Efficient and Tunable Similar Set Retrieval

Aristides Gionis[*]
Stanford University
gionis@cs.stanford.edu

Dimitrios Gunopulos[†]
University of California Riverside
dg@cs.ucr.edu

Nick Koudas
AT&T Laboratories
koudas@research.att.com

Abstract

Set value attributes are a concise and natural way to model complex data sets. Modern Object Relational systems support set value attributes and allow various query capabilities on them. In this paper we initiate a formal study of indexing techniques for set value attributes based on similarity, for suitably defined notions of similarity between sets. Such techniques are necessary in modern applications such as recommendations through collaborative filtering and automated advertising. Our techniques are probabilistic and approximate in nature. As a design principle we create structures that make use of well known and widely used data structuring techniques, as a means to ease integration with existing infrastructure.

We show how the problem of indexing a collection of sets based on similarity can be reduced to the problem of indexing suitably encoded (in a way that preserves similarity) binary vectors in Hamming space thus, reducing the problem to one of similarity query processing in Hamming space. Then, we introduce and analyze two data structure primitives that we use in cooperation to perform similarity query processing in a Hamming space. We show how the resulting indexing technique can be optimized for properties of interest by formulating constraint optimization problems based on the space one is willing to devote for indexing. Finally we present experimental results from a prototype implementation of our techniques using real life datasets exploring the accuracy and efficiency of our overall approach as well as the quality of our solutions to problems related to the optimization of the indexing scheme.

1 Introduction

Object Relational systems [SM96] allow storage and query capabilities on complex data types. The products of many years of research on spatial, multimedia and time series data as well as the indexing techniques and query capabilities defined on them, are finding their way on most commercial Object Relational engines. Indexing in Object Relational databases has been an active area of research for many years. Various indexing techniques tailored to specific data types [GG98] exist, as well as some proposals for unifying popular indexing schemes [HNP95]. Through relational extensions (extenders, cartridges, datablades) users can store and query complex data as well as use novel query capabilities on them, like for example querying based on similarity.

```
create type user_t (
name varchar(20),
address varchar(100),
zipcode char(5),
country varchar(10),
books_bought
setof(varchar(200)));
create table user of type user_t;
```

Figure 1: Example Schema

Set value attributes are a concise and natural way to model complex data sets. Modern Object Relational systems [SM96] support set value attributes and allow various query capabilities on them. Set value attributes are part of SQL3, a fact that is likely to make ubiquitous their use in commercial applications. Although the expressive power of query languages allowing set value attributes has long been studied in various communities, little is known or has been reported to date on the alternate options for storage, indexing and query execution on set value attributes. New applications emerging from the wide spread use of the World Wide Web as front end to database engines, such as *collaborative filtering* and *automated recommendations*, create pressing needs for advanced query processing on sets. Consider for example the simple schema of Figure 1. It keeps track of the books each user bought in a set form. For a specific user u, we would like to identify the users which bought books most 'similar' to u, for suitably defined notions of similarity between sets. This is a common query for recommendation purposes. Analy-

[*] This work was performed while the author was visiting AT&T Labs

[†] Part of this work was performed while the author was visiting AT&T Labs. Supported by NSF CAREER Award 9984729, NSF IIS-9907477, and the DoD.

```
            select name, books_bought
            from user
            where
Similar(u.books_bought,books_bought) > 0.9
```

Figure 2: Sample Query

sis of the result set can then take place with specific algorithms to make recommendations to u for specific books. Such capability is most desirable in popular e-commerce sites and online stores. Similar examples can be formulated for various scenarios of interest. For example such functionality is desirable for the analysis of web site logs based on the IP address of individual connections, for dynamic content generation or dynamic advertising. Assuming the existence of function *Similar* that assesses the similarity of two sets as a fractional number between 0 and 1 (0 if the sets are disjoint and 1 if they are identical), the query in Figure 2 can be formulated. The query retrieves all users and the associated set of books they bought, if they are highly similar to the set of books bought by u. Now assume that a set of books on a specific subject go on sale. The e-tailer might wish to email potentially interested users for the sale. In this case however, the users which have already bought a large fraction of the books on sale, will not be good candidates, since they own most of the books already. One should request users that have already bought a small fraction of the books on sale (say between 40% and 70%), as they might be interested to obtain more related books on the same topic. In this case, one should form the query using a range of smaller similarity values. Support for similarity queries on sets, enables various forms of sophisticated analysis. For example one may retrieve and correlate users with highly *dissimilar* buying patterns (with similarity say less than 0.1) to reason about buying behavior based on other attributes of interest, such as geographical location. Taking this example further, if a "profile set" of books is created for a user class, one may retrieve user classes with similar or dissimilar profiles and analyze user classes further, seeking features or rules responsible for the similar or dissimilar buying behavior.

In their most general form, set similarity queries specify a range of similarity values of interest. The user specifies a lower and upper bound of similarity values of interest and retrieves all sets with similarity within the range of interest. The ability to retrieve sets based on similarity, can serve as a primitive for effective similarity based query processing on sets. It can serve as a basis for the development of efficient set mining algorithms such as clustering algorithms for sets, classification algorithms based on set similarity as well as join algorithms. For example, treating web pages as sets of words they contain, a clustering operation based on set similarity could identify clusters of web pages which are similar but not copies of each other. Such an operation can be very useful towards more efficient (and accurate) implementation of the 'what's related' feature of popular web browsers.

In this paper, we initiate a formal study of indexing schemes capable of supporting similarity queries on sets. As a design principle, we require that the indexing techniques developed rely on data structures readily available in modern Object Relational systems, so that integration with existing infrastructure is not troublesome. Our overall approach is based on a randomized technique capable of capturing similarity between two sets, for suitably defined notion of similarity. We then use the outcome of this randomized procedure to develop *hash based schemes* capable of retrieving sets, similar to a query set. The resulting indexing scheme is approximate and its accuracy depends on a well known tradeoff with space. We analyze this tradeoff and in particular we show how such a hash based scheme can be optimized both for performance and accuracy given a constraint on the space used for indexing. Moreover, we analyze the overall implications of various parameters in the performance of our technique and we formulate index construction as an optimization problem, proposing algorithms for its efficient solution.

This paper is organized as follows: In Section 2 we provide definitions necessary for the bulk of the paper. Section 3 presents our approach for preprocessing a set collection. In Section 4 we introduce hash based data structure primitives that we use in cooperation to construct an index for similarity queries on sets. Section 5 presents algorithms that optimize the resulting indexing scheme for objectives of interest. Section 6 presents experimental results from a prototype implementation of our algorithms using real data sets, analyzing the performance and accuracy of our indexing scheme. In Section 7 we review work related to the work presented herein and finally Section 8 concludes the paper pointing to problems of interest for further study.

2 Definitions

Given a database of sets, the focus of this paper is to develop efficient algorithms for answering queries of the form:

> *"Return the sets in the database that are similar more than 90% with a query set q"*
>
> *"Return the sets in the database that are similar between 80% and 90% with a query set q"*
>
> *"Return the sets in the database that are less than 65% similar with a query set q"*

All of the above query types can be answered by providing queries based on a range of similarities as a primitive to the user. Whenever one deals with similarity queries, one has to define precisely measures that capture the notion of similarity under consideration.

Definition 1 (Similarity Measure) *Given two sets A and B the similarity between them is defined as:*

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

which always assumes values between 0 and 1.

This measure is known in the literature as the *Jaccard Coefficient* of two sets. It expresses the *fraction* of elements common to both sets. Notice that this measure is not a *metric* [1]. Nevertheless, a distance function can be defined in terms of the similarity as $d(A, B) = 1 - sim(A, B)$, and it is easy to show that such a distance function is indeed a metric.

Formally, the problem we address is as follows:

Definition 2 (Set Similarity Range Query) *Given a collection of N sets $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$, preprocess $\mathcal{S}$ so that for any query (q, $[\sigma_1, \sigma_2]$) return* efficiently *all sets $s \in \mathcal{S}$ such that $\sigma_1 \leq sim(s, q) \leq \sigma_2$, where q is a query set and $[\sigma_1, \sigma_2]$ is the target similarity range.*

An obvious attempt to process set similarity queries is to transform sets to binary vectors forming their *indicator vectors*. In other words, assuming that all sets contain elements of an a-priori known universe $U = \{e_1, e_2, \ldots, e_{|U|}\}$, a set S is mapped to the vector $< b_1 b_2 \ldots b_{|U|} >$ where the bit b_i is 1 iff the element e_i belongs to the set S. Unfortunately, such a mapping is inappropriate for our purposes. The dimension of such a vector space is typically huge and even worse, it is typical that not all the elements of the universe are known in advance (e.g. documents represented as sets of the words they contain). We don't assume knowledge of the universe the set elements are derived from, or of the cardinality of sets in our collection. For the types of applications we are interested in, such assumptions are restrictive as the domain of set elements might not be static or known in advance and the size of sets in our collection can be arbitrary.

3 Indexing Similar Sets

In this section we present our proposal for indexing arbitrary sets based on similarity. We will present our approach in the following steps:

- We will first present an embedding of the set collection $\mathcal{S} = \{S_1 \ldots S_N\}$ into a space of vectors $\mathcal{V}$ of fixed dimensionality. This embedding is based on the previously introduced theory of min-wise independent permutations [Coh97, BCFM98]. The embedding essentially derives a reduced dimensionality version of the problem capturing similarity between sets in S. The embedding is probabilistic but has no distortion in expectation. In particular the embedding has the property that for any two elements $S_i, S_j \in \mathcal{S}$ their similarity can be assessed by manipulating the coordinates of their corresponding vectors V_i, V_j in $\mathcal{V}$.

- We introduce an additional embedding of $\mathcal{V}$ into the Hamming space with no distortion. The new embedding makes use of results from coding theory to preserve the nice properties of space $\mathcal{V}$.

- Finally, we will show how the resulting Hamming space can be indexed by introducing hash based schemes which we will optimize for similarity querying.

3.1 Min-wise Independent Permutations (Embedding $\mathcal{S}$ to $\mathcal{V}$)

The first ingredient of our embedding is the Min Hashing technique which was first introduced by Broder et. al., [BGMZ97] (see also [Coh97]). It has been used to identifying mirror web pages and also for estimating the selectivity of boolean as well as twig queries [CKKM00, CJK+01]. Since our proposal uses this technique in its first step, we review Min Hashing here for completeness.

The basic idea of Min Hashing is to implicitly define a random order on the elements universe using hashing. Such a random order is viewed as a random permutation $\pi(\cdot)$. Let $\pi(\cdot)$ be chosen at random over the set of random permutations of the elements universe. For a set A define $\min\{\pi(A)\} = \min\{\pi(x) | x \in A\}$. Then for two sets A and B:

$$Pr(\min\{\pi(A)\} = \min\{\pi(B)\}) = sim(A, B).$$

Note that the *same* permutation is used for both sets A and B. The permutation can be typically approximated using hashing and the technique can be applied independent of the type of set elements (numerical,categorical). The values $\min\{\pi(A)\}$ are represented in practice using a number of fixed precision. Min-wise permutations provide a way to perform the embedding of $\mathcal{S}$ to $\mathcal{V}$. By repeating the Min Hashing process k times on each set S_i of $\mathcal{S}$ we can represent it by the k resulting *min-hash values* and form a vector V_i of $\mathcal{V}$ referred to as the *min-hash signature* of S_i. Given sets S_1 and S_2 let V_1 and V_2 be their min-hash signatures. If the sets have similarity s (for some $0 \leq s \leq 1$), the *expected* number of min-hash values that the two min-hash signatures agree would be $s \cdot k$, and as it was shown in Cohen et al [Coh97] using Chernoff bounds, that the number of equal min-hash values between the min-hash signatures is an *unbiased estimator* of the expectation above.

3.2 Embedding into a Hamming Space (Embedding $\mathcal{V}$ to $\mathcal{H}$)

Let $\mathcal{H}$ be a Hamming space; each element of $\mathcal{H}$ is a binary vector. We use the notation H^ℓ to denote that binary vectors in $\mathcal{H}$ have dimensionality ℓ. We will show how to embed space $\mathcal{V}$ into a Hamming space $\mathcal{H}$ of some fixed dimensionality in a way that preserves the notion of similarity between vectors in $\mathcal{V}$. We will subsequently show how to index the resulting Hamming space.

Definition 3 (Hamming Distance) *Let h_1, h_2 be vectors in a Hamming space. The Hamming Distance $d_H(h_1, h_2)$ is defined as the number of bits in which the two vectors differ.*

Although the notion of Hamming distance is typically used in the literature, we find more convenient for

[1] Recall that metric is a function $m(\cdot, \cdot)$ which is non-negative, symmetric, $m(x, y) = 0$ iff $x = y$, and satisfies the triangle inequality

the description of our algorithms to work with the equivalent concept of Hamming Similarity.

Definition 4 (Similarity in Hamming Space) *Let h_1, h_2 be vectors in a Hamming space of dimension ℓ. The Hamming similarity $S_H(h_1, h_2)$ of h_1,h_2 is defined as the fraction of common bits between the two vectors, thus $S_H(h_1, h_2) = 1 - \frac{d_H(h_1,h_2)}{\ell}$.*

Essentially we wish to construct an embedding of $\mathcal{V}$ to a Hamming space having the following property:

Objective 1 *If vectors $V_1, V_2 \in \mathcal{V}$ have $sim(V_1, V_2) = s$, then for the corresponding vectors h_1, h_2 in the Hamming space, $S_H(h_1, h_2) = s$*

It is easy to see that a straightforward mapping into a Hamming space does not meet out objective. Let $V = (v_1, \dots v_k) \in \mathcal{V}$. Each of the v_i min-hash values is an integer of b bits. A straightforward embedding of v into Hamming space is feasible by transforming v into a binary vector:

$$u(V) = binary(v_1)binary(v_2)\dots binary(v_k) \qquad (1)$$

where $binary(v_i)$ is the binary representation of the integer v_i. This is essentially an embedding of vectors in $\mathcal{V}$ into a Hamming space H^{bk}. Consider two vectors V_1, V_2 of $\mathcal{V}$ with similarity s. Under mapping $u(.)$ they agree in $b \cdot s \cdot k$ bits. Nothing however can be said about the remaining $b \cdot (1-s) \cdot k$ bits obtained from the remaining $(1-s) \cdot k$ min-hash values that vectors V_1, V_2 do not agree upon.

Example 1 *Consider $V_1 = (7, 3, 5, 1), V_2 = (3, 3, 5, 3)$. Then $sim(V_1, V_2) = 0.5$. Also $u(V_1) = 111011101001$ and $u(V_2) = 011011101011$. However, the fraction of bits common between $u(V_1), u(V_2)$ is 0.83* ■

We construct our embedding using error correcting codes (ECC). The theory of error correcting codes is very rich and the reader is referred to [MS93] for a thorough exposure. Here we state the results that we essentially need for constructing our embedding. ECC are functions that map an b-bit string v to a binary codeword $C(v)$ of length m (the length of the code), for some m larger than n.

Definition 5 (Distance of an ECC) *The* distance *of an ECC is d if any two codewords $C(u_1)$,$C(u_2)$ differ in at least d places.*

Assume the existence of a code, with the property that each pair of *distinct* b-bit strings u, v is mapped to codewords $C(u)$ and $C(v)$ of length m bits, such that the distance between $C(u)$ and $C(v)$ is *exactly* $\frac{m}{2}$. Let $ecc(.)$ be a code with this property and $ecc(v_i)$ the binary representation of a codeword corresponding to the binary representation of some min-hash value v_i. We complete out embedding by applying the following transformation to each $V \in \mathcal{V}$

$$h(V) = ecc(v_1)ecc(v_2)\dots ecc(v_k) \qquad (2)$$

$\mathcal{V}$ is embedded in this way into the Hamming space $\mathcal{H}^{mk}$. For this space, we state the following theorem:

Theorem 1 *Let $V \in \mathcal{V}$ and $ecc(.)$ a code with the property that each pair of distinct codewords of length m is at distance exactly $\frac{m}{2}$. Then, the mapping $h(V) = ecc(v_1)ecc(v_2)\dots ecc(v_k)$ is an embedding of the space $\mathcal{V}$ into a $D = (mk)$-dimensional Hamming space with the following property: for any two vectors V_1 and V_2 with similarity $sim(V_1, V_2) = s$, the Hamming distance of the corresponding vectors $h(V_1)$ and $h(V_2)$ is $d_H(h(V_1), h(V_2)) = D - (s + \frac{1-s}{2})D = \frac{1-s}{2}D$.*

It remains to show, how one can construct code $ecc(.)$. A code with the property of $ecc(.)$ can be constructed easily by using *simplex codes* [MS93]. Their generation involves a simple matrix multiplication with a standard matrix. Such codes are fairly standard and further details are available elsewhere [MS93]

3.3 Range Similarity Queries in $\mathcal{H}^{mk}$

Our approach for performing similarity range queries in $\mathcal{H}^{mk}$ is as follows:

> **Preprocessing:** Initially transform all sets in $\mathcal{S}$ by embedding the space to $\mathcal{V}$ and then into $\mathcal{H}^{mk}$. Then preprocess the binary vectors in $\mathcal{H}^{mk}$ by building indices such that similarity queries (in the Hamming space) can be answered efficiently.
>
> **Query Processing:** Given a query set q and a target similarity range $[\sigma_1, \sigma_2]$, first transform the set q into a D-dimensional binary vector q_b applying the same transformations used in the preprocessing phase. Then, using the indices built in the preprocessing phase find all vectors in $\mathcal{H}^{mk}$ that have distance d with the vector query q_b, where $d_1 = \frac{1-\sigma_2}{2}D \leq d \leq d_2 = \frac{1-\sigma_1}{2}D$, and return the sets that correspond to these vectors.

Every similarity query $(q, [\sigma_1, \sigma_2])$ on $\mathcal{S}$ can now be transformed into a similarity query in $\mathcal{H}^{mk}$ using Theorem 1. Thus, we are faced with the following problem:

Problem 1 (Indexing Space $\mathcal{H}^{mk}$) *Given a collection of N binary vectors in $\mathcal{H}^{mk}$ preprocess all vectors so as for any query (q_b, $[d_1, d_2]$) return* efficiently *all vectors $p \in \mathcal{H}^{mk}$ such that $d_1 \leq d_H(p, q_b) \leq d_2$, where $d_H(p, q_b)$ is the Hamming distance between p and q_b. As before, q_b is the embedded (into Hamming space) query vector and $[d_1, d_2]$ is the target distance range in Hamming space.*

Thus our problem has been reduced to the problem of performing range queries in a Hamming space of potentially very large dimensionality. In the next section we describe our approach for constructing hash based indexing structures capable of processing such queries in Hamming space. Our overall approach for executing such queries is not exact but approximate.

4 Filter Indices

We first present two hash based data structures that our algorithm uses as primitives to efficiently execute range queries into Hamming space. We choose to use Hamming similarity in what follows for convenience; the description of the data structures can be restated easily using Hamming distance making use of definition 4. We also assume that each element of $\mathcal{H}$ is augmented with a *set identifier* (sid), denoting the identifier of the set in space $\mathcal{S}$ the binary vector corresponds to.

The first data structure primitive we present, *Similarity Filter Index* $SFI(s^*)$ is a hash based index structure that identifies with very high probability, given a Hamming similarity threshold s^* and a binary query vector q_b, the set identifiers of all vectors in $\mathcal{H}^{mk}$ that have Hamming similarity at least s^* with q_b. The second, *Dissimilarity Filter Index* $DFI(s^*)$, identifies with very high probability the set identifiers of all vectors in $\mathcal{H}^{mk}$ that have Hamming similarity at most s^* with q_b. We will subsequently use these structures in cooperation to construct our indexing scheme.

4.1 Similarity Filter Index

In this section we show how the Similarity Filter Index, with parameter s^* can be constructed, which essentially operates as a probabilistic filter on $\mathcal{H}^{mk}$ to locate with very high probability vectors similar more than s^* with a query vector q_b. For each vector $h \in \mathcal{H}^{mk}$ we select r out of D bits selecting the bit positions at random, generating a reduced dimensionality representation of h, denoting it as $h^{'}$. Let $\mathcal{H}^{'}$ the space resulting after applying this sampling step to each element of $\mathcal{H}^{mk}$. We assume that space $\mathcal{H}^{'}$ consists of tuples $(h^{'}, sid)$, where *sid* is the set identifier of the corresponding set in $\mathcal{S}$.

An observation is that if two elements of $\mathcal{H}^{mk}$ have high Hamming similarity then with high probability the corresponding elements in $\mathcal{H}^{'}$ will be equal. If we hash the sid's based on the value of $h^{'}$, with very high probability sids corresponding to similar sets will hash in the same bucket. To amplify the probability that similar vectors will have equal $h^{'}$ values, we repeat the process l times. Assuming that two vectors v, u have Hamming similarity $S_H(v, u) = s$ the probability that u and v will have the same $h^{'}$ values at least once (out the l repetitions) is:

$$Pr[v \text{ and } u \text{ have equal } h^{'} \text{ values at least once}] = \quad (3)$$
$$1 - (1 - s^r)^l \quad (4)$$

Notice that this probability depends only on the parameters r, l and the Hamming similarity s of the vectors v and u, and not on the vectors themselves. Let this probability be $p_{r,l}(s)$. One can observe that for fixed values of r and l, $p_{r,l}(s)$ is an 'S'-shaped function for $0 \leq s \leq 1$, that approximates the *unit step* function at some turning point s. What it means is that the probability that two $h^{'}$ values are equal, and thus the probability that the two corresponding sid's will hash in the same bucket is close to 1, if the vectors have Hamming similarity greater than s, while it is close to 0 if the vectors have Hamming similarity smaller than s. Function $p_{r,l}(s)$ acts as a probabilistic *filter function.* For any $s^* \in [0, 1]$ we can pick the parameters r and l, so that $p_{r,l}(s)$ has s^* as the turning point. This can be done by setting $p_{r,l}(s^*) = 1/2$. This gives one equation and two unknowns, r and l. All pairs of the parameters r and l that satisfy the equation define the *family* of filter functions $p_{r,l}(s)$ that have s^* as turning point. The relationship between r and l in this equation is "monotonic", i.e. as l increases r should also increase and as a result $p_{r,l}(s)$ becomes steeper. This property introduces a tradeoff between the accuracy of the probabilistic filter function $p_{r,l}(s)$ and the value of l. This tradeoff is explored in section 5.

Our approach for constructing $SFI(s^*)$ consists of first choosing the value of r given a specific value of l; then we repeat the following process l times: We sample r bits from each element of $\mathcal{H}^{mk}$ creating $\mathcal{H}^{'}$. We then hash each element of $\mathcal{H}^{'}$ creating a hash table. Let sid_{count} be the number of sid's that can be accommodated in a bucket. Choosing $O(\frac{N}{sid_{count}})$ hash buckets is sufficient to guarantee that no bucket overflows occur during this hashing process. Thus, at the end of the l repetitions we have created l *hash tables.* To retrieve all the set identifiers which are more than s^* similar to a query vector $q_b \in \mathcal{H}^{mk}$, we repeat the following process l times: At the i-th repetition we sample randomly r bits from q_b and determine bucket j that the resulting value hashes, in the i-th hash table. Let B^i_j be that bucket. The set of sid's that form the query answer is $SimVector(s^*, q_b) = \bigcup_{i=1}^{l} B^i_j$. The entire process is illustrated in Figure 3(a). Structure $SFI(s^*)$ requires $O(N)$ space and answers a query with $O(l)$ bucket accesses.

The *ideal filter function* would be a *unit step* function positioned at a similarity value s^*. For such a function the sids of two vectors which are more than s^* similar would hash in the same bucket with probability 1. Function $p_{r,l}(s^*)$ is not a unit step function and it incurs a certain number of *false positives* and *false negatives.* For a specific set collection $\mathcal{S}$, let $\mathcal{D}_{\mathcal{S}}$(s) be its *similarity distribution function*, that is a function that for every value of $s, 0 \leq s \leq 1$ expresses the number of sets which are s similar. Assuming that all queries are equally likely, the expected number of false positives for a random query is:

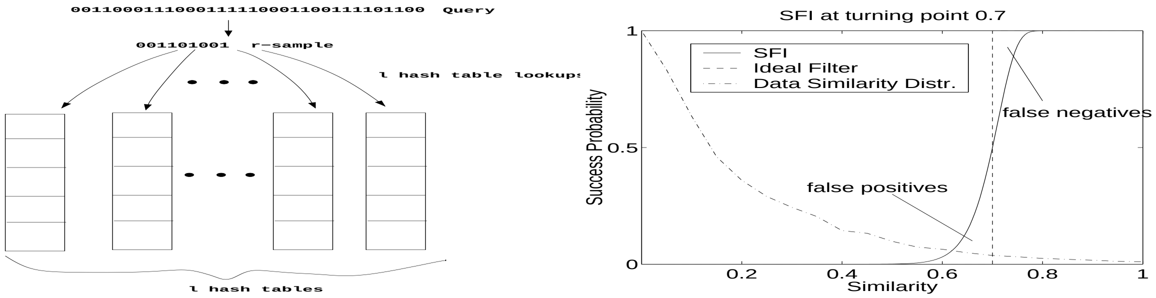
Definition 6 *The expected number of false positives incurred by function $p_{r,l}(s^*)$ is:*

$$P_{p_{r,l}}(s^*) = \int_{s<s^*} \mathcal{D}(s) \cdot p_{r,l}(s) \cdot ds \quad (5)$$

This number quantifies the expected number of set identifiers that are erroneously identified having similarity above s^* with a random query, and is shown graphically in figure 3(b). Similarly:

Definition 7 *The expected number of false negatives incurred by function $p_{r,l}(s^*)$ is:*

$$N_{p_{r,l}}(s^*) = \int_{s>s^*} \mathcal{D}(s) \cdot (1 - p_{r,l}(s)) \cdot ds \quad (6)$$



(a) Hash Based Implementation of SFI (b) Probabilistic Filter Implemented by SFI

Figure 3: Similarity Filter Index and its implementation using hashing

This number quantifies the expected number of set identifiers that are not identified as having similarity above s^* with a random query and is shown graphically in figure 3(b). Both quantities degrade the quality of $p_{r,l}(s)$ as a probabilistic filter and should be minimized.

A first attempt to use structure $SFI(s)$ as a primitive to design an indexing scheme is the following. Split the entire similarity range $[0,1]$ into $k+1$ intervals, by selecting k points $0 = t_0 < t_1 < \ldots < t_k < t_{k+1} = 1$. For each of these k points we create instances of the data structure $SFI(t_1), \ldots, SFI(t_k)$. Now given a query ($q_b$, $[\sigma_1, \sigma_2]$) we find the appropriate points $lo = t_i$, $up = t_j$ that minimally enclose the range $[\sigma_1, \sigma_2]$, and return the set $(SimVector(lo, q_b) \setminus SimVector(up, q_b))$ as the answer to the query (q_b, $[\sigma_1, \sigma_2]$). (Here by $A \setminus B$ we denote the set difference operation.)

The problem with this proposal is that it becomes very inefficient for small values in the similarity range $[0,1]$. To see that, notice that the algorithm performs a difference operation between two sets A and B, where the second set is always a subset of the first. In such a case the *overhead* of the operation is the size of the set B, because it contains elements that we have to remove from the final answer. In contrast, the elements that belong in A and not in B are all the elements of the difference and we *should* report in the final answer. Translating this intuition in our setup, given a query range $[\sigma_1, \sigma_2]$ the overhead of the difference operation is the size of the set $SimVector(up, q_b)$. Obviously this becomes large as the value of s_2 is far away from 1. In those cases the algorithm will have very large and unnecessary overhead.

In the next section we show how to overcome this problem by creating data structures that return all vectors that are *dissimilar* to a query vector q_b.

4.2 Dissimilarity Filter Index

The key observation that permits to avoid the overhead of the set difference operation, is to notice that for a query range $[\sigma_1, \sigma_2]$, the condition that σ_2 is far away from 1 forces σ_1 to be close to 0. Thus, in such cases, it is better to subtract the vectors that have similarity *less* than σ_1 from the vectors that have similarity *less* than σ_2. In turn, this means that we should provide a way to answer queries regarding *dissimilar* vectors. More specifically, given a query vector q_b one should be able to identify the sid's of all vectors $h \in \mathcal{H}^{mk}$ such that $s_H(h, q_b) \leq s$. The next theorem provides the basis for doing so:

Theorem 2 *If we* reverse *all bits of a query vector* q_b, *forming vector* $\bar{q_b}$ *then for every vector* $h \in \mathcal{H}^{mk}$,

$$S_H(h, \bar{q_b}) \geq 1 - s \text{ iff } S_H(h, q_b) \leq s \qquad (7)$$

that is, if a vector is at most s-similar with the query vector q_b, it is at least (1-s)-similar with vector $\bar{q_b}$. This implies that we can solve the problem of finding dissimilar vectors by constructing again a probabilistic filter function. In particular, to build a data structure that retrieves with high probability all sid's at most s^*-similar to a query q_b, we build a data structure capable of retrieving all sid's at least $1 - s^*$-similar to q. This is structure $SFI(1 - s^*)$. Then we reverse (complement) q_b constructing $\bar{q_b}$ and we use the data structure $SFI(1-s^*)$ to find all $(1-s^*)$-similar sid's to $\bar{q_b}$. According to theorem 2, these are the set identifiers that are at most s-similar to q_b. To distinguish this data structure from $SFI(1-s^*)$, since we are querying based on the complement of a query q_b, we refer to it as $DFI(s^*)$ and to the result retrieved from DFI as $DissimVector(s^*, q_b)$.

4.3 Building The Index

Now we are ready to describe the details of our indexing technique.

Preprocessing Partition the Hamming similarity range $[0,1]$ into $k+1$ intervals, by selecting k points $0 = r_0 < r_1 < \ldots < r_m = t_m < \ldots < t_k < t_{k+1} = 1$ and build $k+1$ data structures $DFI(r_1), \ldots, DFI(r_m), SFI(t_m), \ldots, SFI(t_k)$. The $k+1$ points are kept ordered in an in memory data structure.

Query Processing Given a query $(q, [\sigma_1, \sigma_2])$ derive q_b from q applying the embeddings of section 3. Then use the in memory data structure to determine points lo, up that minimally enclose the range $[\sigma_1, \sigma_2]$. The answer to the query depends on the following cases for the points lo and up.

- If $lo = r_i$ and $up = r_j$ let $A = (DissimVector(up, q_b) \setminus DissimVector(lo,q_b))$
- If $lo = t_i$ and $up = t_j$ let $A = (SimVector(lo, q_b) \setminus SimVector(up, q_b))$
- If $lo = r_i$ and $up = t_j$ let $A = ((DissimVector(r_m, q_b) \setminus DissimVector(lo, q_b)) \cup (SimVector(t_m, q_b) \setminus SimVector(up, q_b)))$
- In the special cases of $lo = 0$ and $up = 1$ the set $DissimVector(lo, q_b)$ and $SimVector(up, q_b))$ are empty, so we don't have to perform the corresponding queries.
- Let $set(x)$ denote the set corresponding to a sid in A. Return $\{set(x), x \in A | \sigma_1 \leq sim(set(x), q) \leq \sigma_2\}$

Since answer A contains sids corresponding to false positives, they have to be excluded from the final answer returned to the user. Consequently one has to retrieve all sets from $\mathcal{S}$ with sids in A, evaluate their similarity with the query set and return only those with similarity within the requested similarity range.

The proposed indexing scheme readily supports dynamic operations on the set collection (insertions, deletions), since the data structure primitives it uses, namely hash indices, are fully dynamic. To complete the description of the structure, we should specify the choice of various parameters associated with it. Since filter functions are probabilistic and incur error in terms of false positives and false negatives one primary objective is to maximize the overall accuracy and performance of the indexing scheme. Moreover the exact location of the $k + 1$ points has to be decided as well as the allocation of DFI and SFI structures to those points. In addition the parameters of each DFI and SFI structure have to be determined. In the following section we formulate and solve optimization problems related to the choice of those parameters.

5 Index Optimization

In this section we present our technique for building the index. The proposed indexing scheme is approximate since each probabilistic filter function entails a certain number of false positives and false negatives, quantified in expectation by equations 5,6. A specific query $Q = (q, [\sigma_1, \sigma_2])$ has an exact answer $a(q)$ in $\mathcal{S}$. These are the sets in $\mathcal{S}$ that are between σ_1 and σ_2 similar to q. Evaluating Q using the proposed indexing scheme will yield an answer $ia(q)$, where $ia(q) \subset a(q)$. Let $SFI(lo)$ and $SFI(up)$ be the FIs that minimally enclose the similarity range $[\sigma_1, \sigma_2]$ ($lo < up$). The error in the answer $(a(q) \setminus ia(q))$ is due to false negatives that are induced by $SFI(lo)$, and to false positives that are induced from $SFI(up)$:

$$ia(q) = a(q) \setminus P(up) \setminus N(lo) \tag{8}$$

The index will also bring in memory an additional number of sets, since $[\sigma_1, \sigma_2]$ is not perfectly aligned with partition end points; specifically

$$ie(q) = SimVector(lo, q_b) \setminus SimVector(up, q_b) \setminus ia(q) \tag{9}$$

These sets are between $[lo, up]$ similar to q.

We use the notions of *precision* and *recall* widely used in Information Retrieval [vR79] to quantify the performance of the indexing scheme. Intuitively, recall quantifies how accurate our index is (a recall close to 1 is desirable), and precision quantifies how efficient our index is (a precision close to 1 is desirable). Unfortunately, as we will see, it is difficult to optimize for both at the same time. Increasing one is likely to decrease the other. For example, the simple brute force algorithm has recall equal to 1, but the precision is the lowest for every query because all sets are retrieved.

A probabilistic filter function allows to quantify the number of false positives and false negatives incurred by a query only in expectation (Equations 5,6). Thus our efforts for optimization of our indexing scheme are centered towards optimizing the *expected precision* and the *expected recall* over the workload of all possible queries, which we assume to be uniformly distributed, both in terms of set queries and similarity values[2]. Let $E_{ia}(\sigma_1, \sigma_2) = Average_{q \in \mathcal{S}}\ (|ia(q, [\sigma_1, \sigma_2])|)$ and $E_a(\sigma_1, \sigma_2) = Average_{q \in \mathcal{S}}\ (|a(q, [\sigma_1, \sigma_2])|)$ Thus, $E_{ia}(\sigma_1, \sigma_2)$ is the expected number of similar sets retrieved using the index, over all queries in the similarity range $[\sigma_1, \sigma_2]$, and $E_a(\sigma_1, \sigma_2)$ is the expected size of the answer over all queries in the same range. Then we define expected recall of a similarity range $[\sigma_1, \sigma_2]$ as:

Definition 8 (Expected Recall) *We define the expected recall of the similarity range* $[\sigma_1, \sigma_2]$ *as:*

$$ExpectedRecall(\sigma_1, \sigma_2) = \frac{E_{ia}(\sigma_1, \sigma_2)}{E_a(\sigma_1, \sigma_2)} \tag{10}$$

and the expected precision as:

Definition 9 (Expected Precision) *The expected precision of the similarity range* $[\sigma_1, \sigma_2]$ *is:*

$$ExpectedPrecision(\sigma_1, \sigma_2) = \frac{E_{ia(q)}(\sigma_1, \sigma_2)}{E_{ia}(\sigma_1, \sigma_2) + E_{ie}(\sigma_1, \sigma_2)} \tag{11}$$

We will be optimizing the *expected worst case* of recall (or precision) over all similarity intervals using our indexing scheme [3].

Evaluating equations 5,6 requires knowledge of the similarity distribution function $\mathcal{D}_{\mathcal{S}}$ of the underlying

[2] In the sequel we use the terms precision (recall) and expected precision (expected recall) interchangeably

[3] We choose to relax the definitions of expected recall and expected precision using formulas 10 and 11 as opposed to the average over all queries Q of $Recall(Q)$ and $Precision(Q)$ to make subsequent optimizations tractable

set collection $\mathcal{S}$. From the definition of $\mathcal{D}_{\mathcal{S}}$, and when q is drawn from $\mathcal{S}$, we have

$$\sum_{q \in \mathcal{S}} a(s_1, s_2) = 2 \int_{s_1 \leq s \leq s_2} \mathcal{D}_{\mathcal{S}}(s)\, ds \Rightarrow \tag{12}$$

$$E_a(s_1, s_2) = \frac{2}{|\mathcal{S}|} \int_{s_1 \leq s \leq s_2} \mathcal{D}_{\mathcal{S}}(s)\, ds \tag{13}$$

since in the integral of $\mathcal{D}_{\mathcal{S}}$ we count each pair of sets once but in the sum of answers each pair is counted twice. $\mathcal{D}_{\mathcal{S}}$ can be computed exactly in a preprocessing step, by computing all pairwise similarities between sets in $\mathcal{S}$. The following lemma provides an efficient way to approximate it:

Lemma 1 *Given a collection $\mathcal{S}$ of sets of size n, and a sample bound b, we can compute a random sample of the $\frac{|\mathcal{S}|(|\mathcal{S}|-1)}{2}$ pairwise similarities between sets in $\mathcal{S}$ of size b in one dataset pass.*

As corollary of this lemma, we can efficiently approximate the similarity distribution function $\mathcal{D}_{\mathcal{S}}$ by sampling. Knowledge of this distribution is crucial in quantifying our expectation of false positives and false negatives.

The main parameter in our indexing scheme is the total amount of space that can be allocated to the index structure. Effectively this is translated to a specified number K of hash tables we can use, since each hash table requires the same space for storage, irrespective of the Filter Index (FI) it is assigned to. A problem of interest therefore is the allocation of a fixed number of hash tables to a number of Filter Indices so that the recall and the precision is maximized. For this problem to be fully specified, the number of FIs that we will use, their location on the similarity range $[0, 1]$ (and thus the number of similarity intervals), their kind (SFI,DFI) and the number of hash tables assigned to each FI, should be decided. We present a number of lemmas showing the influence of these parameters on the expected worst case of recall and precision. Based on these lemmas we then formulate and solve the design of the overall index as an optimization problem. The choose to present proof intuition for the lemmas, highlighting the main idea behind the proof. We defer detailed proofs for the full version of this paper.

5.1 Optimizing Recall

The first observation is that the number of false positives and false negatives incurred by the FIs has a large effect on recall.

Lemma 2 *Given b hash tables, and K Filter Indices, the expected worst case recall for a given query set q is maximized when the error incurred at each Filter Index is approximately the same.*

Using Lemma 2 we can show that the expected worst case recall increases when we reduce the number of FIs while the number of hash tables remains constant.

Lemma 3 *Given b hash tables, the expected worst case recall increases when the number of Filter Indices decreases.*

This result is intuitive. If there are fewer false positives or negatives, the answer we find is more accurate. If we take this to the limit we arrive to the brute force algorithm: consider all sets in order to find the ones in the query answer.

5.2 Optimizing Precision

When optimizing for precision, the location of Filter Indices is important. If two consecutive FIs $SFI(lo)$ and $SFI(up)$ are too far apart, a narrow query $(q_b, [\sigma_1, \sigma_2])$ where $lo < \sigma_1 < \sigma_2 < up$ and $(up - lo) >> (\sigma_2 - \sigma_1)$ would retrieve all sets in $SimVector(SFI(lo), q_b) \setminus SimVector(SFI(up), q_b)$, even though the actual answer $a(q, [\sigma_1, \sigma_2])$ would be very small.

Our approach for deciding the location of Filter Indices is based on an equidepth decomposition of the similarity range $[0, 1]$. Using the similarity distribution function one can partition the [0,1] similarity range into intervals that contain approximately the same number of sets.

Definition 10 (Equidepth Decomposition) *Given a collection of sets $\mathcal{S}$, and a number k, a k-wise equidepth decomposition of the similarity range $[0, 1]$ is a decomposition of the range into k intervals $0 = c_0, c_1, \ldots, c_{k-1}, 1 = c_k$ such that for $0 < i < k - 1$,*

$$\int_{c_i \leq s \leq c_{i+1}} \mathcal{D}_{\mathcal{S}}(s)\, ds = \frac{1}{k} \int_{0 \leq s \leq 1} \mathcal{D}_{\mathcal{S}}(s)\, ds \tag{14}$$

Since $\int_{0 \leq s \leq 1} \mathcal{D}_{\mathcal{S}}(s)\, ds = \frac{|\mathcal{S}|(|\mathcal{S}|-1)}{2}$, it follows that $\int_{c_i \leq s \leq c_{i+1}} \mathcal{D}_{\mathcal{S}}(s)\, ds = \frac{|\mathcal{S}|(|\mathcal{S}|-1)}{2k}$. It is important to notice the semantics of this equidepth decomposition. It is possible that for a given query set q some of the intervals contain more similar sets to q in the dataset than other intervals. However, for all intervals $[c_i, c_{i+1}]$, the *expected* number of sets $E_a(c_i, c_{i+1}) = \frac{2}{|\mathcal{S}|} \int_{c_i \leq s \leq c_{i+1}} \mathcal{D}_{\mathcal{S}}(s)\, ds = \frac{|\mathcal{S}|-1}{k}$ is the same for all intervals. In other words if we take the average of similar sets over all query sets q and a given interval, this average is the same for all intervals, assuming that queries q are equi-probable. Such a k-wise decomposition can be computed using $\mathcal{D}_{\mathcal{S}}$, constructing the cumulative distribution from it and determining k quantiles. The following lemma shows that equidepth decomposition optimizes the expected worst case precision.

Lemma 4 *Given K Filter Indices, the expected worst case precision for queries with expected answer size at least a is optimized when the Filter Indices are arranged in equidepth fashion.*

The effect of the number of intervals on expected precision is shown by the following lemma.

Lemma 5 *Given K FIs, decomposing the similarity range $[0,1]$ into $m+1$ ranges, increases the worst case expected precision for queries with expected answer of at least a, over decomposing the similarity range into m ranges, if $m < \frac{T}{1-T}$ where T is the expected worst case recall.*

5.3 Placing Similarity and Dissimilarity Filter Indices

Using both Similar and Dissimilar Filter Indices allows minimization of the amount of work one has to do in memory. Each range similarity query $(q, [\sigma_1, \sigma_2])$ involves a set difference operation between $SimVector(lo, q_b)$ and $SimVector(up, q_b)$ or between $DissimVector(lo, q_b)$ and $DissimVector(up, q_b)$. We wish to minimize the expected size of $SimVector(.)$ and $DissimVector(.)$. To decide the kind of each filter index we adopt the following strategy. We partition the similarity range $[0,1]$ into $[0,\delta]$ and $[\delta,1]$ so that

$$\int_{0\leq s\leq \delta} \mathcal{D}_S(s)\,ds = \int_{\delta\leq s\leq 1} \mathcal{D}_S(s)\,ds \tag{15}$$

If we use SFIs for similarities over δ and DFIs otherwise, we have a bound of $\int_{\delta\leq s\leq 1} \mathcal{D}_S(s)\,ds$ on how large a $SimVector(.)$ or a $DissimVector(.)$ can be. For the FI closest to δ, we place both an SFI and a DFI. In the next section we show how to specify the number and the locations of the Filter Indices.

5.4 Index Construction Algorithm

We are now ready to present our approach for designing the overall index. Lemmas 3 and 5 provide a strategy for optimization. We formulate the following optimization objective:

Objective 2 *Given a threshold T on the expected worst case recall, construct an index that optimizes precision (and thus performance) while the expected worst case recall remains above T.*

We give an optimization algorithm that determines the maximum number of intervals that can be used, while the expected worst case recall remains at least T. The input to the algorithm is a collection of sets S, a bound b on the number of hash-tables that can be used, and the threshold T on expected worst case recall. Based on exact knowledge of the similarity distribution function $\mathcal{D}_S$ (or an approximation thereof using lemma 1) we determine (using equation 15) range $[0,\delta]$ where DFIs will be placed (SFIs are added in $[\delta,1]$). The next step, is designing the locations of the FIs. We start by decomposing the similarity range $[0,1]$ into one interval, and increase the number of intervals by one at each iteration, choosing the location of the FIs using an equidepth decomposition of $\mathcal{D}_S$. Lemma 4 shows that this decomposition optimizes the expected worst case recall. The algorithm distributes the number of hash-tables to the FIs using the Greedy Algorithm that is described below. To compute the expected worst case recall we use lemma 2, and compute the expected recall of similarity ranges of width t around the FIs.

```
Index Construction
    Input: Dataset $\mathcal{S}$, size bound $b$, recall threshold $T$.
    set $i = 1$; recall= 1;
    Determine $\mathcal{D}_S$ either exactly or approximately
    using lemma 1.
    compute the value of $\delta$; Every FI place in $[0,\delta]$
    is a DFI; FIs in $[\delta,1]$ are SFIs; We
    place both an SFI and a DFI, closest to $\delta$.
    while (recall> $T$ and $i < \frac{T}{1-T}$) {
        increment $i$;
        place $i$ FIs in equidepth fashion
        allocate $b$ hash tables using Greedy()
        compute the expected worst case recall
        if recall< $T$, return $i-1$
        }
```

Figure 4: Index Construction Algorithm

Lemma 3 shows that expected worst case recall decreases when the number of partitions increases, and so the algorithm terminates if the expected worst case recall that we calculate drops below the threshold. Lemma 5 shows that the expected worst case precision improves as long as the number of partitions is below $\frac{T}{1-T}$, so the algorithm terminates when this value is reached. So the index that we construct optimizes expected worst case precision (and thus performance) while keeping expected worst case recall (and thus accuracy) above a user defined threshold. The algorithm is shown in Figure 4.

The Index construction algorithm invokes algorithm Greedy (shown in Figure 5) to assign hash tables to each Filter Index. The following lemmas show that the Greedy algorithm optimizes expected worst case recall.

Lemma 6 *Given K Filter Indices, the Greedy algorithm of figure 5 determines the allocation of b hash tables that maximizes the expected worst case recall.*

6 Experimental Results

We have implemented the proposed set indexing technique and in this section we report on its performance. Our goal is to examine the performance of the scheme varying parameters of interest and to evaluate its accuracy. We use real data sets in our evaluation. The data sets are http logs obtained from web sites, which we parse and record for each unique IP address the collection of http log strings associated with that address. We used two such data sets, one obtained from the winter Olympics web site at Nagano Japan and the other obtained from the central web site of a large corporation.

```
The Greedy Algorithm
1. For $i = 1$ to $b$:
      Find the FI that reduces the expected number of false positives and false negatives the most by
      adding a new hash table, and assign hash table $i$ to this FI.
```

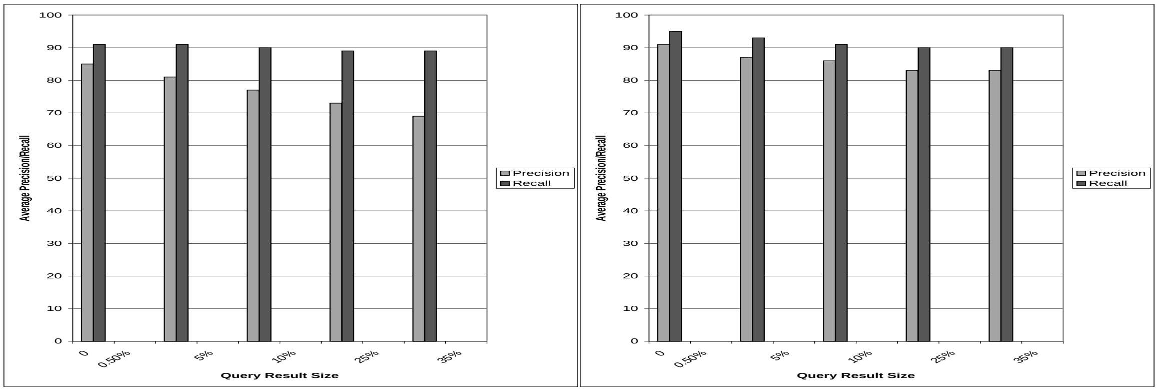
Figure 5: Greedy Allocation of hash tables to Filter Indices

Parsing the first log we generated a data set, consisting of 200,000 sets, of size approximately 400MB, which we refer to in the experiments that follow as *Set1*. Parsing the second data set we created a data set of 200,000 sets of size approximately 500MB which we refer to as *Set2*. Due to space limitations we present only a part of our results for *Set1*.

Query answering using the proposed indexing technique is a two step process. First the set of candidate set identifiers is fetched from disk for a given query, and then the corresponding sets are retrieved from disk, using a conventional data structure such as a B-tree supporting queries on set identifier. The sets retrieved have to be checked against the query set to eliminate false positives. We applied the techniques of section 3 to preprocess the data sets. The parameters of the indexing scheme are determined by the solution of the optimization problem of section 5. The input to this optimization is the constraint of space the user is willing to allocate for indexing and the threshold for recall one wishes to satisfy. We thus conducted experiments to assess the efficiency of our solution to the optimization problem of section 5 in terms of meeting the optimization objectives, namely meeting the recall goals. Thus the first set of experiments reports on the precision and the recall of the indexing scheme for both data sets used, as a function of the optimization constraint, namely disk space allocated for indexing. Moreover we evaluated the overall response time of our indexing scheme including both the time to retrieve the candidate set identifier list as well as the time to filter out possible false positives. The second set of experiments evaluates the overall response time of indexing compared with the default scheme of sequential scan. Sequential scan, simply scans the entire set collection and evaluates the similarity between the query set and the sets in the database, reporting only those sets with similarity inside the target similarity range. In our system this ratio is estimated approximately as $r_{IO} = \frac{ran}{seq} \approx 8$, where ran is the time to perform a random IO for a page and seq the time to perform a sequential IO for a page (a random disk accesses is approximately 8 times more expensive than a sequential disk access). Let the average set size be a pages. One expects that for a set collection of $|S|$ sets, our indexing scheme provides performance benefits, when the query result size $|Q|$ is less than $\frac{|S|(seq \cdot a)}{(ran+seq \cdot a)}$. When a increases (the set collection becomes larger in size) the bound for the query result size becomes larger. For our data sets this estimation translates to a result size of approximately 47,000 sets or 23% of the total number of sets in the dataset. This percentage will increase as the average size of sets in our collection increases. This estimation is rough and ignores the additional processor overhead of sequential scan, since for all sets in $\mathcal{S}$ one has to evaluate the similarity with a query set.

Based on this observation we report performance results as a function of the query result size. We ask random queries obtained from the set collection and we classify them according to the size of candidate set identifiers returned from the index. We form 5 buckets, one for queries with candidate result size less than 0.5% of the total number of sets in our collection, between 0.5% and 5%, between 5% and 10%, between 10% and 25% and finally one for queries with candidate result size between 25% and 35% of the total number of sets in our collection. For each of these buckets, we collect results for one thousand queries and report response time as well as recall and precision averaged over all queries. The query sets are chosen at random from the set collection and the bounds for each similarity range associated with a query are chosen at random as well.

Figure 6(a)(b) shows the precision and recall bars per bucket for both data sets. Figure 6(a) shows the graphs for a 500 total index budget and Figure 6(b) for a total index budget of 1000. Using 500 hash tables we optimize the index for 90% average recall for set1 and for 90% average recall for set2. In both cases the optimization objective is achieved. In the figures, precision, appears to decrease as the result size increases. To explain this behavior, one has to consult the similarity distribution function of the data sets. This function drops sharply as similarity increases. Consequently large query result sets are produced for queries involving small similarity values (closer to zero similarity). In addition small result sizes are produced by queries involving larger similarity values (close to similarity one). As a result, when the threshold for average recall is met by the optimization, the area below the corresponding filter functions involved, is larger for queries with large result sizes than the area below filter functions involved for queries with small result sizes. When we increase the number of hash tables available (Figure 6(b)) we optimize for 90% recall in both data sets. The optimization objectives are met in both cases. Precision consistently increases, since the number of similarity ranges allocated by the algorithm increases in both cases, due to higher availability of hash tables to be allocated. Moreover, precision drops in both cases



(a) 500 hash tables

(b) 1000 hash tables

Figure 6: Precision/Recall bars for set2

as result size increases, for reasons similar to the case of 500 hash table budget.

Figures 7(a)(b) present the overall response time (averaged over 1000 queries per result size bucket). We present the total time in seconds required by sequential scan (labeled *Scan* in the figures) as well as the time required by the proposed indexing scheme to report the query results. The time is reported separately for IO and processor. These graphs correspond to the case when total space available is 1000 hash tables, using 100 min-hash values. The graphs are very similar for the case of 500 hash tables (slightly lower response time for indexing). We observe that for all queries with result size (in terms of number of sets) less than 25% of the total number of sets in our collection, the proposed indexing scheme offers performance benefits. This is consistent with our analytical expectation. The total time to report the result of a query increases with the result size, since more random IOs have to be performed to retrieve the candidate sets and subsequently remove them from the actual result reported to the user.

7 Related Work

Similarity queries have been studied in time series databases and have attracted lots of research interest [AFS93, ALSS95, FRM94]. The unordered nature of sets and the freedom to represent categorical attributes in sets makes the techniques developed for the time series domain inapplicable. Manber [Udi94] considered the problem of retrieving similar files.

Indyk et al., [IM98] introduced *Locality Sensitive Hashing* for L_1, L_2 and hamming space, and applied it to nearest neighbor search in high dimensional data sets using the L_1 metric [AGM99] and subsequently to identifying interesting associations between large item sets encoded using their indicator vectors [CDF$^+$00]. Locality Sensitive Hashing uses a probabilistic filter function to preserve L_1 distance between vectors in an approximate way. Indyk [Ind00], showed a reduction from the furthest neighbor problem to the nearest neighbor problem, using a method similar to our Dissimilarity Filter Index.

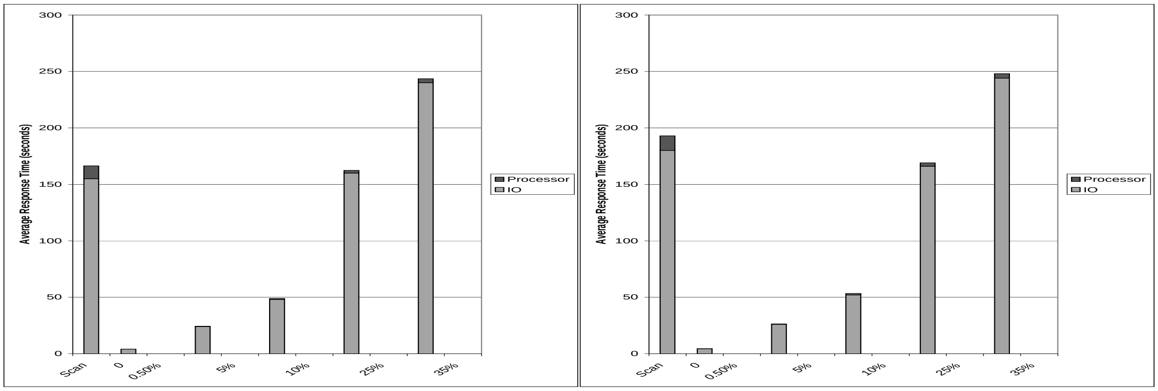
Signature based techniques [Fal85] have been applied to the problem of retrieving subsets of a given set in a large collection of sets [YIO93]. Such techniques are based on an encoding via hashing of sets which is subsequently maintained as a file and scanned in its entirety to answer a query. No indexing mechanism is provided. Such techniques however cannot provide any form of guarantee on their accuracy and analysis is based on restrictive assumptions such as uniformity.

8 Conclusion

In this paper, we considered the problem of indexing sets based on similarity. We have proposed indexing techniques based on hashing that can answer similarity queries on sets in an approximate way. We did so by introducing various embeddings during preprocessing of a set collection. Then we described two data structure primitives, Similarity Filter Index and Dissimilarity Filter Index, and used multiple instances of them to construct our index. We also showed how it is possible to optimize the resulting index for accuracy and we presented an experimental evaluation of our approach showing large performance benefits while achieving acceptable accuracy.

9 Acknowledgments

We wish to thank Suhas Diggavi and N. Sloane of AT&T Research for educating us on coding theory and answering our numerous questions. C. Faloutsos and H. V. Jagadish offered constructive criticism, time to listen to our thoughts, comments and lots of encouragement. Sudipto Guha and H. V. Jagadish read drafts of this paper and offered comments that improved the presentation. Balachander Krishnamurthy, generously provided the data sets used in this study. Finally, we wish to thank Piotr Indyk for all his help and comments.



(a) set1

(b) set2

Figure 7: Average Query Response Times

References

[AFS93] R. Agrawal, C. Faloutsos, and A. Swami. Efficient Similarity Search in Sequence Databases. *Proc. of the 4th Int'l Conference on Foundations of Data Organization and Algorithms*, pages 69–84, October 1993.

[AGM99] P. Indyk A. Gionis and R. Motwani. Similarity Search In High Dimensions Via Hashing. *Proceedings of VLDB*, 1999.

[ALSS95] R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim. Fast Similarity Search in the Presence of Noise, Scaling and Translation in Time-Series Databases. *Proceedings of VLDB*, pages 490–501, September 1995.

[BCFM98] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Minwise Independent Permutations. *Proceedings of STOC*, pages 327–336, 1998.

[BGMZ97] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic Clustering on the Web. *Proceedings of WWW6*, 1997.

[CDF$^+$00] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, R. Motwani, J. Ullman, and C. Yang. Finding Interesting Associations Without Support Prunning. *International Conference on Data Engineering*, 2000.

[CJK$^+$01] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. *International Conference on Data Engineering, to appear*, April 2001.

[CKKM00] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity Estimation for Boolean Queries. *PODS*, May 200.

[Coh97] E. Cohen. Size-Estimation Framework With Applications To Transitive Closure And Reachability. *Journal Of Comput. Syst. Sciences, 55*, pages 441–453, 1997.

[Fal85] C. Faloutsos. Fast Access Methods For Text. *ACM Computing Surveys 17(1)*, pages 49–74, 1985.

[FRM94] Christos Faloutsos, M. Ranganathan, and I. Manolopoulos. Fast Subsequence Matching in Time Series Databases. *Proceedings of ACM SIGMOD*, pages 419–429, May 1994.

[GG98] V. Gaede and O. Gunther. Multidimensional Access Methods. *ACM Computing Surveys, No 30, Vol 2*, pages 170–231, March 1998.

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *Proceedings of VLDB*, pages 562–573, August 1995.

[IM98] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse Of Dimensionality. *30th Symposium on Theory of Computing*, 1998.

[Ind00] P. Indyk. Dimensionality Reduction Techniques For Proximity problems. *SODA*, pages 371–378, 2000.

[MS93] F. J MacWilliams and A. Sloane. *The Theory Of Error Correcting Codes.* North Holland, 1993.

[SM96] M. Stonebraker and D. Moore. *Object Relational Databases: The Next Wave.* Morgan Kauffman, June 1996.

[Udi94] Manber Udi. Finding Similar Files in a Large File System. *Winter USENIX Technical Conference*, October 1994.

[vR79] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, 1979.

[YIO93] M. Kitagawa Y. Ishikawa and N. Obho. Evaluation of Signature Files as Set Access Facility in OODBs. *Proceedings of ACM SIGMOD*, pages 247–256, June 1993.