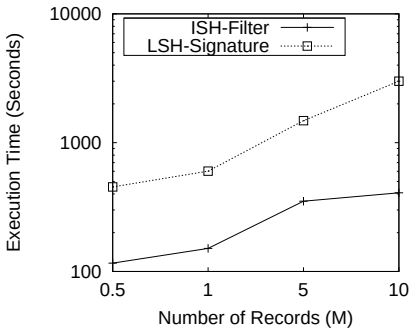

Figure 13: Exeuction Time w.r.t. $|\mathcal{R}|$, Weighted Measure, $\delta = 0.85$

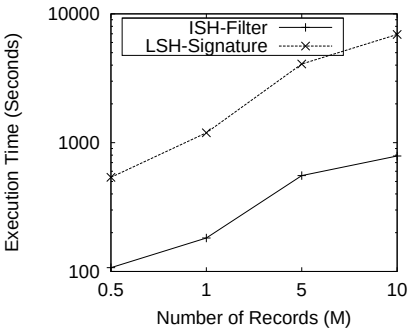

Figure 14: Execution Time w.r.t. $|\mathcal{R}|$, Unweighted Measure, $\delta = 0.85$

which may not identify a small percentage of matching substrings.

10. APPENDIX

Here we discuss how to compute $\eta(k)$: the probability of existing a solid sub-matrix $P(Sig', m')$, such that $|m'| \geq \delta|m|$ and $|Sig'| \geq \tau(m, \delta)$, where $Sig' \subseteq Sig$ and $m' \subseteq m$. For simplicity, let $r = |Sig|$, $c = |m|$ and $p(x, y)$ be the probability of the presence of a solid matrix with exact x rows and y columns. Here each row corresponds to a signature and each column corresponds to a token. The probability for a cell to be set to 1 is $\gamma(k)$ (Section 4.3.1).

For $x = 1$, we have:

$$p(1, y) = \begin{pmatrix} c \\ y \end{pmatrix} \gamma(k)^y (1 - \gamma(k))^{c-y}$$

For $x > 1$, we iteratively compute the probabilities by:

$$p(x, y) = \begin{pmatrix} r \\ x \end{pmatrix} \sum_{y \leq i, j \leq c} p(x-1, i) p(1, j) prob(i, j, c, y)$$

Intuitively, the above equation means that any solid sub-matrix with x rows and y columns comes from the intersection results of a solid sub-matrix A with $x - 1$ rows and i columns ($i \geq y$), and a solid sub-matrix B with 1 row and j columns ($j \geq y$). $prob(i, j, c, y)$ is the probability that the intersection of A and B generates a solid sub-matrix with exact y columns.

$$prob(i, j, c, y) = \begin{cases} 0 & if\ i + j - c > y \\ \dfrac{\begin{pmatrix} j \\ y \end{pmatrix}\begin{pmatrix} c - j \\ i - y \end{pmatrix}}{\begin{pmatrix} c \\ i \end{pmatrix}} & otherwise \end{cases}$$

Note for unweighted case, the number of signatures $\lambda(m, k)$ and the signature threshold $\tau(m, k)$ only depends on $|m|$. For simplicity, we directly rewrite them as $\lambda(|m|, k)$ and $\tau(|m|, k)$, respectively. Given a candidate length $|m|$, we have,

$$\eta(k, |m|) = \sum_{\tau(|m|,k) \leq x \leq \lambda(|m|,k), \delta|m| \leq y \leq |m|} p(x, y)$$

Since $|m|$ is uniformly chosen from $l_E + 1$ (candidates whose length no large than l_E go to exact match module directly) to L (the maximal lookup length). Thus,

$$\eta(k) = (\sum_{l_E < y \leq L} \eta(k, y))$$

11. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] G. S. Brodal and L. Gasieniec. Approximate dictionary queries. In *CPM*, pages 65–74, 1996.

[6] K. Chakrabarti, V. Ganti, J. Han, and D. Xin. Ranking objects based on relationships. In *SIGMOD Conference*, pages 371–382, 2006.

[7] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.

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

[9] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *ICDE*, pages 489–499, 2000.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 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*, pages 491–500, 2001.

[13] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[14] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Inf. Process. Lett.*, 50(4):191–197, 1994.

[15] M. Narayanan and R. M. Karp. Gapped local similarity search with provable guarantees. In *WABI*, pages 74–86, 2004.

[16] R. Ramaswamy, L. Kencl, and G. Iannaccone. Approximate fingerprinting to accelerate pattern matching. In *IMC*, pages 301–306, 2006.

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

[18] A. Singhal. Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–43, 2001.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[20] A. C.-C. Yao and F. F. Yao. Dictionary loop-up with small errors. In *CPM*, pages 387–394, 1995.

[21] X. Zhou, X. Zhang, and X. Hu. Maxmatcher: Biological concept extraction using approximate dictionary lookup. In *PRICAI*, pages 1145–1149, 2006.