PROBABILISTIC SIMHASH MATCHING

A Thesis

by

SADHAN SOOD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2011

Major Subject: Computer Science

PROBABILISTIC SIMHASH MATCHING

A Thesis

by

SADHAN SOOD

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Dmitri Loguinov
Committee Members,	Narasimha Annapareddy
	James Caverlee
Head of Department,	Valerie Taylor

August 2011

Major Subject: Computer Science

ABSTRACT

Probabilistic Simhash Matching. (August 2011)

Sadhan Sood, B.Tech., Cochin University of Science and Technology-Kochi

Chair of Advisory Committee: Dr. Dmitri Loguinov

Finding near-duplicate documents is an interesting problem but the existing methods are not suitable for large scale datasets and memory constrained systems. In this work, we developed approaches that tackle the problem of finding near-duplicates while improving query performance and using less memory. We then carried out an evaluation of our method on a dataset of 70M web documents, and showed that our method works really well. The results indicated that our method could achieve a reduction in space by a factor of 5 while improving the query time by a factor of 4 with a recall of 0.95 for finding all near-duplicates when the dataset is in memory. With the same recall and same reduction in space, we could achieve an improvement in query-time by a factor of 4.5 while finding first the near-duplicate for an in memory dataset. When the dataset was stored on a disk, we could achieve an improvement in performance by 7 times for finding all near-duplicates and by 14 times when finding the first near-duplicate.

To my parents, brother and Sheethal

ACKNOWLEDGMENTS

I would like to sincerely thank Dr. Loguinov for giving me the opportunity to work with him in the Internet Research Lab. I believe that working here has prepared me to deal with future challenges in a much better way. I would like to thank Dr. Annapareddy and Dr. Caverlee for being on my committee.

I would also like to thank my colleagues at Internet Research lab, especially Siddhartha and Rock. Finally, all my achievements were impossible without the support of my parents and Sheethal.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Web crawl often results in a huge amount of data download [1] and finding useful data during runtime which can affect crawl behavior is a challenge. Learning to find data patterns can free the crawling resources for more unique pages and make crawl more responsive to decision made by the crawl administrator. For instance, a crawler can choose to follow/not follow a newly downloaded page if it is similar to an already seen page or clusters the downloaded data in real time such that a crawl administrator can reduce the priority of the links from spam pages, direct the crawl towards certain topics or assign lower budgets to forums and blogs.

This can be done if there is a way to efficiently and accurately determine similar document(s) in a dataset and cluster them together. Clustering data streams require knowing near neighbors of an incoming document [2] to make fast decisions about cluster membership or to create new clusters if none of the existing ones are a good match. The most critical problem with this approach is to find near neighbors of a new arriving document in the least possible time. This is however similar to finding near-duplicates of the new arriving document in the existing data set and therefore in this paper, we propose ideas to efficiently solve this problem. Two documents can be termed near-duplicates if degree of similarity between them is greater than a similarity threshold using a similarity function. Mathematically, it can be said that: Given a set of existing n documents $D = \{d_1, d_2, \ldots, d_n\}$, a similarity function $\omega \in [0, 1]$ and a new arriving document d_q, find all document $d \in D$ such that:

$$\omega(d, d_q) \geq t, \tag{1.1}$$

The journal model is *IEEE Transactions on Automatic Control*.

where t is the similarity threshold for near-duplicates on similarity function ω. The similarity functions often used for evaluation are cosine similarity between document feature vectors, Jaccard similarity between k-grams, hamming distance between document signatures, L_p norm, etc.

Two near-duplicate documents share similar topic and content but differ in small number of words and html tags and if clustered as part of a data set will almost always share the same cluster. Near-duplicates on the internet exist as spam, forums and blogs generated from automated templates, plagiarized content [3], similar news stories, typing errors, mirrored web sites, etc. Fetterly et al. in their research [4] on evolving web document clusters found that 29.2% pages on the web are very similar and 22.2% are virtually identical. Recently, hash-based methods are proved promising for near-duplicate search. The idea is to embed document information into small binary strings such that semantically similar documents have almost similar binary strings differing in few bit positions. The general idea is based on approximate nearest-neighbor algorithms where the objective is to find points which are at most $(1+\epsilon)$ times away from the actual nearest-neighbor. This speeds-up the search but returns an approximation of the exact query results according to a similarity measure.

A common technique for approximate nearest-neighbor search in high-dimensional space is Locality Sensitive Hashing (LSH) which was introduced by Indyk and Motwani [5]. The method uses a family of hash functions to hash similar documents to same buckets with high probability and while doing nearest-neighbor search on a given query document, searches only those buckets to which query document is hashed to in multiple tables generated using different hash functions. This requires a large number of hash tables for maintaining a good search quality which leads to duplication of data and is not efficient in terms of memory usage. In [6], Manku et al. used `simhash` [7] a version of LSH which approximates cosine similarity between

nearest-neighbors in their implementation of near-duplicate search on 8B web-pages. The method was first proposed by Charikar [7] and later evaluated on web-documents by Henzinger [8] for the first time.

Their idea requires multiple permuted copies of existing fingerprints F be maintained in memory and to find near-duplicate of a query fingerprint q with a maximum allowed hamming distance h, a binary search followed by linear scan is performed in all tables until at least one fingerprint $p \in F$ is found such that hamming distance between p and q is at most h or no more matching fingerprints exist in the data set. A large number of permutations increase the space requirement while reducing the number of fingerprints being checked and vice-versa. This duplication of data limits the number of fingerprints that can be stored in main-memory S in online mode although we found that increasing the number of tables not always result in improved performance. In batch mode, the existing fingerprints F are stored on disk and for a new batch of queries Q for which the near-duplicates are to be found, search goes from $F \rightarrow Q$ by creating permuted copies of Q. While this does not pose a limitation on the size of Q if $|Q| << |S|$, we show that this technique is sub-optimal and limits the performance. The method because of its high space requirement is not suitable for memory constrained systems and hence we propose our new method for near-duplicate search on large data sets.

A. Our Contributions

In this paper, we present a system to fast detect near-duplicate matches for a new arriving page in an existing data set which is more space efficient and achieves better query speed for both online and batch queries by sacrificing a small percentage of recall. For a crawler, to decide to follow/not follow a newly downloaded page, just

one match in the existing dataset is enough whereas for deciding its cluster membership, all near-duplicate matches are needed. Therefore we present our solution, the *Probabilistic Simhash Matching* (PSM) system for both the problems: first where search finds all near-duplicate documents (PSM_A) and second where search stops after finding the first near-duplicate document (PSM_F). We compare our results with the solution proposed by Manku et al. [6] in terms of gain in query speed and reduction in space requirement for both cases.

In our experiments on a crawl data set of 60M documents in online mode, we could achieve a reduction in space by factor of 5 while improving the query time by a factor of 4 with a recall of 0.95 for PSM_A. With same recall and same reduction in space, we could achieve an improvement in query-time by a factor of 4.5 for PSM_F. For batch mode, PSM_A can perform ≈ 7 times better and PSM_F can perform ≈ 14 times better than [6]. We propose a solution that given a fingerprint can predict with reasonable accuracy the set of bits that have a higher probability than others of getting flipped between a near-duplicate document pair and term them as *weak* bits. Using this information for any f bit fingerprint q and a maximum hamming distance h, we iteratively flip a combination of *weak* bits and combine it with linear scan to find near duplicate matches in the existing data set. Our method does not require any extra copies of permuted fingerprints to be maintained and is better scalable for large data sets. We also show that for two document x,y and their 64-bit fingerprints p,q: hamming distance $H(p, q) \leq 3$ is as good a similarity measure as cosine similarity with $\cos(x, y) \geq 0.9$ and use this as similarity threshold for near-duplicates in all of our experiments.

CHAPTER II

RELATED WORK

Identical duplicates are easier to find in constant time by normal hashing techniques whereas finding almost similar pages of a newly downloaded page is an $O(n)$ operation which require comparisons with every other document in the data set. This is not efficient and scalable if done for crawled documents which are in scale of billions and is therefore a challenge. Data structure like R-tree [9] and Kd-tree [10] solve the problem of finding k-Nearest-Neighbors(kNN) [11] but their performance degrades with increasing number of dimensions. For example, Kd-tree hierarchically decomposes space along different dimensions and answers nearest neighbor queries in logarithmic time but performs no better than linear scan even if the number of dimensions become greater than eight [12]. Finding exact near-duplicate web documents in a high dimensional space is difficult because of the inherent sparsity of the data points. Besides computational complexities involved with high dimensional data, analysis has shown that every data point tends to be equidistant from all other data points in high dimension for all practical data loads and the amount of data required for maintaining a given spatial density increases exponentially with increase in dimensions. Because of such high cost, common approach is to solve the problem of finding approximate near-duplicate.

Several signature based schemes have been proposed in this direction. LSH is one such popular technique which can map data points close together in Euclidean space(or Cosine space) to a smaller dimension hamming space. To improve the performance of LSH techniques, data aware hashing techniques have been proposed which uses machine learning to improve near neighbor search. In [13], the authors proposed a two step stacked Restricted Boltzman Machine (RBM) to generate compact binary

codes with low hamming distance between similar documents. In Forgiving Hashing technique [14], an AdaBoost classifier is first trained on positive and negative similar pairs and output of all weak learners on a given document is taken as its binary code. These techniques though are proven to perform better than LSH are not suitable for real-time systems sice they need training and the performance is dependent on the training dataset used during classification.

One of the earliest technique for near-duplicate detection is *Shingling* where for a given document d and a positive integer k, k shingles are defined as the set of all consecutive sequence of k tokens in d $S(d)$. The shingles are further encoded into 64-bit hash values $H(d)$ and Jaccard coefficient is calculated between shingle vectors of two documents as similarity measure. The technique still needed to check pairwise Jaccard coefficient and this problem was solved in [15], by Broder et al. who introduced a technique called Min-wise independent permutations. For a document d_j, a random permutation π is used to permute $H(d_j)$ into $\pi(d_j)$ and if x_j^π is the smallest integer in $\pi(d_j)$, then:

$$J(S(d_i), S(d_j)) = Pr(x_i^\pi = x_j^\pi). \tag{2.1}$$

A document *sketch* $\psi(d_j)$ is computed over multiple random permutations and two documents are termed near-duplicates if their sketch overlap with a preset threshold. Near-duplicates with document sketches can be found much faster than between their shingles. Hoad and Zobel [16] study different ideas for selecting k-grams for calculating shingles effectively.

In I-Match algorithm given by Chowdhry et al. [17] first define a lexicon L which is constructed as the union of all terms in the document corpus. The lexicon is then pruned using different collection statistics, the common being the inverse document frequency (IDF) where terms with high and low IDF values are removed

from the collection. Each document vector is then modified by removing terms which no longer exist in the lexicon and a document signature is generated using SHA1 hashing algorithm. The general idea of the algorithm is that two documents will be near-duplicates if and only if their signatures match. The algorithm generates a lot of false-positives for near-duplicate search and therefore in a modified paper, authors propose to create multiple randomized lexicons. They create multiple signatures and two documents are termed near-duplicates only if certain number of signatures match.

In another paper by Theobald et al. [18] on detecting near-duplicate new articles, authors propose the idea of generating "localized signatures" generated from text around stop words in a document. The basic idea is that stop words are uniformly distributed in an English text but occur very rarely in banners or ads. Hence, they define "spot signature" of a document as a collection of all k-tokens after a stop word in the document. They also propose a collection partitioning and pruned inverted index technique for efficiently computing Jaccard similarities on spot signatures.

A. SIMHASH

Charikar's `simhash` [7] is a locality sensitive hashing (LSH) based fingerprinting technique which uses random projections to generate compact representation of a high dimension vector. It has a special property that hamming distance between two document fingerprints is small if the cosine similarity between their feature vectors is high and vice-versa. LSH scheme was first introduced by Indyk and Motwani [5] who showed that such hashing scheme can be used to build efficient data structures for solving approximate near-neighbor queries on the collection of objects. This section gives an overview of this method. An LSH scheme is given as [5]:

Definition 1 *For a given similarity function $\omega(x, y)$ and family of hash functions $\mathcal{F}$*

operating on a universe of objects U such that $\omega : U^2 \rightarrow [0,1]$ *, a locality sensitive hash function is a distribution on* $\mathcal{F}$ *such that x,y* $\in U$ *and* $h \in \mathcal{F}$*:*

$$Pr_{h\in\mathcal{F}}[h(x) = h(y)] = \omega(x, y), \tag{2.2}$$

where $\mathcal{F}$ is called hash function family for given similarity function $\omega(x, y)$. The similarity functions for which hash function families we know are: Jaccard function, cosine similarity and L_p norms for $p = \{1,2\}$. Cosine similarity based LSH scheme proposed by Charikar [7] works as follows: for a collection of vectors in n dimension, a random hyperplane defined by a normal unit vector $\mathbf{r}$ is chosen from a n-dimensional Gaussian distribution to hash an input vector $\mathbf{u}$ into $h_\mathbf{r}$ such that $h_r(\mathbf{u}) = 1$ if $\mathbf{r}.\mathbf{u} \geq 0$ and 0 otherwise. Different choices of $\mathbf{r}$ defines different such hash functions and for two input vectors $\mathbf{u}$ and $\mathbf{v}$, this hashing scheme has the property that [7]:

$$Pr[h_r(\mathbf{u}) = h_r(\mathbf{v})] = 1 - \frac{\theta}{\pi}, \tag{2.3}$$

where θ is the angle between the vectors $\mathbf{u}$ and $\mathbf{v}$. since this hashing produces single bit for every hash function, this shows that the probability that two vector's signatures match in different bit locations is directly proportional to the cosine of angle between them. Henzinger [8] applied this technique on term vectors to web document domain. Algorithm 1 shows how to compute `simhash` of a document containing n words $t_{1,d}, t_{2,d}, \ldots, t_{n,d}$ with feature vector $\mathbf{d} = (w_{1,d},\ w_{2,d},\ \ldots\ w_{n,d})$ using a f-bit standard hash function θ. To compute a f-bit `simhash`, the intermediate step is to compute a f dimension weight vector $\mathbf{W_d}$, each of whose dimension is initialized to zero. The hash function θ is applied on each feature $t_{i,d}$ iteratively and $w_{i,d}$ is added or subtracted from each dimension of $\mathbf{W_d}$ depending on the bit values of hash $\theta(t_{i,d})$ as follows: if the j^{th} bit of hash is 1, then j^{th} dimension of $\mathbf{W_d}$ is incremented by $w_{i,d}$ else j^{th} dimension of $\mathbf{W_d}$ is decremented by $w_{j,d}$. When all n weights have been

Table I. An example computation of `simhash` on a document containing six words using four bit hash values.

Feature	Hash	weight				
$word_1$	0101	0.05	-0.05	+0.05	-0.05	+0.05
$word_2$	1101	0.02	+0.02	+0.02	-0.02	+0.02
$word_3$	0001	0.01	-0.01	-0.01	-0.01	+0.01
$word_4$	1110	0.03	+0.03	+0.03	+0.03	-0.03
$word_5$	0100	0.05	-0.05	+0.05	-0.05	-0.05
$word_6$	0011	0.09	-0.09	-0.09	+0.09	+0.09
$\sum weight$			-0.15	+0.05	-0.01	+0.09
`simhash`			0	1	0	1

added/subtracted based on the bits at specific index for each feature hash, then document `simhash` is computed as follows: j^{th} bit of `simhash` is 1 if j^{th} dimension of $\mathbf{W_d}$ is greater than zero and 0 otherwise. Mathematically, the j^{th} dimension of $\mathbf{W_d}$ can be given as:

$$W_{j,d} = \sum_{i=1}^{n} (2b_{i,j} - 1) w_{i,d}, \tag{2.4}$$

where $b_{i,j} = j^{th}$ bit of $\theta(t_{i,d})$ and $b_{i,j} \in \{0, 1\}$. Table I shows an example computation of `simhash` on a document with six words and using a four bit hash function. Manku et al. [6] proposed a scheme to find near-duplicates using `simhash` in which they create sorted tables of permuted fingerprints to find matching fingerprints within the hamming distance threshold. The idea behind their approach is as follows: Using multiple permutation schemes π_i, create copies of the existing fingerprints and store them in different sorted tables. Given a data set of 2^d f-bit truly random fingerprints, choose d' such that $d - d'$ is a small positive number and for a query fingerprint q, identify fingerprints in different tables such that they match $\pi_i(q)$ in d' most-significant bit positions. The matching fingerprints are then checked if they match $\pi_i(q)$ in at most h bit-positions. If $d - d'$ is small, the number of matching fingerprints is small

Algorithm 1 Calculate `simhash`

$W[f] = \{0\}$;
for $i = 1$ to n **do**
  $X \leftarrow \theta(t_i)$
  for $j = 1$ to f **do**
    if $X[j] = 1$ **then**
      $W[j] \leftarrow W[j] + w_i$
    else
      $W[j] \leftarrow W[j] - w_i$
    end if
  end for
end for
for $i = 1$ to f **do**
  if $W[i] > 0$ **then**
    $S[i] \leftarrow 1$
  else
    $S[i] \leftarrow 0$
  end if
end for
return S

but the number of tables needed is more and vice-versa. For instance, 8B 64-bit fingerprints require 1024K fingerprint matches in 4 tables, 5120 fingerprint matches in 10 tables and 1024 matches in 16 tables in worst-case. First matching fingerprint is found using a binary search and rest are matched with linear scan. Increasing the number of tables not always leads to improved performance due to increased number of binary search operations and additional query permutations which are more expensive than linear scan. Another drawback of this method is that in batch mode the near-neighbor search is performed on query fingerprints instead of existing data set which leads to sub-optimal performance as we show in later sections.

We have implemented this method and chose the number of tables needed and parameter d' based on the size of our data set and tune these parameters to most optimal performance for this algorithm.

CHAPTER III

MOTIVATION

Problem Definition: Given a collection of f-bit fingerprints $\mathcal{C}$, query fingerprint q and a maximum hamming distance h, find a near-duplicate fingerprint $q' \in \mathcal{C}$ such that:

$$H(q, q') \leq h. \tag{3.1}$$

When the existing fingerprints are stored on disk, instead of single query fingerprint a batch of fingerprints are expected to find near-duplicate neighbors. The hamming distance H between two f bit fingerprints $\mathbf{p} = (x_{1,p}, x_{2,p}, \ldots, x_{f,p})$ and $\mathbf{q} = (x_{1,q}, x_{2,q}, \ldots, x_{f,q})$ where $x_{i,j} \in \{0, 1\}$ is:

$$H(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{f} ||x_{i,p} - x_{i,q}||. \tag{3.2}$$

Hamming distance between two document fingerprints is computationally less expensive than cosine similarity between their feature vectors and fingerprints also require much lesser space to store. Table II compares 1000 64-bit fingerprints and feature vectors in terms of time it takes to compute all-pair hamming distance vs. cosine similarity and the space needed to store them. Hamming distance can be computed efficiently by just two operations: bit XOR between fingerprints and counting the number of set bits [19, 20] whereas for calculating cosine similarity we must perform n multiplications and $n - 1$ additions for two n dimensional vectors and is an $O(n)$ operation. Hamming distance also benefits from the small size of fingerprint which can be read into cache more number of times than the feature vectors.

Table II. Comparison between 64-bit fingerprints and word feature vectors while computing all-pair Hamming distance $H(x, y)$ and cosine similarity $\cos(x, y)$ on 1000 web pages and their storage space required.

Method	Time(s)	RAM
H(x, y)	0.04	8000
$\cos(x, y)$	1.914	2.7 x 10^7

A. Naive Solution

A naive solution to this problem would be to match fingerprint q with every other fingerprint in the collection $\mathcal{C}$ and check to see if we can find a q' which matches with it in at least $f - h$ bits. This is very expensive for large sized data sets and is not feasible for making fast cluster membership decisions. For example, with an existing data set of 70M fingerprints, a new arriving set of 10M fingerprints would approximately take 10 days to finish doing near-duplicate search based on the processing time quoted in Table II.

CHAPTER IV

PROBABILISTIC SIMHASH MATCHING

We propose a new method, the *Probabilistic Simhash Matching* (PSM) system which does not need to maintain multiple copies of data for near-duplicate search while still maintaining a good recall percentage. For a given query fingerprint, we explore different existing fingerprints based on the probability of finding a near-duplicate match and limit our number of attempts to achieve good query speed. We show experimentally that our probability estimation of near-duplicate fingerprint match works well by implementing this technique on our data set of 70M web documents collected from IRLbot [1] crawl.

A. Main Idea

Consider a table T that can accommodate all possible f-bit fingerprints contains 2^d ($d \leq f$) f-bit fingerprints in the sorted order. For uniformly distributed fingerprints, the expected distance between two fingerprints is 2^{f-d}. It is easy to see that for a query fingerprint q in the table if there exists a near-duplicate q' also in the table, it has to be different from q in at least one of the d most significant bit-positions. The general idea to find near-duplicate for a given query q and maximum allowed hamming distance h is: flip i ($0 \leq i \leq h$) bits in q in top d bit-positions iteratively in different combinations to generate modified queries q_m and for all fingerprints in T which match q_m in top d bit-positions, check if they differ with q_m in at most $h - i$ bits in least-significant $f - d$ bit-positions.

The procedure described above can locate all fingerprints which differ from q in at most h bit-positions. But generating modified queries is expensive for large d and even small h. The total possible modified queries M that can be generated for a given

f and h is:

$$M = \binom{f}{1} + \binom{f}{2} + \ldots + \binom{f}{h}. \tag{4.1}$$

For example, the total number of modified queries that can be generated for 70M 64-bit fingerprints for $h = 3$ is $= 2951$ queries. Alternatively, one can choose d' such that $d - d'$ is a small positive number and generate modified query fingerprints by flipping bits in top d' bit-positions. This will return an expected $2^{d'-d}$ matches per attempt which have to be checked to see if they differ with q in at most h bit-positions. Although this will lead to an overall more number of query fingerprints getting checked in worst-case, one can find a match while linear scanning the sorted data which is inherently faster than random memory lookups.

The procedure is still not suitable for real time clustering which needs fast cluster membership decisions since it generates a lot of modified query attempts as shown earlier. Ideally, one would want to optimize the number of modified queries getting generated by flipping bit-combinations and number of fingerprints getting checked with linear scanning. A large $d-d'$ wastes time checking large number of fingerprints with linear scanning but will require small number of random memory lookups and vice-versa. We therefore need to optimize the number and order of modified queries getting generated during the process.

The near-duplicate q' of fingerprint q is likely to flip only a few bits of it. The components of intermediate weight vector of a `simhash` are potentially useful in determining which bits to flip. Recall that while generating `simhash` q for document d, the j^{th} component $W_{j,d}$ of the intermediate weight vector $\mathbf{W_d}$ was given as:

$$W_{j,d} = \sum_{i=1}^{n} (2b_{i,j} - 1)w_{i,d}, \tag{4.2}$$

where $b_{i,j} = j^{th}$ bit of $\theta(t_{i,d})$, $j \in [1, f]$ and n is the total number of feature in the

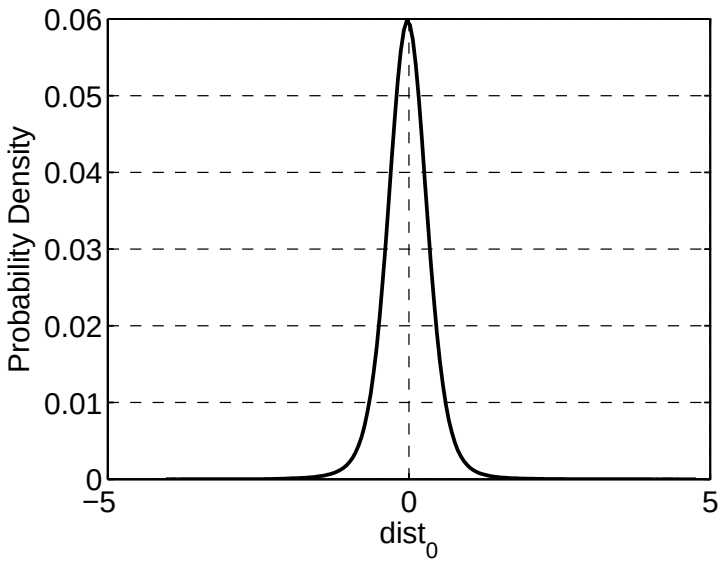


Fig. 1. Distribution of $dist_j$ calculated with 45M document pairs for $j = 0$.

document. For two near-duplicate documents d and d', the distance $dist_{j,d-d'}$ between their weight vectors $\mathbf{W_d}$ and $\mathbf{W_{d'}}$ in j^{th} dimension can given:

$$\begin{aligned} W_{j,d'} &= W_{j,d} + dist_{j,d-d'} \\ &= W_{j,d} - \sum_{i=1}^{a}(2b_{i,j} - 1)w_{i,d} + \sum_{i=1}^{b}(2b_{i,j} - 1)w_{i,d'}, \end{aligned} \tag{4.3}$$

where a is the number of features which are present in d but not in d', b is the number of features present in d' but not in d and $dist_{j,d-d'}$ is the distance between $\mathbf{W_d}$ and $\mathbf{W_{d'}}$ in the j^{th} dimension. The probability that q' flips the j^{th} bit of q is directly dependent on how far away $dist_{j,d-d'}$ is from $W_{j,d}$ on the number line. We can prove this by plotting the distribution of $dist_{j,d-d'}$ which intuitively should follow a normal distribution with mean 0. Figure 1 shows the plot of $dist_0$ for a random sample of 45M document pairs which proves the correctness of our assumption. The distribution

of *dist* for other bit-positions follow similar curve. Therefore, the probability that fingerprint q' flips the j^{th} bit b_j of q can be estimated as:

$$P(b_j(f) \neq b_j(f')) = \begin{cases} P(X \geq -W_{j,d}) & \mathrm{W}_{j,d} \leq 0 \\ P(X \leq -W_{j,d}) & \mathrm{W}_{j,d} > 0 \end{cases} \tag{4.4}$$

which is nothing but the area between under the probability distribution curve of a normal distribution with mean 0 and we can say that the probability of a bit-flip is greater for a bit whose weight $W_{j,d}$ is closer to 0. Mathematically, the probability of bit-flip for j^{th} bit ($j \in [1, f]$) can be estimated as:

$$Pr(b_j(f) \neq b_j(f')) = 1 - \Big| \frac{W_{j,d}}{|\mathbf{W_d}|} \Big|, \tag{4.5}$$

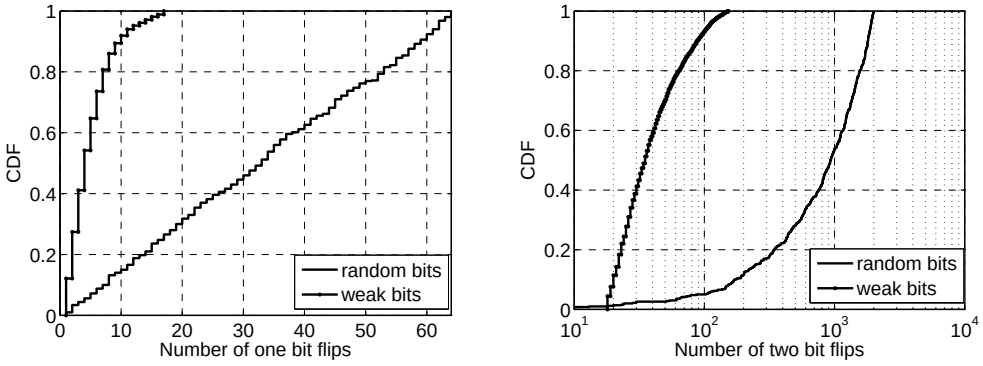
where we divide every weight component $W_{j,d}$ by size of the vector $|\mathbf{W_d}|$ to get a value between 0 and 1. We term the top-k most probable bits of a fingerprint to get flipped by a near-duplicate document fingerprint as the k *weakest* bits of the fingerprint.

Assuming that bits of a fingerprint f are independent, the probability of a combination of h bits represented as a bit-vector $\mathbf{B} = (b_{p_1}, b_{p_2}, \ldots, b_{p_h})$ where $p_i \in [1, f]$ getting flipped by a fingerprint f' can be given as:

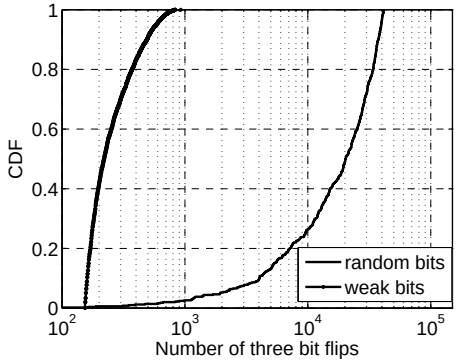
$$Pr(\mathbf{B}(f) \neq \mathbf{B}(f')) = \prod_{i=1}^{h} Pr(b_{p_i}(f') \neq b_{p_i}(f)). \tag{4.6}$$

We will see in the experiments that assumption of bit independence works well and is an acceptable hypothesis. The position vector of a bit-combination of size l is given as $\mathbf{B_p} = (p_1, p_2, \ldots, p_l)$ where $p_i \in [1, f]$.

Figure 2 plots the distribution of the number of h bit flip attempts needed between near-duplicate fingerprint pairs which differ in h bit-positions. The graph was generated by random sampling of 8M fingerprint pairs. It is apparent that concept of *weak* bit flips works well and has an advantage over random flipping of bits.



(a) $h = 1$ (b) $h = 2$



(c) $h = 3$

Fig. 2. Comparison of the number of bit-flip attempts needed between a near-duplicate fingerprint pair by flipping *weak* bits and random bits.

B. Bit-Combination Generation

We now present an algorithm to generate top-k weakest bit-combinations $\mathbf{B}$ of a fingerprint of maximum size h. Assuming that we flip bits only in the leading p ($p \in [1, f]$) bit positions of a f-bit fingerprint and given the `simhash` weight vector $\mathbf{W}$, finding top-k weakest individual bits of a fingerprint in sorted order of probabilities can be done efficiently in $O(p \log k)$ time. But from equation 4.1, we know that finding *weak* bit-combinations is expensive because there are a lot of options to choose from. Since the probability of a bit combination is a product of its individual probabilities

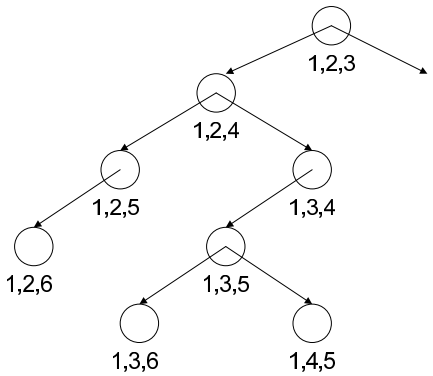


Fig. 3. An example computation of function next for three bits given a bit-combination index vector {1,2,3} which represent first three bits in the list of top-k weakest bits.

which is a number between 0 and 1, we know that a bit combination of size i will always have a probability lower than all its components of size ¡ i. And we can easily say that top-k bit-combinations of maximum size h will only contain top-k *weakest* bits of the fingerprint as their individual component. Effectively to find the maximum number of size i bit-combinations in the top-k list, one can solve this equation for $\min(k_i)$:

$$\binom{k_i}{1} + \binom{k_i}{2} + \binom{k_i}{3} + \ldots + \binom{k_i}{i} \geq k. \tag{4.7}$$

Then in a list of top-k bit-combinations, maximum number of size i bit-combinations is given as $\max(L_1, L_2)$ where:

$$L_1 = k - \binom{k_i}{1} - \binom{k_i}{2} - \ldots - \binom{k_i}{i-1}, \tag{4.8}$$

and

$$L_2 = \binom{k_i - 1}{i}, \tag{4.9}$$

An efficient way to generate top-k weakest bit combinations of a maximum size h is:

- Create a max-heap M and initialize it with highest probability bit-combinations

of all sizes $\leq h$ with bit-combination position vector and its probability being the key,value pair. Note that highest probability bit-combination of size i is the first i bits in the top-k list of weakest bits.

- Perform *Extract* on M and for any bit-combination of size i getting extracted, replace it with next size i bit-combination candidates of highest probability.
- Repeat *Extract* k times.

If we represent a bit-combination by the index of its individual bits (in sorted order) in top-k list of *weakest* bits, we know that for a combination of size two represented by (i, j) the next bit-combination candidate with highest probability is either $(i + 1, j)$ or $(i, j + 1)$. Generalizing this for different sized bit combinations, for a bit-combination of size l represented as $(i_1, i_2, \ldots, i_l)$ the next set of candidates with highest probabilities can be generated by adding 1 to individual components one at a time and leaving out candidates for which $i_j = i_k$ for any $j, k \in [1, l]$. However, this can lead to a maximum of h bit-combinations getting generated for every *Extract* and certain combinations getting repeated multiple times. For example, (1,3) can be generated while extracting (0,3) and (1,2). To avoid generating large values and to restrict repeated values, function *next* generates a maximum of two candidates given a bit-combination of any size and always makes sure that generated combinations are of highest probabilities. Please note that instead of bit-combination position vector which contains position of individual bits in the fingerprint, function *next* takes as input the bit-combination index vector whose components are the index of its individual bits in the list of top-k weakest bits. For example, the highest probability bit-combination of size 2 will be represented by an index vector of (1,2), highest probability bit-combination of size 3 will be represented by an index vector of (1,2,3), and so on. We define the working of function *next*, given a bit-combination

Algorithm 2 Generate highest probability bit-combinations using function next

Input: B = $(p_1, p_2, \ldots, p_l)$ bit-combination position vector of size l

Output: B1,B2 = position vectors of size l

$B1 \leftarrow (p_1, p_2, \ldots, p_l + 1)$

$B2 \leftarrow null$

$j \leftarrow 1$

while $p_{l-j} - p_{l-j+1} \leq 2$ and $j < l$ **do**

  if p_{l-j} - $p_{l-j+1} = 2$ **then**

    $B2 \leftarrow (p_1, p_2, \ldots, p_{l-j-1}, p_{l-j} + 1, p_{l-j+1}, \ldots, p_l)$

    return

  end if

  $j \leftarrow j + 1$

end while

index vector of size l in algorithm 2. An example output of function *next* for size three bit-combinations is shown in figure 3. The maximum size of heap M will never go beyond $k+h-1$ and therefore, total time complexity of this method once the tree using *next* function has been created is: $O((p+k)\log(k+h))$.

CHAPTER V

IMPLEMENTING PSM

In this section we present our algorithm for near-duplicate search in both online and batch mode using the ideas presented in previous section.

A. Algorithm for Online Queries

Online mode is characterized by a data set of existing fingerprints stored in memory and where near-duplicates for the new arriving fingerprints are to be found. These new fingerprints are later added to the existing data set in case of a crawler. We first build a sorted table T of existing f-bit fingerprints and also determine a global value p which we define as the most-significant p bits of the fingerprints from which different *weak* bit-combinations are flipped for generating modified queries. Further, a hash table is created with 2^p entries such that each entry points to a fingerprint which is the first instance of p most-significant bits in Table T. Given a new arriving query fingerprint q and a maximum hamming distance h, to find its near-duplicate :

- Identify the weakest bit-combination to flip in fingerprint q using a max-heap as described in previous section.
- Using the bit-combination generated from previous step, generate the modified query q' by flipping the bits from bit-combination position vector. Identify all such fingerprints in T which match q' in most-significant p bits.
- For each of the fingerprint match identified, check if it differs from q in at most h bit -positions.
- If no match is found, push the next bit-combinations of highest probability

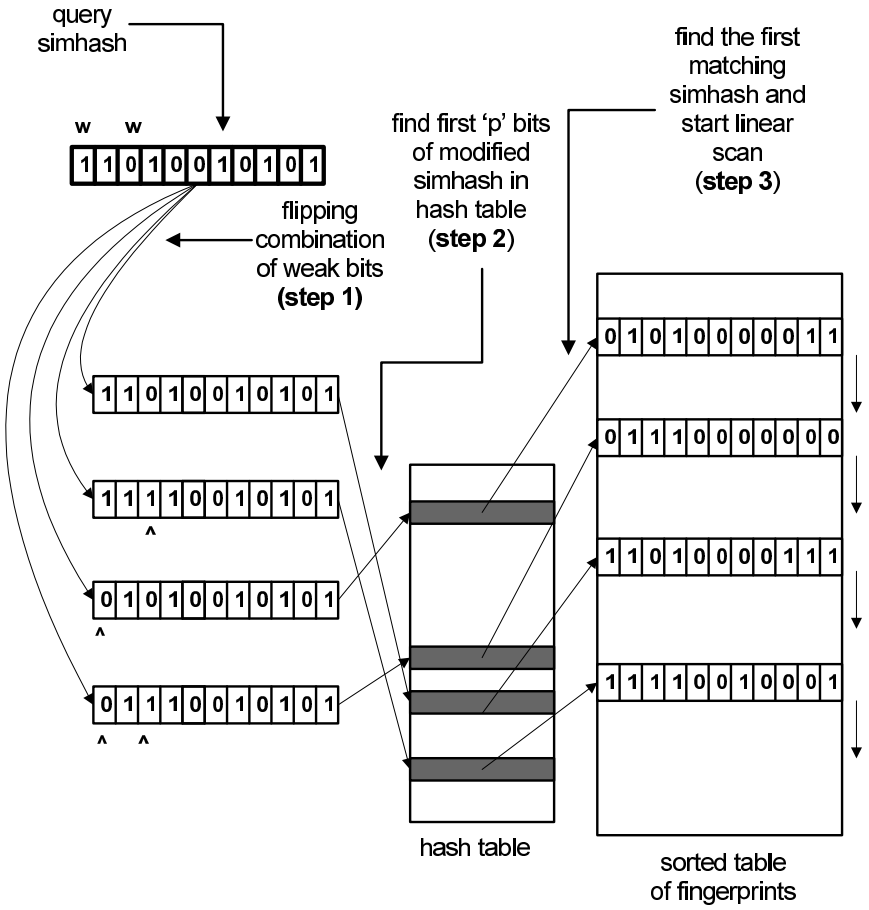


Fig. 4. Working of PSM for $f = 10$ and $p = 4$. Weak bits are flipped to generate modified queries and then matched in sorted table followed by linear scan.

generated with function *next* and repeat step one k times.

Figure 4 explains this design in different steps. Identification of first matching fingerprints is done using the hash table and rest ones are matched with linear scanning. Different choices of p and k can lead to difference in query times. Ideally, the design goals are to: minimize query time and minimize the space requirement. A large value of p can avoid checking too many fingerprints with linear scan but will requires extra number of modified queries k for maintaining a good recall. The size of the hash table

required is also dependent on p and needs 2^p entries.

B. Algorithm for Batch Queries

Batch mode is characterized by a batch of new arriving query fingerprints Q stored in memory and where number of existing fingerprints are large enough to be stored in a file F on disk. In this case, the task of near-duplicate search will be performed by chunks of file F getting read sequentially into memory and hamming distance problem solved for queries Q on each chunk separately. The query batch Q after processing is appended to the file F in case of a web crawl dataset. The query chunk is sorted before appending and hence file F is a collection of sorted chunk of fingerprints of size equal to the size of query batch.

We stop reading chunks from file F as soon as we are done finding one near-duplicate for all fingerprints in the query batch. Since we have to read the whole file in worst-case every time a new batch arrives, ideally we would want to maximize the batch size while keeping the query time minimum. In a multi-core system, the task could be divided into multiple threads where each thread will read subsequent chunks and process near-duplicate search for batch query Q separately. After processing a chunk, threads can share the information about query fingerprints for which near-duplicate has been found such that those fingerprints could be removed in the processing of the next chunks. An important difference in batch mode is that finding top-k weakest bit-combinations for query fingerprints should be done as a separate preprocessing step which is unlike online mode where it is done for every fingerprint during the near-duplicate search. This is to avoid wasting time in generating weak bit-combinations for fingerprints every time a new chunk is loaded into memory.

CHAPTER VI

PERFORMANCE MODELING

We describe previous approach used by Manku et al. [6], evaluate its performance and model our own approach for both online and batch mode.

A. Modeling Previous Method

For finding near duplicates at most h hamming distance away, Manku et al's [6] algorithm build multiple tables: $T_1, T_2, ..., T_t$ of fingerprints by applying different permutations given as: $\pi_1, \pi_2, ..., \pi_t$ over the f bit-positions. Permutation corresponds to dividing the fingerprint into multiple blocks and making the bits in a certain number of blocks as leading bits of the fingerprint. With each table T_i is stored an integer p_i such that for any query fingerprint q: all the fingerprints which match the permuted query fingerprint $\pi_i(q)$ in the top p_i bit-positions are first identified and then checked to see if they differ with $\pi_i(q)$ in at most h bit-positions. The identification of the first permuted fingerprint is done using binary search and hamming distance with other fingerprints is computed as liner scan. The value for number of tables(t) and permutation(p_i) is selected keeping in mind the fact that a large p_i would increase the space requirement while reducing the number of fingerprints being checked and vice-versa.

For f bit truly random fingerprints, maximum allowed hamming distance = h and $n = 2^d$ existing fingerprints, assume that:

The fingerprint is split into z blocks of size $\Delta_1, \Delta_2, \ldots, \Delta_z$ such that:

$$\Delta_1 + \Delta_2 + \ldots + \Delta_z = f. \tag{6.1}$$

For permutation π_i, top p_i bits are aligned with top x blocks such that:

$$\Delta_1 + \Delta_2 + \ldots + \Delta_x = p_i. \tag{6.2}$$

Further assume that for simplicity, fingerprints are divided into blocks of similar size and therefore:

$$p_i = \frac{xf}{z}. \tag{6.3}$$

Near-duplicates which are at most h hamming distance also require:

$$z - x \geq h. \tag{6.4}$$

The absolute minimum number of tables t needed is dependent on h and is determined as t_{min}:

$$t_{min} = h + 1. \tag{6.5}$$

Theorem 1 *For a given query fingerprint, the expected number of fingerprints that have to be checked is given as*:

$$\alpha(f, d, z, x) = L2^{d - \frac{f}{z/x}}, \tag{6.6}$$

where L is the number of tables and upper-bound on x is given from equation 6.4.

The theorem shows that $\alpha(f, d, z, x)$ is a product of two elements: the number of tables created and the expected number of fingerprints in each table that will be checked. It is important how z and x are chosen because a small z/x can exponentially reduces the number of fingerprints getting checked. The minimum x is 1 and therefore maximum z/x is: $1 + h$. To further reduce z/x we need to increment both z and x while maintaining the upper bound on x from equation 6.4. Incrementing z and x also leads to increasing left term of the equation and hence the next best values of z/x are: $(2 + h)/2$,$(3 + h)/3$ and so on. Increasing z and x also leads to a blowup in

space requirement with number of tables L_T given as:

$$L_T = \binom{z}{x}. \tag{6.7}$$

Reducing z/x below f/d will lead to empty table lookups and higher query time because of wasted binary searches in many tables. Also note that if z/x is less than f/d, then for each table binary search is given as $O(d)$ instead of $O(p_i)$. For $f = 64$, $h = 3$, $d = 34$ and an optimal $z = 6$ and $x = 3$, the number of tables needed is 20 and expected number of fingerprints that will be checked in each table is $2^{d-xf/z} = 8$. Alternatively, one could choose a sub-optimal z and x and reduce the space requirement with lesser number of tables at the cost of extra number of fingerprints getting checked. When the search is for finding all near-duplicate matches, the number of tables which are checked for query matches is given as:

$$L = L_T. \tag{6.8}$$

and when the search is for only one near-duplicate match, the number of tables checked for query match is given as:

$$1 \leq L \leq L_T. \tag{6.9}$$

The number of tables that are checked for finding one match is less than the total number of tables since the search stops after first near-duplicate match.

1. Online Mode

We now discuss the performance of this algorithm in online mode. The running time is decided by the time it takes to find at least one near-duplicate of the new arriving fingerprint. The formulation is done assuming the existing data set of fingerprints F is stored in main-memory and multiple tables of permuted fingerprints are already

sorted. The time it takes to search for a near-duplicate of the new arriving fingerprint q in one table is given as:

$$\rho_1 = fc_1 + \frac{xf}{z}c_2 + 2^{d-\frac{f}{z/x}}c_3, \tag{6.10}$$

where x, z, d and f are as defined earlier and c_1, c_2 and c_3 are computational constants. First term of the above equation is the time it takes to permute an f bit fingerprint in any order, second term is the time needed to perform binary search for a fingerprint in one table and the third term is the time spent linear scanning on average in one table. Also note that if z/x is less than f/d, then binary search is given as $O(d)$. Total time to perform near-duplicate search for one document in all the tables is given as:

$$\rho = L\rho_1. \tag{6.11}$$

The total number of fingerprints that can be stored in memory is limited by RAM size s (in bytes) such that:

$$s \geq \binom{z}{x} 2^d \frac{f}{8}. \tag{6.12}$$

Assuming for a RAM size of $s = 16$ GB, using the absolute minimum number of h+1 = 4 tables needed for $h = 3$ and ignoring other data structures needed to be kept in main-memory, for $f = 64$ bit fingerprints this allows for a maximum $d = 29$ or 2^{29} fingerprints that can be processed in online mode with an expected number of fingerprints getting checked per query = 2^{d-16}.

2. Batch Mode

Batch mode is run on a batch of new arriving query fingerprints Q which are stored in main-memory and the dataset of existing fingerprints F stored on disk. After processing, the batch is appended to the existing dataset of fingerprints in case of

a crawler. The technique for near-duplicate detection here is different from online mode since tables are built for query batch Q and near-duplicate search runs from $F \rightarrow Q$. Important thing to note is that for a query fingerprint $q \in Q$, near-duplicate search can stop as soon as one match is found in online mode but the same is not true for a fingerprint $q' \in F$ in batch mode because we are finding potential queries for existing answers, so we should keep searching for near-duplicates even if there is one fingerprint in Q which has not been matched.

Assume that file F on disk has n fingerprints and near-duplicates of a batch of m query fingerprints in main-memory from file Q is to be found. F is read in similar chunks of size l fingerprints/chunk and therefore, the time for disk I/O is given as:

$$\rho_1 = \delta_1 + \frac{nf}{\delta_2}, \tag{6.13}$$

where δ_1 is the disk seek-time in seconds and δ_2 is the read speed in bits/sec. This is computed assuming the file is read sequentially with only one disk seek required. The time taken for near-duplicate search for one fingerprint from F on one table is given as:

$$\rho_2 = (n+m)fc_1 + m\log mc_2 + n\frac{xf}{z}c_3 + nm2^{-xf/z}c_4, \tag{6.14}$$

where x, z and f are as defined earlier and c_1, c_2, c_3 and c_4 are computational constants. First term of the equation is the time needed to permute $n+m$ fingerprints, second term is time needed to sort a table before search starts, third term is the time required for binary search of n fingerprints, and the last term is time needed to linear scan n fingerprints. Also note that if z/x is less than f/d, then binary search is given as $O(d)$. Total time taken when disk I/O is done in parallel with CPU processing is given as:

$$\rho = \max(L\rho_1, \rho_2). \tag{6.15}$$

B. Modeling Our Method

To find near-duplicate for a f bit fingerprint which is at most h hamming distance away, we first define p as the leading p bits of the fingerprint from which different *weak* bit-combinations to flip are generated. A l sized weak bit-combination $\mathbf{B}$ is represented by its position vector $\mathbf{B_p} = (p_1, p_2, \ldots, p_l)$ where $p_i \in [1, p]$ and $1 \leq i \leq l$. Note that we are assuming that bit-position is calculated from its most-significant bit. For a given query fingerprint q and a given *weak* bit-combination B_p of size i, we iteratively flip bits at positions given by B_p to generate q' and all such fingerprints in the existing dataset which match it in most-significant p bits are checked if they differ it in at most h bits with q.

1. Online Mode

Existing fingerprints F are assumed stored in the memory in a sorted table and near-duplicate of a new arriving fingerprint q is to be found. The number of fingerprints in existing dataset is $n = 2^d$ and the RAM size is assumed s. We create an index of the sorted table such that a pointer to the fingerprint with first instance of unique leading p is stored. This accounts for a index size of 2^p entries. The formulation is done assuming that index on the existing dataset is already created. The time it takes to find a near-duplicate of the new arrived document q is given as:

$$\rho = (p + k) \log (k + h) c_1 + k 2^{d-p} c_2, \tag{6.16}$$

where k is the maximum number of weakest bit-combinations to be flipped and d and p are as defined earlier and c_1,c_2 are computational constants. The first term determines the time that is needed to find top k *weakest* bit-combinations of the fingerprint and the second term is the time needed to perform linear scan on the table. It is important

how we choose p and k which determines the number of fingerprints getting checked. Increasing p exponentially reduces the time spent with linear scanning but increases the index size and time spent in finding the weakest bit-combinations. Increasing p also requires increasing k for maintaining a particular recall and hence it is important to find the rate of increase of k with p. If k increase at a rate much slower than exponential, then it is always better to have $p = d$ to achieve the best performance.

The maximum number of fingerprints that can be processed is limited by RAM size s such that:

$$s \geq (2^d + 2^p)\frac{f}{8}. \tag{6.17}$$

Assuming for a RAM size of 16 GB, maximum size of index that is possible with $p = d$ and ignoring other data structures needed to be stored in memory, this allows for a maximum data size of $d = 30$ or 2^{30} fingerprints that can be processed in online mode which is double the number of fingerprints that could be processed with previous method and an expected fingerprints getting checked per query = k.

2. Batch Mode

Batch mode is run on a batch of new arriving query fingerprints Q kept in memory while existing fingerprints in a file F are stored on disk. Queries are received in a batch of size m fingerprints/batch and the number of existing fingerprints can be assumed n. After the processing is over, query batch is sorted and appended to F. Thus existing fingerprints are pre-sorted in chunks of size equal to the query batch. Important thing to note here is that we run our batch process from $Q \rightarrow F$ and try to find near-duplicate matches for new queries unlike previous method which tries to find query matches for existing fingerprints. Hence, we are not constrained to run the near-duplicate search after the first match for a query fingerprint is found.

The time it takes to read fingerprints from disk in chunks of m fingerprints/chunk is given as:

$$\rho_1 = \delta_1 + \frac{nf}{\delta_2}, \tag{6.18}$$

where δ_1 is the disk seek-time in seconds and δ_2 is the read speed in bits/sec. This is computed assuming the file is read sequentially with only one disk seek.

Processing time is given as:

$$\rho_2 = m(p+k)\log(k+h)c_1 + nm2^{-p}kc_2 + m\log mc_3, \tag{6.19}$$

where f,k and p are as defined earlier and c_1, c_2 are computational constants. First term of the equation is the time needed to find top k weak bit-positions of m fingerprints, second term is the time needed to search near-duplicates for m fingerprints in n/m chunks and the third term is the time needed to sort m fingerprints before appending it to the file. Total time taken when disk I/O is done parallel with CPU processing is given as:

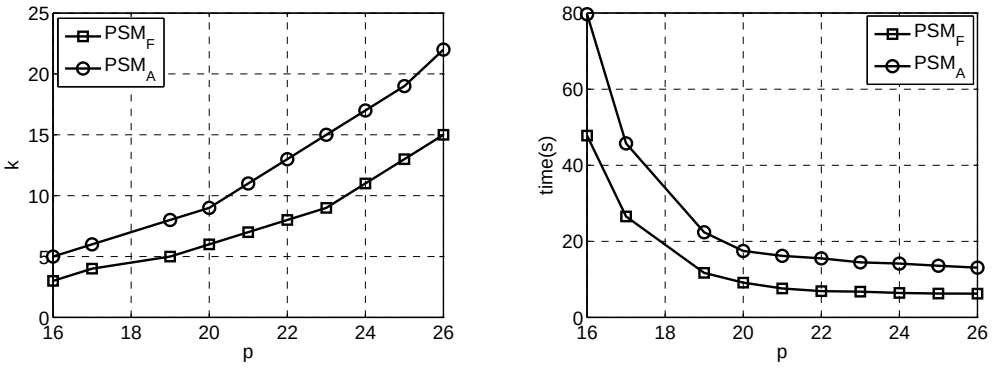
$$\rho = \max(\rho_1, \rho_2). \tag{6.20}$$

3. Linear Scan vs Bit-flip

We evaluate the effectiveness of linear scan vs bit-combination flips in terms of query time and space requirement for near-duplicate search. Expected number of fingerprints getting checked at any p,k can be given as X:

$$X = k2^{d-p}. \tag{6.21}$$

For a fixed dataset size given by 2^d, incrementing p by 1 leads to halving the number of queries getting checked by linear scanning but to maintain a certain recall, number of bit-combination flips k also increases. Hence incrementing p is useful only if required



(a) k vs p (b) time vs p

Fig. 5. Efficiency of bit-flip over linear scan.

k does not double with every step of incrementing p by 1. Figure 5(a) shows the required number of bit-combination flips k for both PSM_A and PSM_F is almost sub-exponential with p for $d = 26$ and hence we see a dramatic improvement in query time with increasing p in figure 5(b). Since, the reduction in linear scan is exponential, the improvement in performance gets smaller as p approaches d.

CHAPTER VII

EXPERIMENTS

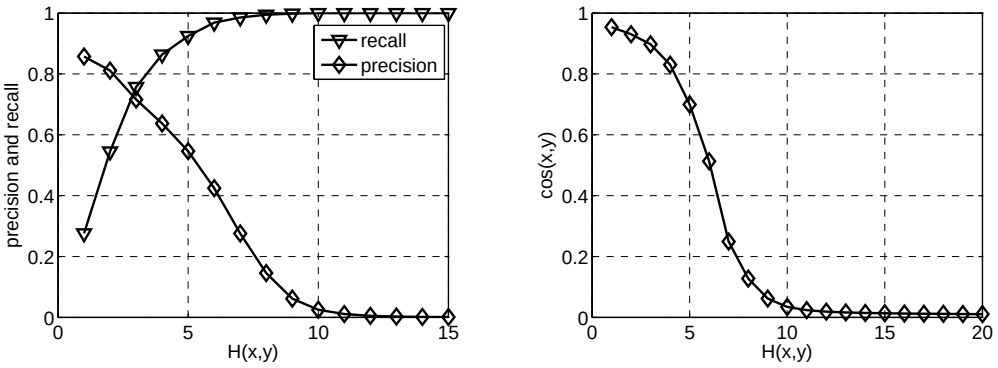
We first show that `simhash` is useful for finding near duplicates in our web document data set. We found that $f = 64$-bit `simhash` with a maximum hamming distance of $h = 3$ is good enough for finding near-duplicates with high precision and recall. This section also presents the evaluation results of our method in comparison with previous method on the web document data set.

A. Dataset

All our experiments were done on a small subset of the web document data set downloaded using the IRLbot [1] crawler containing 100M web pages. We removed all such pages which were of size less than 5 Kb or had no URL present or had an exact duplicate also present in the data set identified using a standard hash function. We also removed pages which contained non-English words and this returned us a total of 70M pages after pruning. In the first step of processing, we parsed the html documents and created feature vectors for word TF-IDF after stemming (using standard porter stemming algorithm) and stop-word removal. For calculating `simhash` fingerprints, we used the 64-bit murmur hash function.

B. Near-Duplicate Search Using Simhash

We randomly sampled 100M document pairs from our data set and calculated cosine similarity between their document feature vectors and hamming distance between their `simhash` fingerprints. For our experiments, we tagged a document pair x,y as similar using their cosine similarity $\cos(x, y)$ and based on the following definition of



(a) Precision/Recall vs h (b) cosine similarity vs h

Fig. 6. `Simhash` performance at various hamming distance.

similarity $\omega(x, y)$:

$$\omega(x,y) = \begin{cases} 1 & \text{if } \cos(x,y) \geq \theta_0 \\ 0 & \text{if } \cos(x,y) < \theta_0 \end{cases} \tag{7.1}$$

where $\theta_0 =$ minimum cosine similarity for pages to be similar. We used a θ_0 value of 0.9 for our experiments. To compute precision and recall, we partitioned the document pairs based on the hamming distance between their respective fingerprints. Precision at hamming distance h is defined as the fraction of document pairs which are found similar with hamming distance at most h.

Precision at hamming distance h is written as P_h:

$$P_h = \frac{\sum_{i=1}^{h} n_{s,i}}{\sum_{i=1}^{h} n_i}, \tag{7.2}$$

and Recall at hamming distance h is calculated as R_h: the fraction of the total number of similar document pairs that are found with hamming distance at most h.

$$R_h = \frac{\sum_{i=1}^{h} n_{s,i}}{T_s}, \tag{7.3}$$

where $\sum_{i=1}^{h} n_{s,i}$ is the total number of similar documents found with hamming distance

at most h, $\sum_{i=1}^{h} n_i$ is the total number of document pairs found with hamming distance at most h and T_s is the total number of similar document pairs in the data set. Figure 6(a) shows the plots for precision and recall calculated for our data set for varying hamming distance. We can see notice the low false-positive and high false-negative rate at smaller hamming distances. Therefore, $h = 3$ would be a sensible choice for balancing the trade-off between precision and recall.

In Figure 6(b), the expected cosine similarity $E[\cos(x, y)]$ between document pairs at different hamming distance is shown. To calculate this, we partitioned document pairs based on the hamming distance between their fingerprints and then computed the average cosine similarity of documents at every hamming distance. We can clearly see the 0.95 expected cosine similarity for document pairs at hamming distance 1, 0.93 for documents at hamming distance 2 and 0.89 for documents at hamming distance 3. Clearly, more similar documents are closer in hamming space than less similar ones.

C. Implementation Details

We implemented our method and the method proposed by Manku et al [6] in C++ programming language. We tested both the methods in online and batch mode for recall, query time and space requirement. Our implementation of Manku et al's [6] method follows the idea presented in their paper as we could not find an existing implementation. An ideal near-duplicate detection algorithm should take less time with less space and be able to identify all near-duplicates in the data set. The recall for our experiments is defined differently for PSM_A and PSM_F. Given a fingerprint data set D with N existing fingerprints and a new arriving M query fingerprints, let L be the total number of query fingerprints which have at least one near-duplicate

match present in D. Then for a near-duplicate detection technique which finds one match for F query fingerprints, the recall is given as R_{PSM_F}:

$$R_{PSM_F} = \frac{F}{L}. \tag{7.4}$$

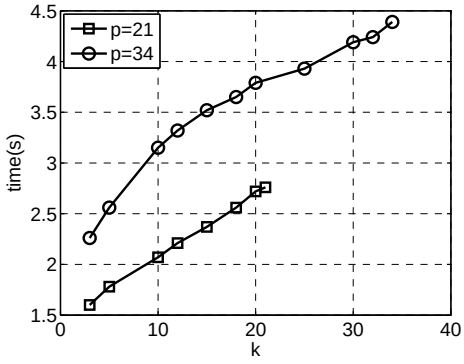
and let Z be the total number of near-duplicate matches for all query fingerprints M, then for a near-duplicate detection technique which finds $X \subseteq Z$ matches, recall R_{PSM_A} is given as:

$$R_{PSM_A} = \frac{X}{Z}. \tag{7.5}$$

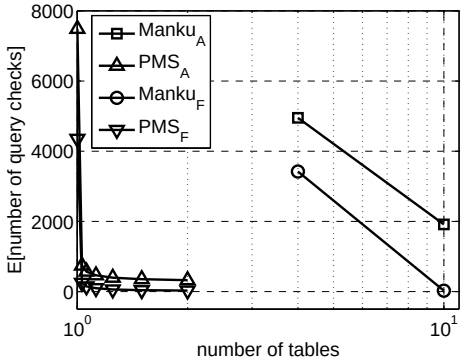
An ideal search algorithm should have a recall of 1.0. In all our experiments we target to achieve a recall of 0.95 with our method unless otherwise specified. Since, it is impossible to compute all-pair hamming distance between fingerprints for our large data set, we take the number of near-duplicates found using Manku et al's method [6] as the ground truth for recall computations. For each fingerprint and its near-duplicate, the hamming distance should be at least 1 and at most 3. For comparing our results with [6], we use: $Manku - A$ where search finds all near-duplicate documents and $Manku - F$ where search stops after finding the first near-duplicate document. For each experiment, we divided the data set of 70M fingerprints by random sampling into two parts: 10M fingerprints to be used as query fingerprints and 60M fingerprints to be used as the existing data set. We then try to find at least one match for the query fingerprints in the existing data set.

D. Hardware

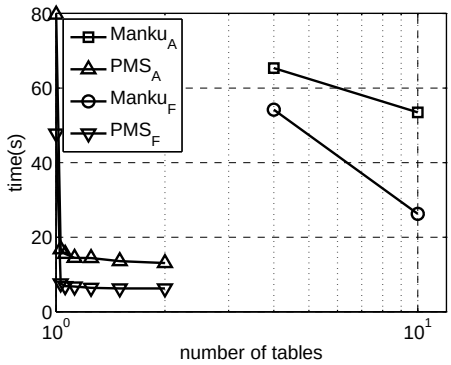
We used an AMD Phenom(tm) II 2.8 GHz six-core desktop machine with 3MB L2 Cache for all our experiments. The machine runs Windows Server 2008 R2 with 16 GB of RAM support and 5 TB of disk space available.



(a) preprocessing time



(b) # of query matches



(c) query time

Fig. 7. (a)The preprocessing time with increasing k at $p = 21$ and $p = 34$ (which is needed when dataset size becomes $\approx$ 8B). (b)The # of query matches with increasing number of tables for $d = 26$. (c)The time in seconds with increasing number of tables.

E. Preprocessing

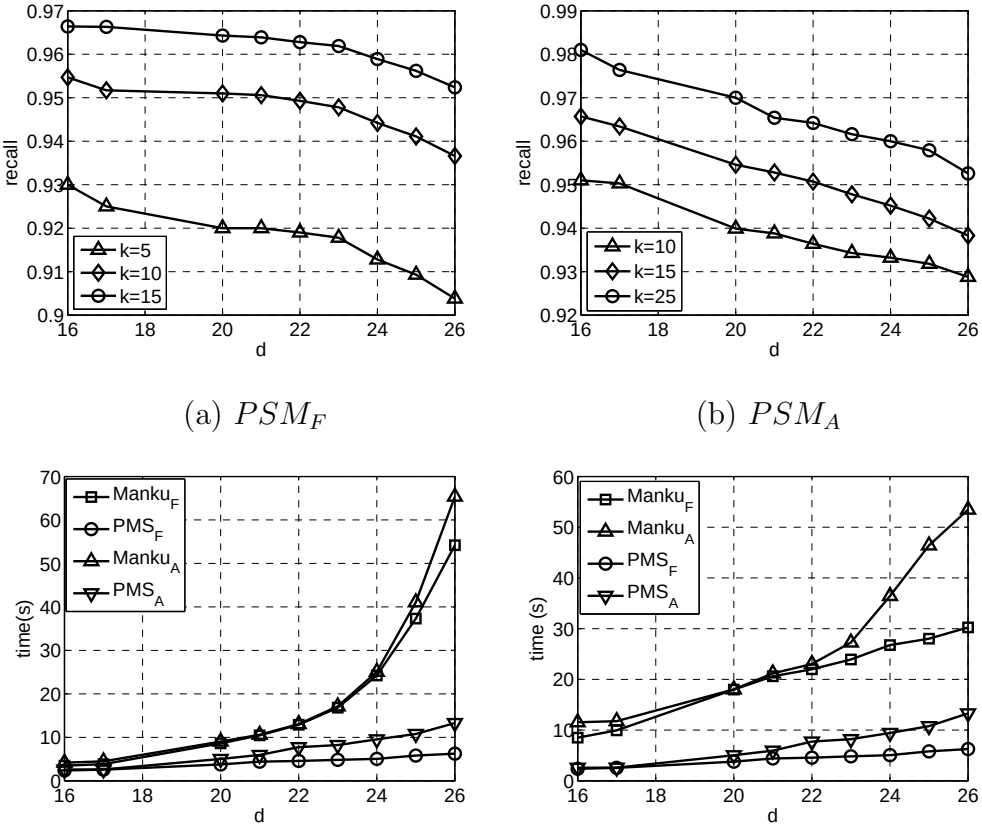
The process of finding top-k weakest bit-combinations is already discussed in section 4. The time it takes to find top-k weakest bit-combinations depends on the number of leading bits from which the bit-combinations are to be found p and the number of bit-combinations to be found k. Figure 7(a) shows the expected preprocessing time for 10M fingerprints for different values of p and k. Note that the preprocessing step is needed before near-duplicate search starts for queries in batch mode whereas for online mode, preprocessing is done at the run-time. To make preprocessing more efficient, we took the log of individual bit weights and solved the problem in a similar way using a max-heap for sum of numbers instead of their product.

F. Online Mode

Our method is very efficient in terms of number of tables required and query time while achieving a maximum of 0.95 recall. Figure 7(b) compares the expected number of fingerprints getting checked for our method and Manku et al's method [6] with increasing number of tables. In our method, hash-table size increases with increasing p and can grow to a maximum of 2^d entries. Therefore, the maximum number of tables in our method can never go beyond 2 for any size of the data set. Please note that number of tables t in our method is computed with equation:

$$t = 1 + \frac{1}{2^{d-p}}, \tag{7.6}$$

where 2^d is the number of fingerprints in the existing dataset. Maximum value of p that we use is d and hance number of tables can never go beyond 2. Increasing p requires more number of bit-combinations k to be flipped for maintaining a particular recall but leads to an overall lesser number of fingerprints getting checked. Manku et



(a) PSM_F (b) PSM_A

(c) Memory = $4|T|$ (d) Memory = $10|T|$

Fig. 8. (a)Recall of PSM_F at increasing size of the dataset with different values of k. (b)Recall of PSM_A at increasing size of the dataset with different values of k. (c)Time efficiency with varying dataset size at constant memory size of $4|T|$. (d)Time efficiency with varying dataset size at constant memory size of $10|T|$.

al's method [6] method can lead to marginally lesser number of fingerprints getting checked by using 5 times more space than our method. However in figure 7(c), which compares the time taken for near-duplicate search between our method and Manku et al's method [6] method we can see that this does not guarantee a lesser query time as our method can perform at least 3.5 times better than their method for all near-duplicate search. With increasing number of tables in [6], the cost of binary search in extra tables for each query fingerprint offsets the reduction in cost with reduced hamming distance computations to an extent which is not the case with our system.

The results show that our method is significantly better than previous method in terms of space usage and query time both with 0.95 recall. This is much suitable for real-time systems which are memory constrained and need faster performance but are ready to sacrifice a small recall percentage.

Figure 8(a) and 8(b) shows the change in recall with increasing dataset size. We can see that for PSM_F decreases much slower than PSM_A. For a recall of 0.95, PSM_A needs $k = 23$ and PSM_F needs $k = 15$. Figure 8(c) and 8(d) shows the relationship between increasing dataset size and query time at constant memory size of $4|T|$ and $10|T|$. Each dataset is built by random sampling of a subset of the existing fingerprints. The value of p was set as $p = d$ for every subset and hence the total number of tables required by our method is 2 at all times. The recall is maintained at 0.95. The results show that at constant memory size, the increase in query time with our method is much lower than the previous method. Also note that in previous method, time required is more with $10|T|$ as compared to $4|T|$ until $d = 25$ because of the increased cost of binary searches with empty table lookups.

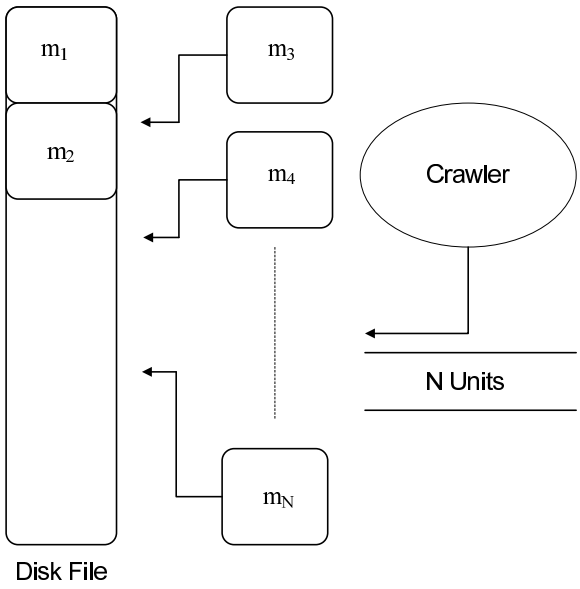
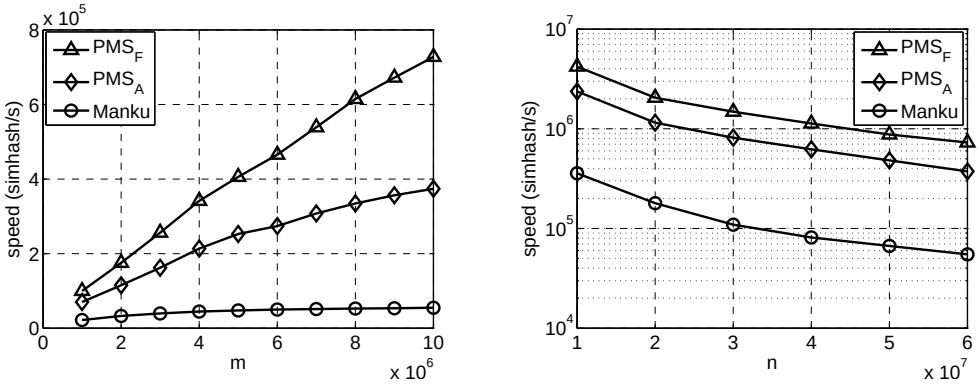


Fig. 9. Working of the batch-mode.

G. Batch Mode

Experiments for batch mode are done for an existing dataset of 60M fingerprints which are read from the disk and a query batch of 10M fingerprints stored in the memory. The task is to find near-duplicates of batch queries in minimum possible time. Batch mode is expensive since every time a new batch arrives, the existing fingerprints have to be read into memory and then near-duplicate search runs over it. Therefore, it is important to be able to process queries of larger batch size at minimum possible time.

Figure 9 shows the working of batch mode in case of a web-crawler. The data is input in batches and when the processing is over, is appended to the disk file after sorting. Every time a new batch arrives, the disk file is read into memory sequentially and near-duplicate search is run. Figure 10 shows the relation of processing speed with increasing query batch size and increasing disk file size. In figure 10(a), the experiment



(a) speed vs m (b) speed vs n

Fig. 10. Efficiency of query performance in batch mode.

of finding processing speed with increasing batch size is done with constant disk file size of 60M fingerprints and in figure 10(b), experiment with increasing disk file size is done at constant batch size of 10M fingerprints. Please note that Manku's method [6] is similar for finding first near-duplicate and all-near duplicates since it matches fingerprints in disk file to fingerprints in the batch and hence cannot stop after the first match. In the first graph, the difference in speed between PSM_A and Manku et al's [6] method keeps growing starting from 3.7 times at 1M fingerprint to 7.72 times at batch size of 10M fingerprints. The difference between PSM_F and [6] grows to $\approx$ 14 times at batch size of 10M fingerprints. In the second graph, the difference in query time between our method and Manku et al's [6] method remains constant at $\approx$ 7 times at different sizes of the disk file size. Therefore, we can say that our method will perform increasingly better at even bigger batch sizes and maintain that performance advantage over at any size of the disk file. For Manku et al's [6] method, since near-duplicate search runs over query batch of 10M fingerprints, we used the 4 table design which proved to be performing better than 10 table design from previous experiments.

CHAPTER VIII

CONCLUSION

In this thesis, we presented a way to find near-duplicates using `simhash` in large datasets which takes less space and performs faster by sacrificing a small recall percentage. For both online and batch mode, our method consistently outperformed [6] in terms of query speed and memory requirement. We also showed with our experiments that our probabilistic method works quite well and is significantly able to reduce the number of fingerprints getting checked with a maximum of two tables. Future work involves studying feature selection for web pages to effectively use our method and cluster our collection of web pages. We would like to work on parallel clustering of large datasets and make it faster and more scalable by using this technique of near-duplicate detection.

REFERENCES

[1] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 billion pages and beyond," *Proc. ACM WWW*, pp. 427–436, 2008.

[2] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental clustering for mining in a data warehousing environment," *Proc. Intl Conf. Very Large Data Bases (VLDB)*, pp. 323–333, 1998.

[3] B. Stein, S. M. zu Eissen, and M. Potthast, "Strategies for retrieving plagiarized documents," *Proc. ACM SIGIR*, pp. 825–826, 2007.

[4] D. Fetterly, M. Manasse, and M. Najork, "On the evolution of clusters of near-duplicate web pages," *1st Latin American Web Congres*, pp. 37–45, 2003.

[5] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," *Proc. ACM STOC*, pp. 604 613, 1998.

[6] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," *Proc. ACM WWW*, pp. 141–150, 2007.

[7] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," *Proc. ACM STOC*, pp. 380–388, 2002.

[8] M. Henzinger, "Finding near-duplicate web pages: A large-scale evaluation of algorithms," *Proc. ACM SIGIR*, pp. 284–291, 2006.

[9] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[10] J. L. Bentley, "K-d trees for semi-dynamic point sets," *ACM Symposium on Computational Geometry (SCG)*, pp. 187–197, 1990.

[11] T. Mitchell, *Machine Learning*, New York, McGraw Hill, 1997.

[12] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," *Proc. Intl Conf. Very Large Data Bases (VLDB)*, pp. 194–205, 1998.

[13] R. Salakhutdinov and G. Hinton, "Semantic Hashing," *Int. J. Approx. Reasoning*, vol. 50, pp. 969–978, 2009

[14] S. Baluja and M. Covell, "Learning forgiving hash functions: Algorithms and large scale tests," *Proc. 20th international joint conference on Artifical intelligence*, pp. 2663–2669, 2007.

[15] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *Journal of Computer and System Sciences*, vol. 60, pp. 327–336, 1998.

[16] T. C. Hoad and J. Zobel, "Methods for identifying versioned and plagiarised documents," *J. Am. Soc. Inf. Sci. Technol.*, vol. 54, pp. 203–215, February 2003.

[17] A. Chowdhury, O. Frieder, D. Grossman, and M. C. McCabe, "Collection statistics for fast duplicate document detection," *ACM Trans. Inf. Syst.*, vol. 20, pp. 171–191, April 2002.

[18] M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: Robust and efficient near duplicate detection in large web collections," *Proc. ACM SIGIR*, pp. 563–570, 2008.

[19] P. Wegner, "A technique for counting ones in a binary computer," *Commun. ACM*, vol. 3, pp. 322–322, May 1960.

[20] D. Knuth, *The Art of Computer Programming*, Boston, MA, Addison-Wesley 3rd edition, 1997.

VITA

Sadhan Sood received his Bachelor of Technology degree in mechanical engineering from Cochin University of Science & Technology-Kochi, India in June 2006. He received his Masters of Science degree in computer science in August 2011 at Texas A&M University, College Station.

His research interests include computer networks and information retrieval. He can be reached at:

c/o Department of Computer Science and Engineering
Texas A&M University
College Station
Texas - 77843-3128

The typist for this thesis was Sadhan Sood.