Approximate String Membership Checking: A Multiple Filter, Optimization-Based Approach

Chong Sun [1], Jeffrey Naughton [2], Siddharth Barman [3]

Computer Science Department, University of Wisconsin, Madison
1210 W. Dayton St., Madison, WI 53705 USA
[1] sunchong@cs.wisc.edu
[2] naughton@cs.wisc.edu
[3] sid@cs.wisc.edu

***Abstract*— We consider the approximate string membership checking (ASMC) problem of extracting all the strings or substrings in a document that approximately match some string in a given dictionary. To solve this problem, the current state-of-art approach involves first applying an approximate, fast filter, then applying a more expensive exact verification algorithm to the strings that pass the filter. Correspondingly, many string filters have been proposed. We note that different filters are good at eliminating different strings, depending on the characteristics of the strings in both the documents and the dictionary. We suspect that no single filter will dominate all other filters everywhere. Given an ASMC problem instance and a set of string filters, we need to select the optimal filter to maximize the performance. Furthermore, in our experiments we found that in some cases a sequence of filters dominates any of the filters of the sequence in isolation, and that the best set of filters and their ordering depend upon the specific problem instance encountered. Accordingly, we propose that the approximate match problem be viewed as an optimization problem, and evaluate a number of techniques for solving this optimization problem.**

I. 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 a movie 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 the web portal. There has been a great deal of research on how to efficiently conduct both exact string membership checking [2], [3] and approximate string membership checking [4], [5], [6], [7], [8], [9]. In this paper, we focus on approximate membership checking.

A naive approach 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 [9], 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 with the actual dictionary strings. Many filters, including the length filter, counter filter, position filter [8], prefix filter ([7], [9]), LSH filter [10], token distribution filter (TDF) [11], ISH filter [6], and others, have been proposed. These filters use only partial information about the dictionary strings (e.g., the length filter [8] uses string lengths while the prefix filter [7] uses prefixes of string tokens.) Since different filters use different information, different filters are good at eliminating strings with different properties. We suspect that no single filter will dominate everywhere, as the properties of the strings to check affect the effectiveness and efficiency of the string filters. For example, if each document string has far fewer tokens than every dictionary string, then the length filter may be the best filter to use; on the other hand, the length filter may not be efficient if each string to check has nearly the same number of tokens as some dictionary string.

If it is indeed the case that no single filter dominates all other filters for all problem instances, then given an approximate string membership checking problem instance and a set of available string filters built over the dictionary strings, we need to select the optimal filter for this problem instance to maximize the overall membership checking performance. A further observation is that since filters take sets of strings as input and produce sets of strings as output, we can concatenate filters, so that, for example, a string membership checking algorithm could involve applying a filter f_1 to the document strings, then a filter f_2 to the strings that pass f_1, then performing a final, exact check on only the strings that pass both f_1 and f_2. We found in our experiments that in many cases a sequence of the filters dominates any of the filters in isolation, and that the choice of filters and their ordering in a pipeline affects performance. Because the choice of filters and their ordering in a pipeline affects performance, the multiple filter approach to string membership checking is an optimization problem. To the best of our knowledge, this optimization problem has not been studied in the literature; studying the problem is the goal of this paper.

We describe our extended filter-verification framework with an optimizer for approximate string membership checking in Figure I. The extended framework consists of two phases: the

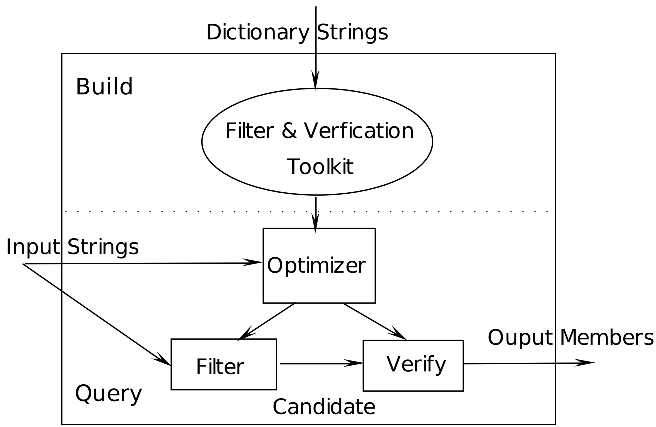


Fig. 1. ASMC Framework with Optimizer

build phase and the query phase. The build phase is offline, in which we build all the potentially useful string filters and verification operators based on the dictionary strings. Each string filter or verification operator is considered as a "tool" in the "toolkit." More tools could be incrementally added into the toolkit as new filters and verification operators are developed.

The query phase is an online phase, in which we conduct dictionary membership checking for the incoming document strings. The optimizer analyzes the characteristics of the document strings, the string filters, and the verification operators. As a result of this analysis, the optimizer determines which string filters to use, and how to order those filters. This optimization problem is similar to the expensive predicate placement problem in relational database optimization [12], [13], and the pipelined filter ordering problem in stream data processing [14], [15]. It is different, however, in that there is a final, exact verification step, so all of the filters are optional.

We summarize our contributions in this paper as follows.

- We propose that the approximate string membership checking problem be viewed as an optimization problem.
- We study this optimization problem, and prove that in general it is NP-hard. We also study special cases of the problem, and develop several approximation algorithms for its solution.
- We conduct experiments with both synthetic and real data to evaluate (a) the premise that no single filter dominates, that multiple filters can dominate single filters, and that the order of filters affects performance, and (b) the efficacy of our proposed algorithms to solve the optimization problem.

In the rest of the paper, we start with a survey of the related work in Section II. In Section III, we discuss the basics of the approximate membership checking problem and the filter-verification framework. We consider the approximate string membership checking problem as an optimization problem of selecting the optimal single filter and building the optimal filter pipeline using multiple filters separately in Section IV and Section V. Finally we conduct a performance study in Section VI and conclude in Section VII.

II. Related Work

Approximate string matching is a classic problem in computer science and many algorithms have been proposed for its solution [16]. 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. In [8], Gravano et al. exploit the existing facilities in commercial databases to support approximate string joins based on tokenizing each string into a set of q-Grams. In [7], 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 [9], 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 [8], use the edit distance similarity. The prefix filter was proposed in [7] and later extended in [9]. A novel signature scheme PartEnum, based on *partitioning* and *enumeration* was presented in [17], in which by controlling the number of string partitions, we guarantee that any two approximately matching strings must be the same in at least one or more partitions. The token distribution filter in [11] compares the token distributions of two strings to determine whether they can approximately match or not. LSH (locality sensitivity hashing) [10], 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 ([6], [18], [19], [20]) 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 [19], efficient exact algorithms are proposed to conduct approximate string checking based on merging token inverted lists. In [6], 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. In [20], the authors check whether a document string can approximately match a dictionary string by merging the inverted lists of tokens for all the tokens in the document string. They focus on progressive computation, discussed in [6], to avoid the redundant computation of merging the same set of inverted lists in checking all the substrings of one document string. Progressive computation is orthogonal to the filtering techniques and we do not consider progressive computation in comparing different filters in our work. Different filters have been used in an ad hoc fashion to accelerate the membership checking, e.g., the length

filter is used in [19], [20]. However, none of the previous work exploits the many existing string filters and uses them systematically. The work in [21], [22] focuses on using cosine similarity metrics based on TF/IDF to approximately match one string against the strings in a large database.

As mentioned previously, we view the approximate membership checking problem as an optimization problem that is closely related to some work in query optimization [23], especially the predicate placement problem [12], [13] studied in relational database query optimization, and the pipelined set cover problem (or the filter ordering problem) studied in stream data processing [14], [15]. The pipelined set cover problem (or the min-sum set cover problem [24]) has been proven to be NP-hard. However, our problem differs from the predicate placement problem and the pipelined set cover problem in that the final verification step means that all filters are optional; the problem is not "apply all these filters in the optimal way," instead, it is "decide which of these filters to apply and how to order them so that the entire membership checking time is minimized."

III. Preliminaries

In this section, we introduce the approximate string membership checking problem and describe the filter-verification approach.

A. Approximate String Membership Checking (ASMC)

Assume we have a string dictionary R. An input string s' is a member of a 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. Many similarity metrics have been proposed, e.g., *edit similarity* and *Jaccard similarity*. Formally, we define the approximate string membership checking (ASMC) problem as in [6]:

Definition 1: Given a string dictionary R, a string similarity function $sim()$ and a similarity threshold τ, find every string or substring s' in a string 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.

B. 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.

C. The String Filters

Most string filters are built based on different string signature schemes, such as using the string length or the prefix [7] 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 [6]) to speed the application of a filter to the strings.

We use two properties to characterize a filter: its filtering cost f^c and its filtering ratio f^r. Suppose we have a filter f. Suppose that we have a set of n_{in} strings as the input to f, and that it takes time t to apply the filter to these strings, and n_{out} strings pass f. Then the filter ratio is $(1 - n_{out}/n_{in})$ and the filtering cost is t/n_{in}. Note that the same filter can have different f^c and f^r on different datasets and different filters can have different f^r and f^c on the same dataset.

IV. Using a Single String Filter

In this section, we consider using a single string filter to solve the ASMC problem. As mentioned previously, many string filters have been proposed, and different filters have been designed based on different string information. As a result, the performance (filtering ratio and filtering cost) of a string filter depends on the characteristics of both the document strings and dictionary strings. In the following, we analyze different properties of three string filters, the length filter, prefix filter and token distribution filter, to motive our work on the optimization problem of selecting the optimal filter for a given ASMC problem instance.

Length Filter [8]: Assume we have two strings $s_i = \langle t_1^{s_i}, t_2^{s_i}, \ldots, t_v^{s_i} \rangle$ and $s_j = \langle t_1^{s_j}, t_2^{s_j}, \ldots, t_m^{s_j} \rangle$ $(m \geq v)$. We represent each string s as a sequence of tokens (e.g., words, phrases or q-Grams) $\langle t_1, t_2, \ldots, t_l \rangle$. Assume we use Jaccard similarity with a matching threshold τ to check strings. String s_i approximately matches s_j only if $|s_i \cap s_j|/|s_i \cup s_j| \geq \tau$, in which $|s_i \cap s_j|$ 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 tokens in s_i and s_j. Correspondingly, we get s_i matches s_j only if $\tau \cdot |s_i| \leq |s_j| \leq |s_i|/\tau$. Checking two strings using the length filter means comparing the string lengths, which is very efficient. However, the length filter can not differentiate strings with similar lengths. For example, two strings $s_1 = \langle t_1, t_2, \ldots, t_{10} \rangle$ and $s_2 = \langle t_{11}, t_{12}, \ldots, t_{20} \rangle$ approximately match based on the length filter.

Prefix Filter [7]: Suppose strings s_i and s_j have l_c common tokens. Based on Jaccard similarity, s_i matches s_j only if $l_c/(m+v-l_c) \geq \tau$. Accordingly, 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. Therefore, presuming we set the string

prefix length to be l_p, s_i approximately matches s_j only if there are at least $(l_c + l_p - v)$ common tokens in the prefixes of s_i and s_j Here, we assume all the tokens in each string follow a universal ordering, and the string prefix covers the first l_p tokens in the string. Checking two strings using the prefix filter means measuring the common tokens of the two strings in the string prefix. Therefore, the prefix filter can not differentiate two strings which have many common tokens in the prefix but few common tokens in other parts of the strings. For example, two strings $s_1 = \langle t_1, t_2, \ldots, t_{10} \rangle$ and $s_3 = \langle t_1, t_2, t_3, t_{11}, t_{12}, \ldots, t_{17} \rangle$ approximately match according to the prefix filter, assuming we set the prefix length to be 3.

Token Distribution Filter [11]: Suppose there is a partitioning scheme ϕ on a string token space that splits all the tokens in the space into u partitions ($P = \langle P_1, P_2, \ldots, P_u \rangle$). We use the number of tokens of a string in each partition as the token distribution of the string. The intuition is that any two approximately matching strings must have similar token distributions. Suppose a token space is split into u partitions ($P = \{P_1, P_2, \ldots, P_u\}$) and a string s_i has $|P_x^{s_i}|$ tokens in partition $P_x \in P$. Any string s_j approximately matches s_i based on Jaccard similarity only if s_j has at least $(\frac{(|s_i|+|s_j|)\tau}{1+\tau} - M^{out})$ tokens and at most $(\frac{(1+\tau)|P_x^{s_i}|+|s_j|-|s_i|\tau}{1+\tau})$ tokens in partition P_x in which $M^{out} = min((|s_i| - |P_x^{s_i}|), |s_j|)$. For example, suppose we have an ordered string token space $P = [t_1, t_2, \ldots, t_{20}]$ split into four partitions: $P_1 = [t_1, \ldots, t_5]$, $P_2 = [t_6, \ldots, t_{10}]$, $P_3 = [t_{11}, \ldots, t_{15}]$ and $P_4 = [t_{16}, \ldots, t_{20}]$. The token distribution of string $s_4 = \langle t_1, t_2, t_{11}, t_{12}, t_{16}, t_{17} \rangle$ is $\langle P_1\{2\}, P_3\{2\}, P_4\{2\} \rangle$, which says s_4 has 2 tokens in partition P_1, 2 tokens in P_3 and 2 tokens in P_4. String $s_5 = \langle t_3, t_4, t_{13}, t_{14}, t_{18}, t_{19} \rangle$ has the same token distribution as s_4, even though many tokens are different in the two strings. In the example for prefix filter, strings s_1 and s_3 have the same prefix but different token distributions.

A. The Single Filter Optimization Problem

We see that different string filters have been designed based on comparing different string information, such as string length, prefix or token distribution. The performance of a string filter depends on the characteristics of the dictionary strings and document strings. Accordingly, we define the optimization problem in the filter-verification approach to the ASMC problem using a single filter as follows.

Given an ASMC problem instance (including a collection of documents and a dictionary of strings), and a set of available string filters and verification operators, the optimization problem of using a single filter is to select the optimal string filter and verification operator such that the overall checking time cost of first applying the filter and then the verification operator is minimized.

Each filter affects the document strings that need to be checked by a verification operator, thus affecting the verification cost. Assume we have n string filters and m verification operators, to search for the optimal filter and verification operator, we explore a space of $n \cdot m$ execution plans.

B. Cost Estimation

To search for the optimal string filter and verification operator, we need to calculate the cost of using each possible plan over all the document strings. Assuming we use a filter f followed by a verification operator v in the plan, the time cost of checking m document strings is $m \cdot f^c + m \cdot (1 - f^r) \cdot v^c$, in which f^c is the filtering ratio and f^r is the filtering cost, and v^c is the verification cost of checking each string passing the filter f. The problem is that we do not know f^c and f^r, and cannot afford to compute them exactly by testing every string with every filter. Accordingly, we need some inexpensive technique to estimate f^c and f^r.

We consider two main broadly used estimation approaches: estimation based on some default values and system statistics ([25], [23]), and estimation based on sampling techniques ([26], [27], [28]). Unlike the case with relational database systems, in string matching problems statistics about the collections of strings and how they interact with filters may not be available. Furthermore, the performance of a filter depends on both the document and dictionary strings. We may have a fixed dictionary, but various documents to check. Accordingly, in this paper, we explore estimation based on sampling techniques [26], specifically, stratified sequential sampling.

The basic idea of sequential sampling-based parameter estimation is as follows. Take estimating the filtering cost f^c of a single filter f, for example. We split the data (document strings) into m disjoint partitions and we divide the m partitions into k disjoint sets (strata). Then we sequentially sample over the strata. At each sampling step (j_{th} step), we randomly and uniformly select one data partition (observation x_i^j) from each (i_{th} stratum) of the k strata, and use x_i^j as the input to the filter to compute the filtering cost $f^c(x_i^j)$. We repeat this sampling n steps until the following stopping conditions satisfy [26]:

$$\tilde{V}_n^c > 0$$

and

$$\epsilon \cdot max(k^{-1} \cdot \sum_{i=1}^{k} \sum_{j=1}^{n} f^c(x_i^j), n \cdot d) \geq z_p \cdot (n \cdot k \cdot \tilde{V}_n^c)^{1/2}$$

where $\tilde{V}_n^c$ is a unbiased and strongly consistent estimator of $\tilde{\sigma}^2 = k^{-1} \sum_{i=1}^{k} \sigma^2$ for the estimated filtering costs over the random observations; d is a parameter to control the steps of the sampling; Z_p refers to $\Phi^{-1}((1 + p)/2)$ where Φ is the cumulative distribution function for a standardized normal random variable and p is the confidence level for the estimation error bounded by $\epsilon \cdot m \cdot \mu_d$ and $\mu_d = max(|D|/m, d)$. We get the estimation of f^c as

$$\tilde{f}^c = \frac{m}{k \cdot n} \sum_{i=1}^{k} \sum_{j=1}^{n} f^c(x_i^j)$$

with the probability p for the error to be within $\pm\epsilon \cdot m \cdot \mu_d$. Achieving small errors (small ϵ) and high confidence (high p) may require a lot of sampling; in some cases it may be

preferable to do less sampling at the expense of larger errors and less confidence. Unless specified otherwise, we use default values with $\epsilon = 0.2$ and $p = 0.95$.

C. Searching for the Optimal Filter

The brute force approach to search for the optimal filter and verification operator considers every combination of each string filter and each verification operator, as the verification cost may depend on the strings passing the filter. For each combination of a string filter and a verification operator, we use the sequential sampling technique to estimate the cost of checking all the document strings. Then we output the string filter and verification operator with lowest estimation cost. Assume we have n string filters and m verification operators, the time complexity of the brute force approach is $O(n \cdot m \cdot s_{cost})$, in which s_{cost} is the sampling cost of estimating the cost of a plan and s_{cost} varies with the parameters ϵ, confidence level p in the sampling and the cost of using the filters. As the major cost in the optimization is the sampling and we may need several samplings in the optimization, we use the number of samplings required to measure the cost of an optimization approach.

An additional complication to the optimization problem is that the cost of applying the verification operator to a string may vary from string to string. In our work we make the simplifying approximation that the cost per string is uniform. In our experiments we found this to be substantially the case for the data we used; if this is not the case, it will introduce another source of error into the optimization process. Whether or not this error is substantial enough to cause optimization errors is an interesting area for future work. We design the algorithm of searching for the optimal filter and verification operator as follows. First, we estimate the cost per string for each verification operator, and keep the verification operator v with lowest estimated cost v^c. Then, we estimate the filtering ratio f^r and filtering cost f^c of each string filter f. We select the filter to minimize $f^c + (1 - f^r) \cdot v^c$. The time complexity of this algorithm is $O(n \cdot s_{cost} + m \cdot s_{cost})$.

V. The Multiple Filter Optimization Problem

As mentioned previously, we adopt the filter-verification framework to handle the ASMC problem. More importantly, since different filters are good at eliminating different document strings, and filters accept strings as input and produce strings as output, we can concatenate a sequence of filters in a pipeline to function as one composite filter. If we regard the final verification phase as a final, *accurate* filter, we can consider the problem to be one of finding the most efficient sequence of filters, ending with an *accurate* filter. Correspondingly, we extend the filter-verification framework for approximate string membership checking by adding an optimizer component. The optimizer selects the optimal sequence of filters, including a verification operator, based on the properties of the input strings and available string filters. Then we use the selected string filters and the verification operator to conduct the membership checking for the input strings. In the following of this section, we describe the optimization problem in detail and how we solve the problem.

A. Overview of Multiple Filter Optimization

The optimization problem presented by our multiple filter approach to the approximate string matching problem can be stated as follows. Given a dictionary of strings and a set of n filters $F = \{f_1, f_2, \ldots, f_n\}$ and a set of m verification operators $V = \{v_1, v_2, \ldots, v_m\}$ built on the dictionary strings, and a similarity function sim and a corresponding matching threshold τ, the optimization problem is to build a pipeline of k filters and a single verification operator as $\check{f}_1, \check{f}_2, \ldots, \check{f}_{k-1}, \check{v}_p$ ($\check{f}_i \in F$ and $\check{v}_p \in V$) to check the document strings with the least time cost.

We note that each filter pipeline should contain one and only one *accurate* filter (or verification operator) and it is only meaningful to have it at the end of a pipeline. By having one *accurate* filter in the pipeline, we guarantee that any document string passing the pipeline is a dictionary member. Suppose we have d candidate strings, and let $\check{f}_i^r$ and $\check{f}_i^c$ be the filtering ratio and cost of filter $\check{f}_i$ in the filter pipeline, $\check{v}_p^c$ be the verification cost. We represent the total cost function of checking d strings using the filter pipeline as

$$\sum_{i=1}^{k-1} \check{f}_i^c \cdot W_i + \check{v}_p^c \cdot W_k$$

Where

$$W_i = \begin{cases} d, & i = 1 \\ d \cdot \prod_{j=2}^{i} (1 - \check{f}_{j-1}^r), & i > 1. \end{cases}$$

To build the optimal filter pipeline, we search a space of $(\sum_{k=1}^{n} k! \cdot \binom{n}{k} \cdot m)$ pipeline plans, in which n is the number of the filters and m is the number of verification operators.

B. Cost Estimation

To search for the optimal pipeline, we need to estimate the time cost of each filter pipeline plan over all the document strings. As in our approach to estimating the cost of single filter plans, we use sequential sampling to estimate the filtering ratio and the filtering cost of each filter in a pipeline.

C. Searching for the Optimal Pipeline

1) Basic Approach: The brute force approach to search for the optimal pipeline is to estimate the time cost of using every possible plan for the document strings and output the plan with the least time cost. The time cost of searching for the optimal pipeline in a space of n filters and m verification operators is $O(\sum_{k=1}^{n} k! \cdot \binom{n}{k} \cdot m \cdot s_{cost})$.

We improve the searching performance by iteratively exploring the plan space using an improved approach as in Algorithm 1. We first build filter pipelines without considering the verification operators and then we select one verification operator for each pipeline. Finally, we output the pipeline including one verification operator with the least estimated cost as the optimal filter pipeline.

Algorithm 1: search for optimal pipeline

```
input : n filters F, m verification operators V and
        the document strings doc
output: filter pipeline P
set P_1={1-filter pipelines};
set k=2;
foreach k < n do
    set G_k={All k-filter pipelines};
    foreach g_k ∈ G_k do
        g_{k-1}= g_k - last filter;
        /* remove the last filter */
        if g_{k-1} ∉ P_{k-1} then remove g_k from G_k;
        else c(g_k) = estimate_cost(g_k, doc );
        /* estimate the cost of g_k by
           sampling doc */
    end
    foreach g_k ∈ G_k do
        if c(g_k)=min{q_k | q_k ∈ G_k&q_k contains the
        same set of filters as g_k} then
            add g_k into P_k
        end
    end
end
set cost = ∞;
foreach pipeline P' ∈ ⋃_{k=1}^{n} P_k do
    foreach verification operator v ∈ V do
        c = estimate_cost(P' + v, doc);
        if c < cost then cost = c and P = P' + v;
    end
end
return P;
```

To build the pipeline with the set of n filters, in the first pass of Algorithm 1, we estimate the cost of using every 1-filter pipeline and keep them in a set P_1. A subsequent pass, say pass k, consists of two phases. First, we generate all the candidate optimal k-filter pipelines with the estimated optimal $(k-1)$-filter pipelines based on the property that the first $(k-1)$-filter pipeline in each optimal k-filter pipeline is also optimal. The correctness is guaranteed by the fact that the filter ordering in a pipeline with a fixed set of filters does not affect the strings passing the pipeline. Therefore, the sub-pipeline of the first $(k-1)$ filters in each optimal k-filter pipeline must be an optimal $(k-1)$-filter pipeline for the set of $k-1$ filters. Next, we estimate the time cost of each candidate k-filter pipeline and keep the pipeline with the least cost for each different set of k filters in P_k.

Then, we select verification operators to add to the pipelines. For each pipeline, we estimate the cost of each pipeline plus every verification operator. Finally we output the pipeline plan (including a verification operator) having the least estimated cost as the optimal filter pipeline plan for the n filters and m verification operators.

To build the filter pipeline, in the first pass, we estimate the costs of $\binom{n}{1} \cdot (n-1)$ 2-filter pipelines. In the k_{th} pass, we estimate the costs of $\binom{n}{k} \cdot (n-k)$ $(k+1)$-filter pipelines. Therefore, the total number of samplings is $O(\sum_{k=1}^{n-1}(n-k) \cdot \binom{n}{k})$. To add the verification operators, we estimate the cost of $O(\sum_{k=1}^{n}\binom{n}{k} \cdot m)$ pipelines. The total time cost of the algorithm is $O((\sum_{k=1}^{n}\binom{n}{k} \cdot m + \sum_{k=1}^{n-1}(n-k) \cdot \binom{n}{k})) \cdot s_{cost})$.

Even though the improved approach is more efficient in searching for the optimal filter pipeline than the brute force approach, it is still an exponential algorithm. Unfortunately, it is unlikely we can do better with an optimal algorithm, for as we prove next, that the problem of searching for an optimal filter pipeline is NP-hard. To prove that this general optimal plan searching problem is NP-hard, we first consider a "simpler" version of the searching problem, in which we assume that any filter spends the same time on checking each string. We call this assumption as the *uniform cost assumption*.

Theorem 1: Suppose we have a document of strings and n candidate filters and m verification operators. If the uniform cost assumption holds for the filters and verification operators, it is NP-hard to determine the optimal filter pipeline.

Proof: In the appendix we show that the so called Filter Coverage problem is NP-hard. Filter Coverage is a formalization of the problem at hand and hence we get the desired result. ■

As we have shown that the "simpler" problem of searching for the optimal filter pipeline with the uniform cost assumption in NP-hard, the general searching problem is also NP-hard.

2) Approximation Algorithms: To solve the optimization problem efficiently, we design an approximation algorithm in Algorithm 2. Algorithm 2 consists of two stages: the building stage and the refining stage. During the building stage, Algorithm 2 takes iterations to build the filter pipeline incrementally based on a greedy selection strategy. In each iteration, we estimate the filtering ratio and the filtering cost of each filter or verification operator after applying the filters in the pipeline. Then we select the filter or verification operation with the largest ratio of the filtering ratio to the filtering cost to add into the filter pipeline. We continue to add filters into the pipeline until a verification operator is selected. In the refining stage, we start from the last filter in the pipeline and estimate if we can get better performance by removing the filter from the pipeline. If we can improve the estimated cost of the pipeline, then we remove it. We continue the checking and removing until we can not improve the estimated performance.

During the building stage, in the worst case, we need $(n+1)$ iterations to construct the filter pipeline by using all the filters and one verification operator. In iteration 0, we make $(n+m)$ samplings to estimate the cost of using each filter and verification operator to check the document strings. During iteration k, we estimate the cost of using the pipeline with each of the $n-k$ left filters and m verification operators. In this iteration, we conduct $n-k+m$ samplings. Accordingly, the total number of samplings in the building stage is $(n^2+n)/2+(n+1)\cdot m$. In the refining stage, we can make at most $n-1$ sampling estimations. Therefore, the time complexity of Algorithm 2 is $O((n^2+n\cdot m)\cdot s_{cost})$.

Algorithm 2: search for optimal pipeline

```
input  : n filters F, m verification operator V and
         document strings doc
output: a filter pipeline P

set P=ϕ;
while P contains no verification operator do
    set cost = −∞;
    set op =null
    foreach o ∈ (F ∪ V) and o ∉ P do
        c(o), r(o) = estimate_{cost,ratio}(o, doc);
        /* estimate the filtering cost c(o)
           and ratio r(o) of o after
           applying P over doc                 */
        if r(o)/c(o) > cost then
            set cost = r(o)/c(o) and op = o;
        end
    end
    if op ∈ V then
        for (i = 1; i < P.size; i++) do
            P' = P− last filter;
            c = estimate_{cost}(P + op, doc);
            c' = estimate_{cost}(P' + op, doc);
            if c' < c then set P = P';
            else add op to P and return P;
        end
    else
        add op to P;
    end
end
```

Algorithm 3: search for optimal pipeline

```
input  : n filters F, m verification operator V and
         document strings doc
output: a filter pipeline P

Set P=ϕ;
Set L=ϕ;
foreach o ∈ (F ∪ V) do
    if o ∈ F then
    c(o), r(o) = estimate_{cost,ratio}(o, doc);
    else c(o) = estimate_{cost}(o, doc);
    add o to L;
end
sort L based on the decreasing order of θ(o) Where
```

$$\theta(o) = \begin{cases} r(o)/c(o), & o \in F \\ 1/c(o), & o \in V. \end{cases}$$

```
while L ≠ null do
    select o ∈ L with the largest θ(o);
    if o ∈ F then remove o from L ;
    add o to P ;
    else add o to P and return P;
end
```

When the uniform cost assumption holds for all the filters and verification operators, we have the following theorem on the optimality of the filter pipeline generated by the greedy selection used in Algorithm 2.

Theorem 2: Assume the uniform cost assumption holds for all the filters and verification operators. Also assume that after each filter application the number of dictionary strings in the document is no more than the number of non-dictionary strings, then using the greedy selection in Algorithm 2 achieves a 10 approximation for the optimal filter pipeline problem.

Proof: Under the above assumptions the algorithm given in the appendix is same as Algorithm 2 and hence we can apply Theorem 5 (see Appendix). Hence we get the desired approximation guarantee. ■

The assumption that the number of dictionary strings in the document is less than the number of non-dictionary strings is in fact true in most practical scenarios.

We note the approximate algorithm to solve the optimization problem still takes a quadratic number of sampling estimations, which could be expensive for an optimization step of the string membership checking problem. Correspondingly, we present another approximate algorithm in Algorithm 3.

The motivation of Algorithm 3 is that a good pipeline should have filters with high filtering ratio and low filtering cost in the front. To be specific, in Algorithm 3, we first estimate the filtering ratio and filtering cost of each flter independently. Then we incrementally add string filters and verification operators into the pipeline according to the decreasing order of θ, where θ is the ratio of the filtering ratio to the filtering cost of a filter and θ is the ratio of 1 to the filtering cost of a verification operator. We stop when we add one verification operator into the pipeline.

In Algorithm 3, we need $(n+m)$ samplings to estimate the filtering ratio and filtering cost of each filter and verification operator. The cost of ordering the filters and verification operators based on the sampling is $O((n+m)log^{(n+m)})$. Then we select at most $(n+1)$ filters and verification operators to build the pipeline. Therefore the total cost of the algorithm is $O((n+m) \cdot s_{cost} + (n+m) \cdot log^{(n+m)})$.

Suppose the uniform cost assumption holds for all the filters and the string data distribution before applying a filter f is the same as that after apply the filter f. In other words, any filter in a pipeline would not affect the filtering ratio and the filtering cost of any other filter appearing later in the pipeline. We say the filters are independent of each other. In this case, we can accurately estimate the filtering ratio and the filtering cost of each filter independently, and we can incrementally add filters into the pipeline and we stop when we meet a verification operator.

We have Theorem 3 to show that Algorithm 3 find the optimal filter pipeline when the filters are independent and the uniform cost assumption holds for the verification operators.

Theorem 3: Assume we have a set F of n independent fil-

ters and a set V of m verification operators. The optimal time cost of checking a set of document strings via a filter pipeline built of the filters and verification operators is achieved by putting the filters in decreasing order of $\theta(o)$, where

$$\theta(o) = \begin{cases} f^r(o)/f^c(o), & o \in F \\ 1/f^c(o), & o \in V. \end{cases}$$

Proof: Observe that interchanging the order of multiple filters can be decomposed into interchanging the filter order pairwise. Without loss of generality, assume we have the optimal filter pipeline P built using the n filters and m verification operators. Assume that filter o_i appears right before filter o_j in P. If $f^r(o_i)/f^c(o_i) < f^r(o_j)/f^c(o_j)$, then we can interchange the order of o_i and o_j and decrease the time cost of using the filter pipeline P. This contradicts that P is the optimal filter pipeline. Therefore, all the filters in the optimal filter pipeline must be ordered in the decreasing ratio of the filtering ratio to the filtering cost. Similarly, we can show that in the optimal filter pipeline any filter o_i appearing before a verification operator o_j, we must have $f^r(o_i)/f^c(o_i) > 1/f^c(o_j)$; otherwise, $f^r(o_i)/f^c(o_i) \leq 1/f^c(o_j)$. ■

As we have proven, on the problem, Algorithm 2 is an approximation algorithm with a provable bound, and for other conditions, Algorithm 3 is optimal. When the conditions do not hold, both of these algorithms become heuristics, and we evaluate their performance in the next section.

VI. Performance Study

In this section we conducted experiments to explore the optimization approach to the approximate string membership problem. We first explore the performance of various filters and filter pipelines (to provide evidence of the need for an optimization-based approach), and then explore the performance of our estimation and optimization algorithms. We do this both with synthetic and "real" data sets.

A. Experiment Setup

We ran all the experiments on a 2.4 GHZ Intel Duo Core PC with 3GB memory. All the strings filters and the membership checking were implemented in Java. We use the time cost to measure the ASMC performance. The cost of ASMC may consist of three components: optimization cost (only if we use an optimization based approach), filtering cost (only if string filters are used) and verification cost.

1) Data Sets: To investigate the performance of the various filters and pipelines of filters, we used both both synthetic and real data sets in the experiments.

Synthetic Data: We designed a data generator consisting of a dictionary generator and a document generator, which can be customized using two types of parameters: shared parameters, which specify correlations between the generated dictionary and documents; and exclusive parameters, which 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	

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. The Token Distribution refers to the token frequency distribution in the document and dictionary strings. 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 (10K); Token Distribution (*normal*); Number of Strings (1M); Average, Min and Max Length (10, 5, 15). This configuration generates a dictionary of 1M strings composed of 10K distinct tokens and the frequencies of all the tokens in the dictionary follow a standard *normal* distribution. The average, minimum and maximum lengths of the dictionary strings are 10, 5 and 15. Similarly, we specify the default values for the document generator as: Token Space Coverage (10K); Token Distribution (*normal*); Number of Strings (10K); Average, Min and Max Length (20, 15, 25). Furthermore, we set the Overall Token Space to contain 10K tokens, the Token Space Overlap to be 0.5 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 [29] as the dictionary. The DBLife data consists of 10K web pages and the total data size is approximately 48MB. 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 it contains about 86K distinct tokens.

2) Filters and Filter Pipeline: We implemented four individual string filters and one verification operator, and each is assigned a ID for ease of notation as listed below. We also implemented code to "glue" them together into pipelines of filters.

Length Filter (1): We implemented the length filter according to [8]. With the length filter, a string extracted from the documents needs to be verified only if the length of the string is in a range determined by the dictionary strings.

Compressed Inverted List Filter (2): We built inverted lists for the dictionary string tokens and we used a fixed n-bit array for each token to store the hash values of the string ids, rather than using the id list. By adjusting the bit array size n, we can trade filtering cost for accuracy. That is, a larger n leads to a higher cost and higher accuracy, while a smaller n leads to a lower cost and less accuracy.

TDF Filter (3): We implemented the token distribution filter in [11]. We set the default token partitioning number b to be $|D|/100$, where $|D|$ is the number of tokens in the Overall

Token Space. We adjust n according to the memory usage, so that the *TDF* consumes a similar amount of memory to other filters in the experiments.

ISH Filter (4): We implemented the ISH filter [6] using prefix signatures. Similar to the inverted lists, ISH stores a list of string signatures for each token rather than string ids. We use k (number of tokens in string prefix) and n (bit-array size for each token) to control the performance and memory usage. We experimented with different values for k and n and picked k=3 and n=1024 since those numbers worked well in our experiments.

Inverted List Filter (v): This is the only verification operator we considered. We implemented the DivideSkip algorithm [19] to merge the inverted lists of tokens for candidate strings.

Filter Pipeline: We cascade different filters to form a filter pipeline used in the multiple filter approach, e.g., a pipeline plan "3,4,v" refers to applying the TDF filter and the ISH filter in order before the verification operator.

B. Experimental Results

1) Using a Single String Filter: First we explore the performance of the individual filters, to ascertain whether or not they indeed show different performance. The results are shown in Figure 2(a). That graph has a lot of information on it. The three groups of bars correspond to varying numbers of document strings. Each group contains a bar for each of the filters (plus the verification operator; recall that these filters are only approximate, so they cannot be used without being followed by the verification operator.) For comparison, we also include the performance of only the verification operator indicated by "v". Finally, the bar labeled "opt" corresponds to our optimization procedure described in Section IV-A. The results demonstrate that selecting the correct filter is important, and that our optimization approach was effective, finding the optimal filter with only about 20% overhead.

Figure 2(b) shows the same information for varying the token space overlap. The point made by Figure 2(b) is that the relative performance of the filters changes with different data sets; the relative effectivess of the filters is different between Figure 2(a) and Figure 2(b), and also different in the different groups in Figure 2(b). Also note that our optimization method is close to the optimal in all cases. This lends credence to our belief that no one filter dominates everywhere, and that our optimization-based approach is an effective way to choose filters. Figure 2(c) makes the same point by varying the dictionary token frequency, the average number of each dictionary token.

In addition to the synthetic data sets, we also evaluated the ASMC performance using single filters on real data sets with the crawled web pages in DBLife as the documents and the paper titles extracted from the DBLP Bibliography as the dictionary in Figure 2(d) and 2(e). The results in Figure 2(d) and 2(e) over our real data set confirm the conclusions we got over the synthetic data sets.

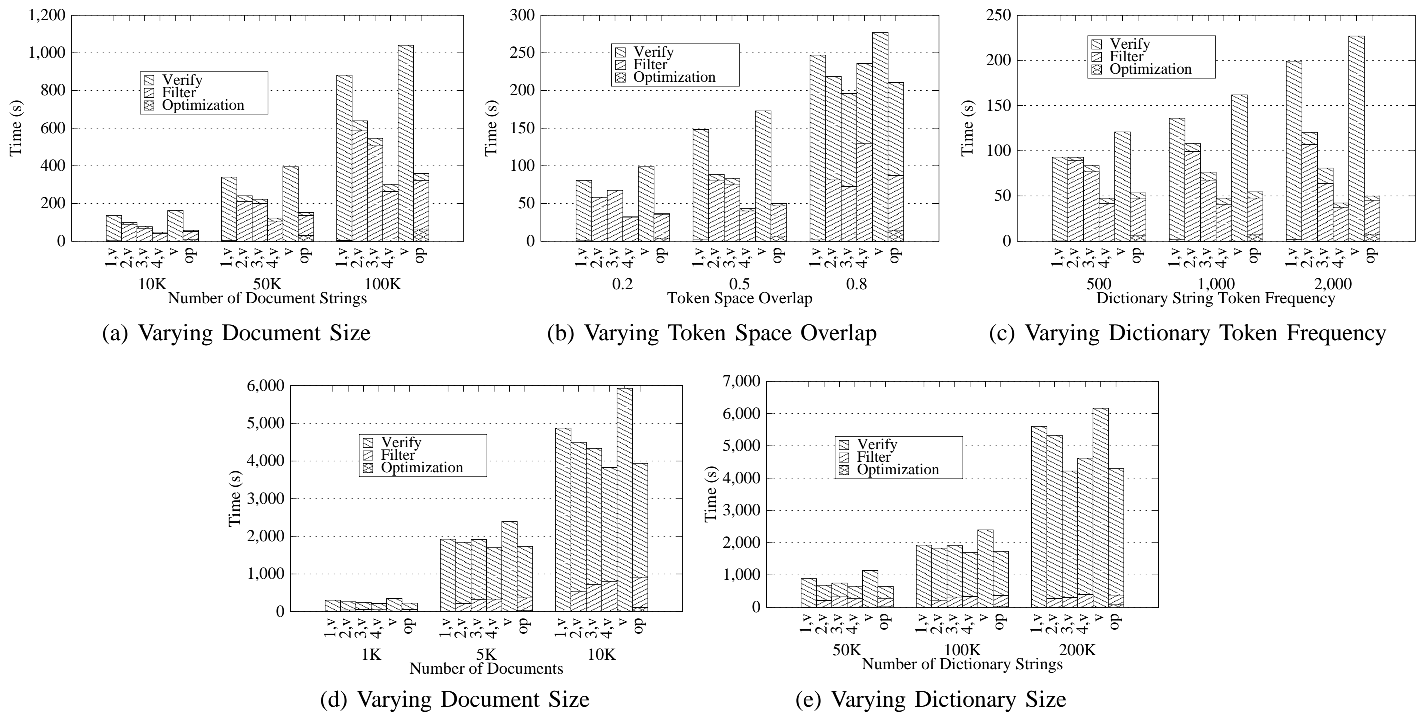
To summarize, using different string filters in the ASMC problem may result in different performance. There might be no single filter which is also superior to others in all scenarios. Our optimization approach can find the optimal filter for various problem instances we have studied on both synthetic and real data sets. More importantly, the optimization overhead is relatively small.

2) Using Multiple Filters: In Figure 3 we move to consider the case where we use multiple filters in a pipeline (rather than a single filter plus a verification operator.) We wish to explore (a) whether a pipeline of filters can indeed perform better than a single filter, and (b) whether the order of filters in a pipeline affects performance, and (c) whether the best order for a set of filters is constant across different problem instances, and finally (d) how well our various optimization algorithms perform in this environment.

In Figure 3(a),we explore different pipelines on different document sizes. We see that on these data sets, adding filters improves performance, and that using all of the filters is the fastest. Figure 3(b) considers different orders of filters. It shows that the performance is indeed sensitive to order.

These two points suggest that some optimization-based approach might be effective. Hence, in Figure 3(c) and 3(d), we evaluate the performance of the optimization approach in searching for the optimal filter pipeline. The basic approach (Algorithm 1) to our optimization problem is guaranteed to find the optimal filter pipeline, when the sampling based estimation is accurate. However, it requires an exponential number of sampling estimations. For example, even with 4 filters, we need to make more than 30 estimations and each estimation may take 1%-2% of the overall cost. Making 30 estimations may take longer than just running a reasonable plan. Therefore, we did not consider the basic approach in our performance study. We only evaluated the performance of the two approximation algorithms (Algorithm 2 and Algorithm 3) to solve the optimization problem, which are separately represented as "Optimization_A1" and "Optimization_A2" in Figure 3(c) and 3(d). For comparison, we also show the best performance of the pipelines with 1, 2, or 4 filters before verification and the worst performance of the 4-filter pipeline. In Figure 3(c), we vary the number of document strings from 10K to 100K. We find that both two approximation algorithms can find the optimal filter pipeline. However, the optimization overhead of "Optimization_A1" is large, i.e., more than 30% of the overall cost. Due to the large overhead, "Optimization_A1" only achieves similar performance to using the best 2-filter pipeline. Still its performance is much better than that of the worst 4-filter pipeline. The approach "Optimization_A2" not only finds the optimal filter pipeline, but also only add a small amount of optimization cost, i.e., about 10% of the overall cost. We get similar results when we vary the token space overlap from 0.2 to 0.8 in Figure 3(d). Both approximation algorithms in the optimization approach can find the optimal filter pipeline for each problem instance, and "Optimization_A2" requires little cost in searching for the optimal filter pipeline than "Optimization_A1".

Besides the synthetic data sets, we evaluated the effectiveness of the optimization approach on real data sets with the



(a) Varying Document Size (b) Varying Token Space Overlap (c) Varying Dictionary Token Frequency

(d) Varying Document Size (e) Varying Dictionary Size

Fig. 2. ASMC Performance Using Single Filters

crawled web pages in DBLife as the documents and the paper titles extracted from the DBLP Bibliography as the dictionary. In Figure 3(e), we show the best performance of using 1, 2 and 4 filters before the verification and also the worst performance of using 4 filters to compare with performance we achieved by the optimization approach using the two approximation algorithms. We vary the number of documents from 1K to 10K, while keeping the dictionary size at 100K. We get similar results on real data sets as in Figure 3(c). The two approximation algorithms indeed find the optimal filter pipeline for each case. "Optimization_A2" achieves the second best performance over all the approaches. The overhead of the optimization is relatively small compared with the overall time cost.

To summarize, our experiments show that not only the set of filters but also the ordering of the filters in the pipeline is critical to the ASMC performance. The two approximation algorithms we proposed to solve the optimization problem work well in many problem instances we have studied on both the synthetic and real data sets. In particular, in our experiments Algorithm 3 found the optimal pipeline with acceptable overhead.

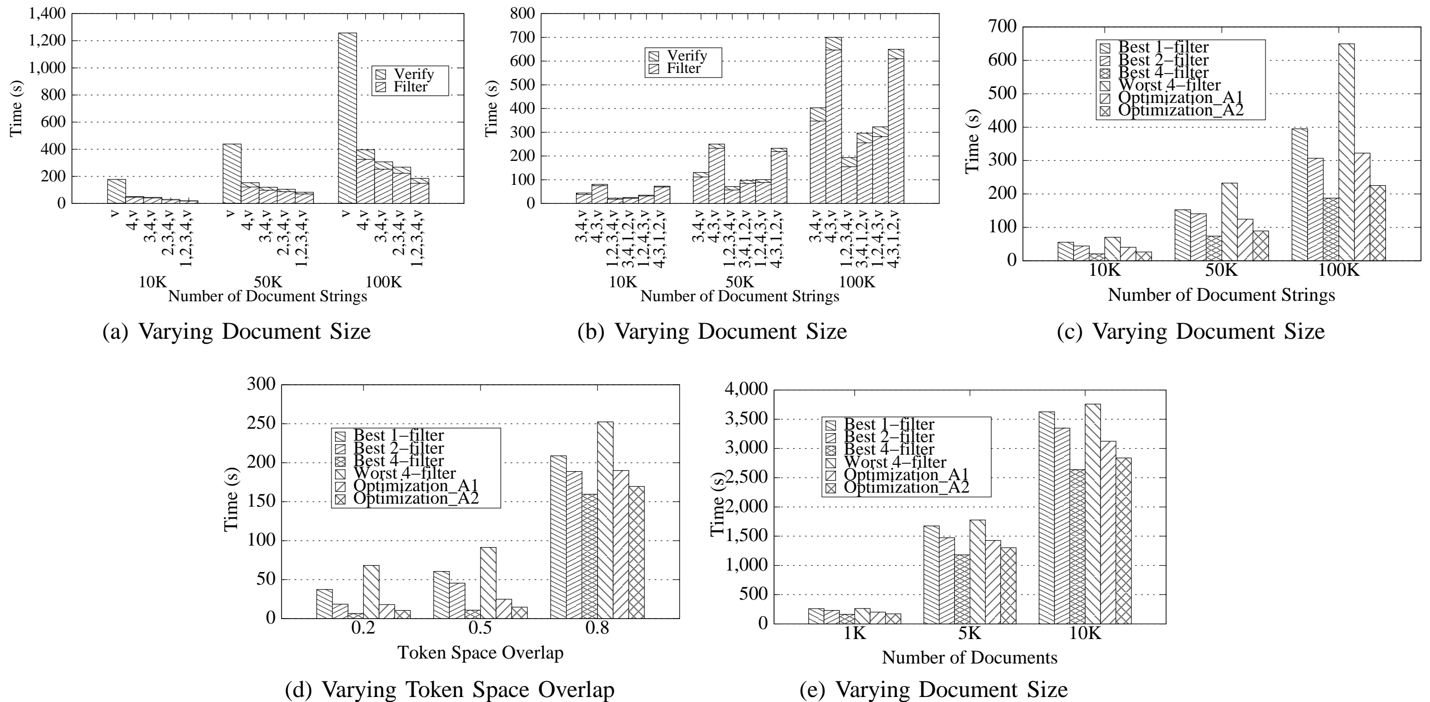
VII. Conclusion

The filter-verify approach to the ASMC problem relies upon a powerful but simple observation: it is often possible to find a relatively simple characterization of a set of strings D that can be used to efficiently determine that a candidate string s cannot possibly be an approximate match for any string in D. A wide variety of such characterizations are possible; each leads to a different filter. Furthermore, since different characterizations capitalize on different aspects of the strings, different filters are effective for different instances of the string membership problem.

The multi-filter framework we advocate in this paper exploits this observation to arrive at what in some sense amounts to a "toolkit" approach: when faced with an instance of the string membership problem, one looks into one's "tookit" of filters and tries to assemble a pipeline of these filters that will work well on the specific problem at hand. Of course, choosing the appropriate filters and assembling them in the proper order is in general a non-trivial problem (NP-hard); to do so automatically is an optimization problem that requires some mechanism for estimating the cost and effectiveness of possibly many different potential filtering pipelines.

In this paper we have explored and evaluated this new multi-filter, optimization-based approach. Considering different estimation cost in the optimization approach, we designed two different approximation algorithms to solve the optimization problem. We have found through experiments with synthetic and real data sets that our optimization approach over a "toolkit" of filters performs better overall than any of the filters individually or even any statically chosen combination of filters. We regard this as promising evidence that the multi-filter, optimization-based approach has merit.

A great deal of room for future work remains. Certainly the set of filters we considered do not exhaust the space of all possible filters; characterizing what constitutes a "complete" set of filters for the toolkit and possibly devising new filters to achieve this completeness is an interesting and challenging task. Also, while our optimization techniques were effective in our experiments, it is possible that more dynamic approaches



(a) Varying Document Size (b) Varying Document Size (c) Varying Document Size

(d) Varying Token Space Overlap (e) Varying Document Size

Fig. 3. ASMC Performance Using Multiple Filters

that "route" candidate strings through networks of filters may outperform single pipelines of filters in interesting cases. Investigating such approaches is also an interesting direction for exploration.

REFERENCES

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

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

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

[4] A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh, "Text indexing and dictionary matching with one error," in *J. Algorithms*, 2000.

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

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

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

[8] 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.

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

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

[11] C. Sun and J. Naughton, "The token distribution filter for approximate string membership checking," in *WebDB*, 2011.

[12] J. M. Hellerstein, "Practical predicate placement," in *SIGMOD Rec.*, 1994.

[13] J. M. Hellerstein and M. Stonebraker, "Predicate migration: optimizing queries with expensive predicates," in *SIGMOD Rec.*, 1993.

[14] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive ordering of pipelined stream filters," in *SIGMOD*, 2004.

[15] K. Munagala, S. Babu, R. Motwani, and J. Widom, "The Pipelined Set Cover Problem," in *ICDT*, 2005.

[16] G. Navarro, "A guided tour to approximate string matching," in *ACM Computing Surveys*, 2001.

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

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

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

[20] G. Li, D. Deng, and J. Feng, "Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction," in *SIGMOD*, 2011.

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

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

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *SIGMOD*, 1979.

[24] U. Feige, L. Lovász, and P. Tetali, "Approximating min-sum set cover," 2003.

[25] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *SIGMOD Rec.*, 1996.

[26] P. J. Haas and A. N. Swami, "Sequential sampling procedures for query size estimation," in *SIGMOD*, 1992.

[27] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja, "Statistical estimators for relational algebra expressions," in *PODS*, 1988.

[28] R. J. Lipton, J. F. Naughton, and D. A. Schneider, "Practical selectivity estimation through adaptive sampling," in *SIGMOD Rec.*, 1990.

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

APPENDIX

For a formal treatment, we abstract the optimization problem under the uniform cost assumption as minimum cost filter coverage problem which is described as follows. In the filter coverage problem we are given a set U along with a subset $D \subset U$, which we call the dictionary. We also have a set of filters $\mathcal{F}$ and verifiers $\mathcal{V}$. Each $f \in \mathcal{F}$ has an associated set $S_f \subset U \setminus D$ and a processing cost (per unit element) $c_f \in \mathbb{R}_+$. Also each verifier $v \in \mathcal{V}$ has an associated processing cost $c_v \in \mathbb{R}_+$ and the set associated with v is all of $U \setminus D$. For

ease of notation, at places, we will denote the set and the cost associated with f as $S(f)$ and $c(f)$ respectively. Write the *filtering cost* of applying a set of k filters, $F = \{f_i\}_i$, in order σ as $c(F, \sigma) := \sum_{i=1}^{k} c(f_{\sigma(i)}) \left|\left(U - \cup_{j=1}^{i-1} S(f_{\sigma(j)})\right)\right|$. Also the filtering cost of applying F in order σ and verifier v, denoted as $c(F, \sigma, v)$, is defined to be $c(F, \sigma) + c_v \times \left|U - \cup_{i=1}^{k} S_{f_i}\right|$.

The goal of the minimum cost filter coverage problem is to select a subset of filters, $F \subset \mathcal{F}$, and an ordering σ over F along with a verifier $v \in \mathcal{V}$ such that the filtering cost $c(F, \sigma, v)$ is minimized. Overall an instance of the filter coverage problem is specified as $(U, D, \mathcal{F}, \mathcal{V})$. Next we show that the problem is in fact NP-Hard.

Theorem 4: The Filter Coverage problem is NP-Hard.

Proof: We will reduce the Pipelined Set Cover problem [15] to Filter Coverage. An instance of the Pipelined Set Cover problem consists of a set of elements U' and collection of subsets of U', $A' = \{S'_1, \ldots, S'_k\}$. For all $i \in [k]$ a processing cost c'_i is associated with set S'_i. In the decision version of the problem the objective is to determine whether there exists an ordering, π, over the sets such that the pipelined cost is no more than T. Here the pipelined cost is defined to be $\sum_{i=1}^{k} c'_{\pi(i)} \left|U' - \cup_{j=1}^{i-1} S'_{\pi(j)}\right|$. Given an instance of the Pipelined Set Cover problem (U', A') we construct an instance, $(U, D, \mathcal{F}, \mathcal{V})$, of the Filter Coverage problem as follows. Set $U = U'$ and $D = \phi$ and the set of filters is constructed from A', in particular for each set S'_i we construct a filter f_i by setting $S(f_i) = S_i$ and $c(f_i) = c'_i$. We construct a single versifier v with $c_v = T + 1$, by definition we have $S(v) = U \backslash D = U$. In the decision version we enquire whether there exists a solution (F, σ, v) with filtering cost no more than T.

Next we show that a solution, (F, σ, v), of filtering cost T exists iff there is an ordering π of pipelined cost T. In the forward direction, say there exits a solution (F, σ, v) of cost no more than T. Note that we must have $\cup_{f \in F} S(f) = U$. Otherwise, $|U - \cup_{f \in F} S(f)| \geq 1$ and the processing cost of the verifier by itself would be at least $T + 1$, contradicting the fact that the overall filtering cost is no more than T. This gives us a pipelined solution, π, follows: initially set π to be the sets corresponding to the filters in F ordered as in σ and then augment it with the remaining sets of filters in $\mathcal{F} \setminus F$, in any order. We have $\cup_{f \in F} S(f) = U$ and the filtering cost is no more than T. The sets after $\cup_{f \in F} S(f)$ have to cover no element hence there processing cost in π is zero, overall this implies that the pipelined cost of π is no more than T. To prove the other direction, given π with cost no more than T then we consider $(\mathcal{F}, \pi, v)$. In particular, $\cup_{f \in \mathcal{F}} = U$, hence the verifier processes no element. Since the pipelined cost is no more than T we have $c(\mathcal{F}, \pi) \leq T$, which in turn implies that the filtering cost of $(\mathcal{F}, \pi, v)$ is no more than T hence we have the desired claim. ■

Next we present a greedy approximation algorithm for the Filter Coverage problem and show that it achieves an approximation factor of 10. Given an instance of the Filter Coverage Problem $(U, D, \mathcal{F}, \mathcal{V})$, say $|D| = n$ and $|U \backslash D| = m$. Note that the dictionary D is disjoint from the sets associated with the given filters and verifiers and hence will be "processed" by all of them. We essentially apply a set cover like greedy algorithm till the number of uncovered elements not in the dictionary is at least n, after that we pick a verifier with minimum processing cost c_v. In particular, say at step i we have M_i uncovered elements which are not in the dictionary (initially $M_0 = m$), if $M_i \geq n$ we pick the filter (or verifier) which minimizes the cost ratio c_j/m_j where m_j is the number of uncovered elements in the set associated with the filter and c_j is the processing cost of the filter. On the other hand if $M_i \leq n$ we select a verifier v with the smallest c_v value. We have the following theorem stating the approximation factor achieved by the algorithm.

Theorem 5: The above algorithm achieves an approximation factor of 10 for the Filter Coverage problem.

Proof: Let (F, σ, v) be the solution generated by the algorithm. Say $|F| = k$ and without loss of generality assume that the filters are selected by the algorithm in order that is f_1 through to f_k. Note that the set being processed at step i has cardinality $M_i + n$ and hence we get $c(F, \sigma) = \sum_{i=1}^{k} c_i(M_i + n)$, where c_i is the processing cost of filter f_i. Moreover whenever we select a filter we have $M_i \geq n$ and hence $c(F, \sigma) \leq 2 \sum_{i=1}^{k} c_i M_i$. Now consider the Pipelined Set Cover [15] instance $(U \setminus D, A')$, where A' is the collection of sets associated with filters and verifiers. Say the optimal pipelined cost of this instance is O^*_p.

Note that an optimal solution, (F^*, σ^*, v^*), of the Filter Coverage problem can be transformed into a solution for the Pipelined Cover problem. We simply extend σ^* by augmenting sets of filters and verifiers from $(\mathcal{F} \cup \mathcal{V}) \setminus (F^* + v^*)$ in any order. Also $s(v^*) = U \setminus D$, hence the processing cost of the augmented sets is zero. We note that the pipelined cost of (F^*, σ^*, v^*) is no more than the filtering cost of (F^*, σ^*, v^*). The filtering cost includes processing all of U whereas the pipelined cost includes processing only $U \setminus D$. Write $O^*_f = c(F^*, \sigma^*, v^*)$, we have $O^*_f \geq O^*_p$.

The relevant observation is that F, the initial set of filters selected by the algorithm, would be the same if the greedy 4-approximation algorithm for the Pipelined Set Cover [15] was applied to the instance $(U \setminus D, A')$. The processing cost of $U \setminus D$ under F is $\sum_{i=1}^{k} c_i M_i$, since this is part of the greedy 4-approximate solution for the Pipelined Set Cover problem we have $\sum_{i=1}^{k} c_i M_i \leq 4O^*_p$, therefore $\sum_{i=1}^{k} c_i M_i \leq 4O^*_f$. As stated before $c(F, \sigma) \leq 2 \sum_{i=1}^{k} c_i M_i$ and hence $c(F, \sigma) \leq 8O^*_f$. Finally we account for the cost incurred by the verifier. Say it is applied after k filters, at that time we have $M_{k+1} \leq n$. Hence its processing cost, $c_v(M_{k+1}+n)$, is no more than $2c_v n$. By selection we have $c_v \leq c_{v^*}$. The dictionary D is processed by the verifier v^* in the optimal solution, hence $O^*_f \geq c_{v^*} n$. Overall we get that $c_v(M_{k+1} + n) \leq 2O^*_f$. Since the filtering cost of the solution generated by the algorithm is equal to $c(F, \sigma)$ plus the processing cost of the verifier we get that the algorithm achieves a cost no more than 10 O^*_f implying an approximation factor of 10. ■