Probabilistic Near-Duplicate Detection Using Simhash

Sadhan Sood
Texas A&M University
College Station, TX 77843
sadhan@cs.tamu.edu

Dmitri Loguinov[*]
Texas A&M University
College Station, TX 77843
dmitri@cs.tamu.edu

ABSTRACT

This paper offers a novel look at using a dimensionality-reduction technique called `simhash` [8] to detect similar document pairs in large-scale collections. We show that this algorithm produces interesting intermediate data, which is normally discarded, that can be used to predict which of the bits in the final hash are more susceptible to being flipped in similar documents. This paves the way for a probabilistic search technique in the Hamming space of `simhashes` that can be significantly faster and more space-efficient than the existing `simhash` approaches. We show that with 95% recall compared to deterministic search of prior work [16], our method exhibits 4-14 times faster lookup and requires 2-10 times less RAM on our collection of 70M web pages.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Clustering

General Terms

Algorithms

Keywords

Hamming distance, similarity, simhash, clustering

1. INTRODUCTION

Many clustering problems in data mining involve *similarity matching*, which is a process of finding pairs of documents in collection $\mathcal{D}$ that are similar (in some sense) to each other. To decouple similarity from page semantics and ultimately human judgement, it is common to preprocess the data by extracting d-dimensional *feature vectors* (i.e., syntactic elements perceived to be important) from each page and solving the matching problem on them.

When the number of dimensions d is large, many traditional clustering approaches [4], [12] are a $\Theta(n^2)$ endeavor, where $n = |\mathcal{D}|$, that requires an all-to-all comparison across the entire dataset. Since this is infeasible for collections larger than a few thousand pages, another approach [6], [8], [14] is to devise *approximate* algorithms that can compute similarity using much smaller data structures and sub-quadratic overhead. One promising technique in this category, called `simhash` [8], [13], [16], relies on replacing feature vectors with b-bit fingerprints that preserve cosine similarity of the vector space.

The main challenge with `simhash` is to quickly find all pairs of fingerprints within a certain Hamming distance h of each other. For large collections (i.e., billions of pages), the currently fastest and most space-efficient solution [16], which we call *Block-Permuted Hamming Search* (BPHS), relies on permuting chunks of each hash and performing lookups in multiple similarly-permuted copies of the dataset. However, since $\mathcal{D}$ must be replicated at least four times in RAM, BPHS may be unsuitable for certain memory-constrained systems. Even if enough RAM exists to hold $4n$ items, unless the number of copies is scaled with dataset size, the search complexity of this method is still quadratic in n.

Noticing that `simhash` is already probabilistic in nature and does not guarantee 100% recall on vector pairs, it makes sense to ask whether by sacrificing a small additional percentage of recall one can reduce RAM consumption of BPHS, speed up the search, and improve scalability as $n \to \infty$. We investigate this question next.

1.1 Our Contributions

We first analyze the construction process of `simhash` and observe that certain bits are much more *volatile* (i.e., likely to be flipped by modification of features) than others. This leads to a conjecture that the Hamming-distance problem on `simhash` can be solved efficiently by an iterative process that intelligently searches to distance h within the volatile bits and performs lookups in a *single* copy of the dataset, keeping RAM overhead at the theoretical minimum.

Thus, the main question with using this approach is how to design an efficient algorithm that decides which bits to flip, in what order, and when to stop (if less than 100% recall is desired). For a given page u, let $p_i(u)$ be the probability that bit i of u's hash is different from that in other pages of the collection and $\mathcal{S} \subseteq \{1, 2, \ldots, b\}$ be some subset of hash bits. Then, the probability that there exists another `simhash` that differs from u only in bits contained in $\mathcal{S}$ can be estimated as:

$$p(u, \mathcal{S}) = \prod_{i \in \mathcal{S}} p_i(u) \prod_{j \notin \mathcal{S}} (1 - p_j(u)), \tag{1}$$

where independence between the bits is assumed from the `simhash` algorithm [8].

[*]Supported by NSF grants CNS-0720571 and CNS-1017766.

To maximize useful work and allow the search to stop after $k \ll \binom{b}{h}$ attempts, one requires a method for obtaining the top-k subsets $\mathcal{S}$ ($1 \leq |\mathcal{S}| \leq h$) in decreasing order of their probability $p(\mathcal{S})$, but with much lower complexity than exponential needed to compute all possible values (1) and sort them. To solve this problem, we offer a novel algorithm we call *Volatility Ordered Set Heap* (VOSH), which performs the job in the optimal $\Theta(k \log k)$ time and $\Theta(k)$ space. For the common setting $h = 3$ and $b = 64$ suggested for large datasets [16], we compare VOSH with a matching algorithm that flips bits in random order without repetition. Our results show that for 100% recall, the former requires 61 times fewer attempts than the latter; for 80% recall, 151 times fewer; and for 50% recall, 347 times fewer.

We incorporate VOSH in a system we call *Probabilistic Simhash Matching* (PSM) for finding similar documents in large collections $\mathcal{D}$ under two modes of operation – *online* and *batch*. The former case, assuming that $\mathcal{D}$ fits in RAM, aims to minimize the latency of answering queries about incoming pages u. The latter case, assuming that $\mathcal{D}$ is kept on disk, aims to maximize the query throughput rate. We compare our method against BPHS [16] in terms query speed and RAM usage using a web collection with 70M documents. For 95% recall, our results show that PSM requires $2 - 10$ times less RAM, while being $5 - 14$ times faster than BPHS. We also model both approaches and show that with a fixed number of copies of the dataset in RAM, PSM scales as $O(n \log^h(n) \log \log n)$ as opposed to BPHS's $\Theta(n^2)$.

2. RELATED WORK

The first approach to matching similar documents is to convert them to some canonical form in which they become exact duplicates. By removing certain tokens perceived dispensable (e.g., high/low IDF words [9]) and hashing the resulting page to a single fingerprint, similar documents can be found in $\Theta(n \log n)$ time using sorting.

Detecting similar pages over a general space of d-dimensional feature vectors is a more challenging task, especially at non-trivial scale. In the small number of dimensions $d \ll \log_2 n$, special data structures (e.g., R-tree [12], Kd-tree [4]) can find near neighbors in $\Theta(n \log n)$ time; however, their performance deteriorates to that of an exhaustive search (or worse) as d increases. Since web-scale collections usually exhibit dictionaries with millions of unique words, each typically mapping to a feature, this approach is generally inapplicable to such cases.

A more viable solution for large d and n involves approximate algorithms that sacrifice some precision and recall in favor of manageable speed. These include *Locality-Sensitive Hashing* (LSH) [14], *Min-Wise Independent Permutations* [7], `simhash` [8], and many variations [2], [3], [6], [5], [10], [13], [16], [18]. This is the direction we pursue below.

3. FUNDAMENTALS

We start by motivating the usage of fingerprints, analyze the brute-force approach to finding near duplicates, and formulate the problem we aim to solve.

3.1 Framework and Motivation

Since our focus is on large web crawls (i.e., billions of pages), `simhash`'s low space and time complexity [16], as well as high precision [13] compared to other hashing techniques [7], are quite appealing. With this in mind, we next set up the terminology and notation that will be needed later.

n	Cosine $s(u,v)$		Hamming $H(x,y)$	
	Time	RAM	Time	RAM
1M	91 days	1.1 GB	34 min	8 MB
64M	1,020 years	70 GB	97 days	512 MB
8B	261K years	9 TB	68 years	64 GB

Table 1: Extrapolated delay to compute cosine similarity and Hamming distance on all pairs (one core of AMD Phenom II 2.8 GHz).

Let $\mathcal{F} = \{1, 2, \ldots, d\}$ be the set of all unique features across the entire collection and $\mathcal{F}(u) \subseteq \mathcal{F}$ be the set of features present on page $u \in \mathcal{D}$. The number of features $f(u) = |\mathcal{F}(u)|$ can vary from a few hundred to a few thousand depending on which features are selected and the specific dataset. Each feature $i \in \mathcal{F}(u)$ is described by a certain real weight $w_i(u) \in \mathbb{R}$, which measures the importance and contribution of i to u. A combination of $\mathcal{F}(u)$ and weights $\{w_i(u)\}_{i \in \mathcal{F}(u)}$ represents the *feature vector* of page u.

Define $\mathcal{V}$ to be the set of all feature vectors produced by $\mathcal{D}$ and assume some similarity measure $s : \mathcal{V}^2 \to [0, 1]$ that maps pairs of vectors to real numbers (i.e., values close to 0 indicate dissimilar documents and those close to 1 indicate similar ones). In the context of shingles and min-wise hashing [7], $s(u, v)$ is usually Jaccard's coefficient; however, for `simhash` and non-integer weights it is more common to utilize cosine similarity [8], which we also do below whenever comparison in the vector space is needed.

To understand the scale at which similarity can be sought in an all-to-all search among feature vectors, we have the following analysis. Each pair of vectors (u, v) requires $f(u) + f(v)$ operations, some of which can be quite expensive (e.g., multiplication for cosine similarity), for a total overhead of $E[f(u)]n^2$. Assuming ζ bytes are needed to store each feature index and its weight, the expected space requirement of $\mathcal{D}$ is $nE[f(u)]\zeta$. Using our dataset of webpages with $E[f(u)] = 141$ and $\zeta = 8$, the left side of Table 1 shows the amount of time and RAM needed to find all duplicate vectors under cosine $s(u, v)$. Notice in the table that both space and time are prohibitive, even for relatively small datasets.

To make storage and computation more reasonable, a second level of approximation (i.e., `simhash`) is a one-way mapping $\mathcal{V} \to \{0, 1\}^b$ that converts feature vectors to b-bit binary strings. Assuming $\mathcal{H}$ is the collection of hashes produced by $\mathcal{V}$, the main similarity metric on $\mathcal{H}$ is Hamming distance $H(x, y)$ whose benefit over $s(u, v)$ lies in the fact that it can be computed with a handful of CPU instructions [20].

Replacing each feature vector with its 64-bit `simhash`, the right side of Table 1 shows an improvement in computational speed by a factor of 3800 (i.e., from 70K pairs/sec to 268M pairs/sec) and a reduction in space by a factor of 141, both of which are substantial. While this solves the problem for n up to a few million pages, the 3 months for medium-sized collections and the 68 years for large datasets shown in the table are still undesirable in practice. This explains our interest in sub-quadratic techniques for computing Hamming distances over large document sets.

3.2 Problem Formulation

There are two classes of matching problems we consider in this paper. In the first class (e.g., clustering [11]), the

Algorithm 1 Simhash (u)

```
1: $W \leftarrow$ array of $b$ zeros
2: for $i \in \mathcal{F}(u)$ do                        ▷ Examine each feature
3:     $\phi_i \leftarrow$ UniformHash($i$)              ▷ Compute $b$-bit hash
4:     for $j = 1$ to $b$ do                             ▷ Iterate through each bit
5:         if $\phi_{ij} = 1$ then                       ▷ $j$-th bit of $\phi_i$
6:             $W[j] \leftarrow W[j] + w_i$              ▷ Add feature weight
7:         else
8:             $W[j] \leftarrow W[j] - w_i$              ▷ Subtract feature weight
9:         end if
10:    end for
11: end for
12: for $j = 1$ to $b$ do                                ▷ Revisit all bits
13:    if $W[j] \geq 0$ then
14:        $B[j] \leftarrow 1$                           ▷ Positive weight, set bit to 1
15:    else
16:        $B[j] \leftarrow 0$                           ▷ Negative weight, set bit to 0
17:    end if
18: end for
19: return array $B[1 \ldots b]$                         ▷ simhash
```

goal is to find *all* matches for a given page. Knowing pairwise similarity among pages, one can use separate clustering algorithms, which we do not consider here, to combine the various documents into groups.

OBJECTIVE 1. *Given x, find all $y \in \mathcal{H}$ s.t. $H(x,y) \leq h$.*

In the second class (e.g., duplicate elimination [13], plagiarism detection [19]), the goal is to determine if there exists *at least one* similar page in the dataset, without finding all matching documents, which often can save significant processing overhead and increase performance.

OBJECTIVE 2. *Given x, find any $y \in \mathcal{H}$ s.t. $H(x,y) \leq h$.*

It should be noted that these goals are not specific to what features are being used (e.g., individual words, shingles, HTML tags, anchor text, URLs), their weights (e.g., frequency, TF-IDF, HTML highlight), dimensionality of the vectors, or classification purpose.

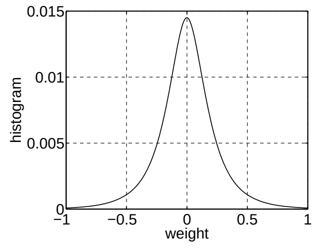
4. UNDERSTANDING SIMHASH

We next explain the `simhash` construction algorithm and dissect the properties of its hashes.

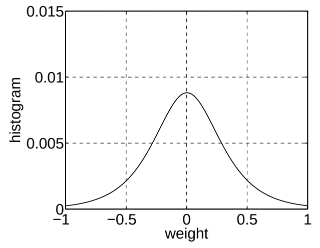
4.1 Algorithm

Recall that `simhash` is a feature fingerprinting technique that uses random projections to generate compact representation of high-dimensional vectors. Its main characteristic is that the Hamming distance between two document fingerprints is positively correlated with cosine similarity between the corresponding feature vectors.

Using the notation of the previous section, Algorithm 1 explains the `simhash` process of [13], [16], which we discuss in more detail next. Denote by W a vector of temporary weights that the `simhash` function generates. As we will see later, these weights are important elements of the proposed framework. For each feature i in the current document u, Algorithm 1 first computes its uniformly random hash ϕ_i and then decides to either add or subtract the feature's weight w_i to/from W_j based on whether the j-th bit ϕ_{ij} of the uniform hash is zero or one. After all features have been processed, bits of `simhash` with non-negative W_j are set to 1 and the remaining bits are set to 0.



(a) weight $W_1(u)$



(b) re-write weight Y_1

Figure 1: Histograms of simhash weights (bin size 1/150, collection of 70M documents).

4.2 Weights and Hamming Distance

We are now ready to examine the distribution of weights $\{W_j\}$ and lay the foundation for understanding which bits are more likely to be different between similar documents.

Personalizing each variable in the algorithm with its page u, it is not difficult to notice that `simhash` computes a dot-product between a global vector of iid random variables and a vector of u's specific feature weights:

$$W_j(u) = \sum_{i \in \mathcal{F}(u)} (2\phi_{ij} - 1) w_i(u), \tag{2}$$

where $\{\phi_{ij}\}_{ij}$ are iid Bernoulli variables with $E[\phi_{ij}] = 1/2$. This makes each weight $W_j(u)$ zero-mean:

$$E[W_j(u)] = \sum_{i \in \mathcal{F}(u)} E[2\phi_{ij} - 1] E[w_i(u)] = 0. \tag{3}$$

Using normalized TF-IDF weights $w_i(u)$, Figure 1(a) shows the distribution of $W_1(u)$ from (2) using our collection of 70M documents. We focus only on the first bit since the others produce identical results. While the Central Limit Theorem suggests that constant feature weights (e.g., used in [13]) are expected to converge $W_j(u)$ to a Gaussian distribution, TF-IDF weights interestingly lead to a Laplace-like distribution (log-linear plots of tails are omitted for brevity).

To control the variance and distribution of resulting sums, one can generalize (2) to non-Bernoulli cases. Assume $\nu_i = (\nu_{i1}, \ldots, \nu_{ib})$ is a vector permanently associated with feature $i \in \mathcal{F}$ whose elements are drawn from a zero-mean symmetric distribution. Then, (2) becomes:

$$W_j(u) = \sum_{i \in \mathcal{F}(u)} \nu_{ij} w_i(u), \tag{4}$$

while the rest of the algorithm remains the same. Assuming $\theta \in [0, \pi]$ is the angle between two d-dimensional vectors, [8] shows that if ν_{ij} are zero-mean Gaussian variables, the probability that the corresponding fingerprints collide on a given bit is $q = 1 - \theta/\pi \approx (\cos\theta + 1)/2$. While no similar closed-form result is available for (2), intuition suggests that a correlation between $\cos\theta$ and q still exists.

Since $w_i(u)$'s positive contribution to bit j_1 has no bearing on whether its contribution to another bit j_2 will be positive or negative, it follows that bits in the final `simhash` are *pairwise independent*, which implies that $H(x,y)$ is a binomial random variable with parameters b and $1-q$. While a-priori this tells us nothing about which bits are likely to differ in a random pair (x,y), we next show that utilizing the knowledge of x's `simhash` weights $\{W_j\}$ allows an informed decision.

4.3 Why Bits Flip

Suppose page u undergoes a re-write, which includes deletion of α existing features and addition of β new features. This modifies each `simhash` weight j to:

$$W'_j(u) = W_j(u) - X_j^\alpha + X_j^\beta, \tag{5}$$

where X_j^α and X_j^β are summations in the form of (2) that correspond to the deleted/added features. Define $Y_j = X_j^\alpha - X_j^\beta$ to be the random weight change that occurs in response to these modifications. Since we are interested in changes to u that exist in our collection $\mathcal{D}$ (as opposed to an infinite number of hypothetical modifications), Y_j can be viewed as a random variable with the same distribution as $\{W_j(u) - W_j(v)\}_{u,v\in\mathcal{D}}$, which is shown in Figure 1(b). The main difference from part (a) is the doubled variance (i.e., $Var[Y_j] = Var[W_j(u)] + Var[W_j(v)] = 2Var[W_j(u)]$) and a slightly more Gaussian shape; however, the log-scale shows that both tails remain exponential rather than Gaussian.

Conditioned on $W_j(u) = c$, it follows that $W'_j(u) = c - Y_j$, which suggests that the difficulty of flipping bit j is directly related to the magnitude of c. Indeed, first notice that Y_j is zero-mean since $E[X_j^\alpha] = E[X_j^\beta] = 0$, which follows from (3). Second, averaged over all possible pairs (α, β), the distribution of Y_j is symmetric around 0 since X_α and X_β are iid. We can make this argument due to the non-adversarial nature of changes applied to the page. As a result, we have $P(Y_j > |c|) = P(Y_j < -|c|)$. Finally, in order to flip bit j, one must encounter a modification that satisfies both a) $|Y_j| > |c|$ and b) $\text{sign}(Y_j) = \text{sign}(c)$. The probability of this happening is $P(Y_j > |c|) = 1 - F_j(|c|)$, where $F_j(x)$ is the CDF of Y_j. Due to the monotonicity of CDFs, this probability monotonically decays with $|c|$.

This brings us to our main result of the section. Observe that each `simhash` contains certain bits that are much more volatile than others. Assume page v is similar to u according to cosine similarity at some threshold (e.g., 0.9). Then, if v manages to flip some bits in u's `simhash`, the likely order of these flips follows from the smallest $|W_j(u)|$ to the largest. For example, consider `simhash` weights $(1.9, 0.01, -0.5)$ for the first three bits and the distribution of Y_j from Figure 1(b). If anything at all is flipped by small changes, it will most likely be a single bit 2. Much more effort is needed to flip bit 3, while bit 1 requires massive summations of added/deleted features in (2) to overpower its weight 1.9.

While it is straightforward to generate lookups for Hamming distance $h = 1$ (i.e., by sorting all bits in the increasing order of $|W_j(u)|$), multi-bit flips are more difficult. For example, is the two-bit combination with weights $(1.9, 0.01)$ more likely than a single-bit combination with weight -0.5? We study this question next.

5. BIT ORDER

This section develops an efficient technique for deciding the order in which bits should be attempted during similarity search on $\mathcal{H}$.

5.1 Model

Define $p_j(u) = P(Y_j > |W_j(u)|)$ to be the probability of that another page in $\mathcal{D}$ has flipped bit j in the `simhash` of a given page u. While obtaining a closed-form model for $p_j(u)$ might be possible in future work, an empirical distribution of Y_j sampled from pages in $\mathcal{D}$, e.g., as in Figure 1(b), is sufficient for estimating $p_j(u)$ in practice.

Let $\mathcal{S} \subseteq \{1, 2, \ldots, b\}$ be a non-empty subset of hash bits. Then, the probability that there exists a page in $\mathcal{D}$ that differs from u only in the bits contained in $\mathcal{S}$ is:

$$p(u, \mathcal{S}) = \prod_{i\in\mathcal{S}} p_i(u) \prod_{j\notin\mathcal{S}} (1 - p_j(u)). \tag{6}$$

To limit the Hamming search to the most likely subsets, one requires an ordering of $\{\mathcal{S}\}$ according to (6). The brute-force approach would be to generate all possible subsets $\mathcal{S}$, sort them in the decreasing order of $p(u, \mathcal{S})$, and then select the top-k elements, where k controls the tradeoff between overhead and recall. The main problem with this method is that it requires $\Theta(l \log l)$ operations and $\Theta(l)$ space, where $l = \sum_{i=1}^{h} \binom{b}{i}$ is often quite large. We next offer a better algorithm that solves this problem in optimal space and time.

5.2 Volatility Heap

We start by considering the problem at fixed Hamming distance h. We later generalize the solution to all distances up to h. Throughout this section, it is convenient to refer to bits in decreasing order of their probability $p_j(u)$ rather than their physical position in the hash (i.e., bit 1 is the most volatile and bit b is the least). Similarly, each set $\mathcal{S}$ is assumed to be sorted in decreasing volatility of its bits (e.g., $\mathcal{S} = (1, 3, 7)$ refers to the first, third, and seventh most volatile bits of the hash).

Define $\prec$ to be a lexicographical comparison operator on $\{\mathcal{S}\}$. If $\mathcal{S}_1 \prec \mathcal{S}_2$, then there exists an index $i < h$ such that the two sets share the leading i elements, but $\mathcal{S}_2$ is larger in the $(i+1)$-st element. For example, $(1, 3, 7, 15) \prec (1, 3, 9, 10)$. Then, we have the following important result.

LEMMA 1. *If two sets $\mathcal{S}_1, \mathcal{S}_2$ of the same size h differ in exactly 1 bit and $\mathcal{S}_1 \prec \mathcal{S}_2$, then $\forall u : p(u, \mathcal{S}_1) \geq p(u, \mathcal{S}_2)$.*

PROOF. Suppose the bits that differ are i in $\mathcal{S}_1$ and j in $\mathcal{S}_2$. Since (6) has $h - 1$ terms in each product that are common between $p(u, \mathcal{S}_1)$ and $p(u, \mathcal{S}_2)$, we get:

$$\frac{p(u, \mathcal{S}_1)}{p(u, \mathcal{S}_2)} = \frac{p_i(1 - p_j)}{p_j(1 - p_i)} = \frac{p_i - p_i p_j}{p_j - p_i p_j}. \tag{7}$$

Recalling that $\mathcal{S}_1 \prec \mathcal{S}_2$, notice that $i < j$ and $p_i \geq p_j$, which immediately shows that (7) is lower-bounded by 1. □

This result paves the way for an algorithm that sorts bit combinations using a structure we call *Volatility Ordered Set Heap* (VOSH). It starts with the optimal set $\mathcal{S}_0 = (1, 2, \ldots, h)$ in the root and iteratively generates for each existing node $\mathcal{S}_i$ two children, each of whom succeeds $\mathcal{S}_i$ according to $\prec$ and differ from the parent in exactly one bit. From Lemma 1, observe that each child's $p(u, \mathcal{S})$ is always no larger than the parent's. Using transitivity of $\prec$ and $\geq$, we obtain that each $\mathcal{S}_i$ contains bits whose combination is at least as likely as that of any node in its entire subtree.

We next describe how the tree is constructed using Algorithm 2. Given node $\mathcal{S}$ with h bits, VOSH attempts to generate two children. The left child always increments the last bit of the parent as long as the result does not exceed b (if it does, the left branch stops). The right child scans the parent's bits from right to left until it finds the first gap

Algorithm 2 ProduceChildren($\mathcal{S}$)

```
1:  if $\mathcal{S}[h] \le b - 1$ then                         ▷ Attempt left child
2:      $\mathcal{S}_L \leftarrow \mathcal{S}$;   $\mathcal{S}_L[h] \leftarrow \mathcal{S}_L[h] + 1$     ▷ Add 1 to last bit
3:  else
4:      $\mathcal{S}_L \leftarrow \emptyset$                            ▷ Left child is empty
5:  end if
6:  for $j = h - 1$ downto 1 do                   ▷ Attempt right child
7:      gap = $\mathcal{S}[j+1] - \mathcal{S}[j]$
8:      if gap = 2 then                    ▷ Should increment bit $j$?
9:          $\mathcal{S}_R \leftarrow \mathcal{S}$;   $\mathcal{S}_L[j] \leftarrow \mathcal{S}_L[j] + 1$
10:     else if gap > 2 then
11:         $\mathcal{S}_R \leftarrow \emptyset$;   break              ▷ Right child is empty
12:     end if
13: end for
14: return $(\mathcal{S}_L, \mathcal{S}_R)$                     ▷ Produce both children
```

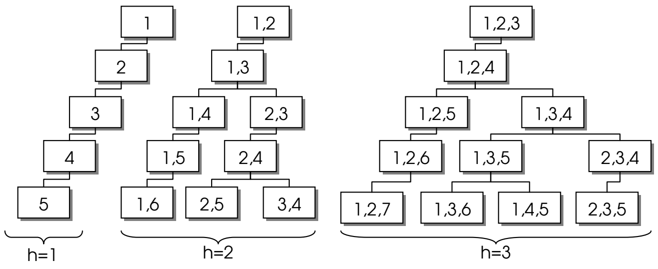


Figure 2: Top five levels of three volatility heaps.

in bit numbers of size at least 2. If the gap is exactly 2, the bit on the left of the gap is incremented. Otherwise, the right child is omitted. The last nuance is necessary to prevent generation of duplicate nodes along different branches of the tree. Figure 2 shows the top five levels of three VOSH heaps that correspond to $h = 1, 2, 3$.

Upper-bounding each heap size by k, space and time complexity of constructing all VOSH trees to depth h is $\Theta(k)$. We next explain how to use them during Hamming search to decide the order of bit flips.

6. PROBABILISTIC SEARCH

In this section, we present our algorithm for near-duplicate search in the `simhash` space. We start with the bit-generation process, verify its effectiveness, and then discuss online/batch-mode operation on large datasets.

6.1 Bit Selection

While VOSH ensures that each subtree should not be traversed before its root, it does not (and cannot) decide which of the two siblings at each level is more optimal. As explained next, we make this decision during run-time using an additional max-heap M that operates on `(key, value)` pairs, where the key is $p(u, \mathcal{S})$ and the value is set $\mathcal{S}$.

Given a page u, we first compute its weights $\{W_j(u)\}$ and the corresponding $\{p_j(u)\}$. We then populate M with tuples $(p(u, \mathcal{S}), \mathcal{S})$ generated by h root nodes of VOSH trees, each corresponding to a different number of bits. At each bit-flip, we extract from M the node with the largest $p(u, \mathcal{S})$, obtain its children from the corresponding VOSH tree, compute their probabilities $p(u, \mathcal{S})$, and insert their tuples into M. This guarantees traversal of bit combinations in the decreasing order of $p(u, \mathcal{S})$ and keeps the total complexity of k flips at the optimal $\Theta(k \log k)$.

In [16], $b = 64$ and $h = 3$ were tested in an 8B-page collection and found to work well. We later verify that these numbers are also quite appropriate for our dataset; in the meantime, use them as our target combination for all experiments and analysis. To understand the performance of VOSH combined with M, we extract from our dataset 8M random pairs of `simhashes` at Hamming distance $h = 1, 2, 3$. We then compare our VOSH-driven approach to random flipping of bits, where the latter considers all possible combinations of exactly h bits, ensuring there is no repetition.

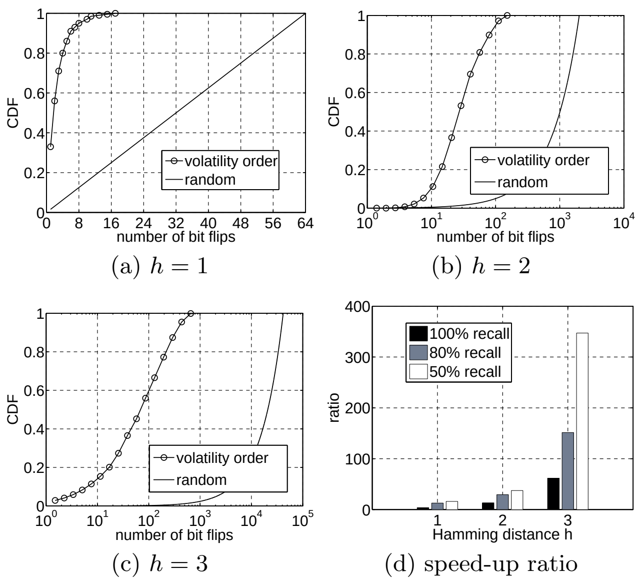


Figure 3: Bit flips needed in VOSH compared to random (64-bit hashes).

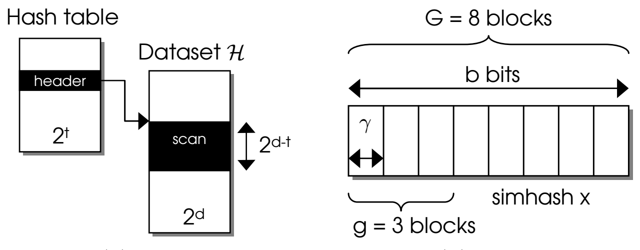
Figure 3(a) shows the CDF of the number of bit attempts needed to find each of the matches at distance $h = 1$. Observe in the figure that the first VOSH flip finds the matching pair in over 30% of the cases and four flips accomplish the same 80% of the time. All pairs at Hamming distance 1 are discovered in 17 or fewer attempts, while the random approach requires 64 flips to achieve the same 100% recall. For the exact distances $h = 2$ and $h = 3$ (drawn on a log-linear scale), VOSH finds all pairs in 152 and 675 flips, respectively. This compares favorably to 2,016 and 41,664 attempts needed by the random approach.

The difference between VOSH and random flipping becomes more pronounced as h increases and recall decreases. This is illustrated in Figure 3(d), which plots the ratio of the number of attempts needed between the two methods for 50%, 80%, and 100% recall. At $h = 1$, VOSH is 3.7 times faster than random at 100% recall and 16 times faster at 50%. These numbers increase to 13 and 37 respectively for $h = 2$, eventually becoming 61 and 347 for $h = 3$.

With these encouraging results in mind, we next describe how VOSH can be applied to large-scale datasets in a framework we call *Probabilistic Simhash Matching* (PSM).

6.2 Online Queries

In online mode, the existing fingerprints $\mathcal{H}$ are stored in memory to support similarity queries about arriving pages. The main performance metric in this setup is delay τ needed to compare a new document against a set $\mathcal{H}$ of n `simhashes`.



(a) PSM (b) BPHS

Figure 4: Lookup in PSM and BPHS.

To support efficient lookups, assume $\mathcal{H}$ is sorted before the algorithm starts and let $n = 2^d$ be a power of 2. Using VOSH across all b bits is extremely wasteful as it produces a large number of hashes that do not exist in the collection, which is especially noticeable when $n \ll 2^b$. Given $b = 64$, most datasets $\mathcal{D}$ fall into this category. It thus makes sense to limit the random lookups to some small range of $t \leq d$ uppermost bits, which we call the *header*, and perform a linear scan to match the remaining bits.

The matching process may perform a d-step binary search on $\mathcal{H}$ or utilize a hash table of size 2^t for efficient header lookups. We use the latter approach in the paper and illustrate the resulting system in Figure 4(a). Given a query page u with `simhash` x, the first lookup is x itself at Hamming distance 0. We then use VOSH to generate k new queries $x_1, \ldots, x_k$ by working with t most-significant bits of x. In each lookup, if i bits are flipped in the header, the linear scan flags all pages whose Hamming distance to x in the remaining $b - t$ bits is no more than $h - i$.

If only a single-match is desired, the linear scan stops as soon as it identifies the first similar page; otherwise, it continues until a header mismatch. To differentiate these cases, we use PSM$_F$ (first) and PSM$_A$ (all), respectively.

6.3 Batch Queries

In batch mode, the existing collection $\mathcal{H}$ is stored on disk. New fingerprints are accumulated in set $\mathcal{Q}$ until it reaches size m. After $\mathcal{Q}$ becomes full, PSM scans file $\mathcal{H}$ by reading it in chunks of m fingerprints and matching each $x \in \mathcal{Q}$ against the loaded chunk using the online method described above. After $\mathcal{Q}$ is processed to completion, it is sorted and appended to the existing file. This removes the need to sort chunks again when they are loaded in RAM.

It should be noted that PSM$_F$ can stop reading the file as soon as finds at least one near-duplicate for each fingerprint in $\mathcal{Q}$, which in certain cases can save significant amounts of overhead. In the worst case, however, both PSM$_F$ and PSM$_A$ read the entire file for each $\mathcal{Q}$. In a multi-core system, the lookups can be easily parallelized by splitting $\mathcal{Q}$ across threads. It also makes sense to pre-process $\mathcal{Q}$ by identifying the volatile bits in each hash and retaining probabilistic information only related to them. This not only saves memory, but also saves time in identifying top bit-combinations every time a new chunk is loaded into memory.

7. PERFORMANCE MODELING

This section models the currently fastest and most space-efficient `simhash` matching approach [16] and compares its overhead to that of PSM.

7.1 Online BPHS

Efficient search in the Hamming space is an old problem [17] that has remained difficult to solve at large scale. For small h, [16] offers an algorithm, which we call *Block-Permuted Hamming Search* (BPHS), for dealing with large collections $\mathcal{D}$. Suppose we are interested in finding all hashes $y \in \mathcal{H}$ within Hamming distance $h \geq 1$ of x. BPHS first splits the b-bit hash x into $G \geq h+1$ non-overlapping blocks of $\gamma = b/G$ consecutive bits. It then selects an integer g between 1 and $G - h$. If $H(x, y) \leq h$, then block-by-block comparison between x and y is guaranteed to have at least g exact matches. The goal of BPHS is to group all possible combinations of g blocks at the front of the hash and perform the search only on them. This is shown in Figure 4(b) for $G = 8$ and $g = 3$, which is a combination that can support all $h \leq 5$.

Similar to PSM, define the leading g blocks of the hash as its *header*. Then, there are $T = \binom{G}{g}$ ways of selecting the g blocks for the header. Call each of these selections a *block permutation* $\pi_i(x)$ of the original hash. Since the order of blocks neither in the header nor in the rest of the hash matters, there are exactly T unique permutations, which applied to $\mathcal{H}$ produce T copies of the dataset $\mathcal{H}_1, \ldots, \mathcal{H}_T$.

After sorting the copies (which are called *tables* in [16]), the lookup proceeds in T iterations. For the i-th step, BPHS uses a binary search to find the numerically smallest hash in $\mathcal{H}_i$ whose header matches that of $\pi_i(x)$. Starting with that hash, it scans $\mathcal{H}_i$ linearly until the first header mismatch. During this scan, it computes the Hamming distance in the lower $b - (g\gamma)$ bits between each target hash and x, with the rest of the algorithm being similar to PSM. We consider two versions: BPHS$_F$ stops after finding the first match, while BPHS$_A$ always checks every eligible hash.

For the analysis, we assume a sufficiently large b to ignore round-off effects during division and model only the overhead of BPHS$_A$. Define δ_R to be the RAM latency during random access, δ_P to be the delay needed to permute the blocks of x, and δ_L to be the per-hash delay needed to perform Hamming-distance calculations during linear scan. Then, define the number of bits that are matched during binary search as:

$$t = \min(g\gamma, d) = \min\Big(\frac{gb}{G}, d\Big), \tag{8}$$

which has a similar meaning to t in PSM. Note that the min function is necessary since the binary search exhausts the entire dataset in d steps. Then, the total the lookup latency of BPHS in T tables is:

$$\tau = \binom{G}{g} \left(\delta_P + t\delta_R + 2^{d-t}\delta_L\right). \tag{9}$$

The storage overhead (in bytes) of BPHS is simply:

$$\Omega = \binom{G}{g} \frac{nb}{8}. \tag{10}$$

We next analyze the issue of optimally selecting G. For a fixed g and assuming $g\gamma \leq d$, (9) breaks into two terms:

$$\tau = \frac{(G-1)!b}{(G-g)!(g-1)!}\delta_R + \binom{G}{g}\Big(\delta_P + 2^{d-gb/G}\delta_L\Big), \tag{11}$$

both of which monotonically increase in G. For $g\gamma > d$, we have an even simpler situation:

$$\tau = \binom{G}{g}(\delta_P + d\delta_R + \delta_L), \qquad (12)$$

which also shows that choosing the smallest possible G leads to best performance. From (10), we can make the same observation about RAM overhead. Since both space and time decrease with G, it follows that its optimal value is its lower bound $g + h$. Re-writing (9), we have:

$$\tau = \binom{g+h}{g}\left(\delta_P + t\delta_R + 2^{d-t}\delta_L\right). \qquad (13)$$

Next, notice that increasing $g\gamma$ beyond d hurts performance in (13) as it keeps $t = d$ constant, but increases the leading binomial coefficient, which makes the final model:

$$\tau(d) = \binom{g+h}{g}\left(\delta_P + \frac{gb}{g+h}\delta_R + 2^{d-gb/(g+h)}\delta_L\right), \quad (14)$$

where

$$g \le \frac{dh}{b-d}. \qquad (15)$$

Determination of optimal g is impossible without knowing the relationship between δ_R and δ_L, as well as the value of b in comparison to d. Assuming 8 GB RAM and the absolute minimum number of tables $T = 4$ for $h = 3$ (i.e., $G = 4, g = 1, \gamma = 16$ bits), BPHS admits datasets up to $n = 2^{28}$ with an expected length of linear scans $2^{28-16} = 2^{12}$.

For a RAM-restricted system that cannot grow T to infinity, notice that $\tau(d)$ scales exponentially with d, or in other words, linearly with set size n. Thus, in such cases, similarity search for all document pairs in $\mathcal{H}$ requires complexity $\Theta(n^2)$. We verify this finding in the next section.

7.2 Batch BPHS

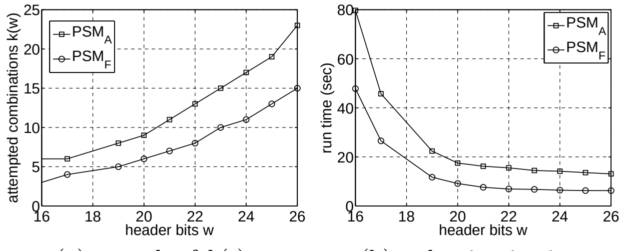
Batch mode in BPHS is similar to that in PSM, with two exceptions explained in [16]. First, matching in BPHS proceeds by checking each fingerprint y from the loaded chunk against T tables built around `simhashes` in $\mathcal{Q}$. This is necessary to prevent repeated construction of T permuted copies of each loaded chunk. One peculiar side-effect of this optimization is that BPHS cannot stop when it finds the first match for y since there might be fingerprints $x \in \mathcal{Q}$ that still do not have any matches. Thus, in batch mode, BPHS$_F$ is identical to BPHS$_A$. The second difference from PSM is that batches $\mathcal{Q}$ do not have to be sorted before being appended to the file since the search runs against $\mathcal{Q}$ rather than $\mathcal{H}$.

To process a batch of $m = |\mathcal{Q}|$ hashes, the I/O delay is the time needed to read the entire file:

$$\tau_{disk} = \frac{nb}{8D}, \qquad (16)$$

where D is the read speed of the hard drive. For high-performance RAID-based configurations and overlapped I/O, computation can be executed while the next chunk is being read. In such cases, the bottleneck is in the CPU portion of the overhead, which consists of the time to permute the m incoming hashes T times, sort the corresponding tables, and perform n lookups in them:

$$\tau_{CPU} = Tm(\delta_P + \log_2 m) + n\tau(\log m), \qquad (17)$$



(a) growth of $k(t)$ (b) reduction in time

Figure 5: Optimal selection of t in PSM.

where $\tau(.)$ is given by the online model (14). Finally, the throughput of this system in hashes per second is:

$$r = \frac{m}{\max(\tau_{disk}, \tau_{CPU})}. \qquad (18)$$

In the experiments below, $\tau_{CPU} \approx 20\tau_{disk}$ dominates and rate r is determined solely by the performance of the studied algorithms rather than the disk speed.

7.3 Online PSM

Recall that VOSH flips k combinations in the upper t bits of each fingerprint x. PSM then searches for these combinations in a single copy of the dataset $\mathcal{H}$ in RAM. The latency of each lookup consists of the VOSH overhead δ_V per examined element in the heap (which hash depth $\log_2 k$), a single visit to the header hash table, and the linear scan:

$$\tau'(d) = k(t)(\delta_V \log_2 k(t) + \delta_R + 2^{\max(d-t,0)}\delta_L), \qquad (19)$$

where we make k explicitly depend on t. Since increasing t increases $k(t)$, the following simple analysis helps choose the optimal t. Recall that each VOSH combination is limited to h bits out of t possible, which means that $k(t)$ is upper-bounded by:

$$k(t) \le \sum_{i=1}^{h}\binom{t}{i} = \Theta(t^h). \qquad (20)$$

Therefore, ignoring the small terms $\delta_V(k)$ and δ_R in (21) and only considering $t \le d$, the dominating term of the delay is $\Theta(t^h 2^{-t})$, which is minimized when t is maximized. This shows that the optimal choice is $t = d$ and:

$$\tau'(d) = k(d)(\delta_V \log_2 k(d) + \delta_R + \delta_L). \qquad (21)$$

To understand the growth of $k(t)$ with t, we partition our 70M-page web collection into two parts – 10M random pages are selected to arrive in online mode and the remaining 60M are chosen for the main dataset $\mathcal{H}$. We first pass each online page through BPHS to find all matches in $\mathcal{H}$ within Hamming distance $h = 3$. We then select for each t such $k(t)$ that achieve 95% recall in PSM compared to BPHS.

The result is plotted in Figure 5(a). As t grows from 16 bits to 26, the number of combinations required by PSM$_A$ increases from 6 to 23 and that for PSM$_F$ from 3 to 15. The exact growth rate cannot be readily ascertained over this small range, but it is visibly super-linear. The total run time to verify 10M hashes in online mode is shown in Figure 5(b). As predicted, the delay is minimized when t is at its maximum (i.e., d), in which case the processing speed reaches 1.6M arriving hashes/sec for PSM$_F$ and 765K/sec for PSM$_A$.

Since PSM maintains a hash table with 2^t entries and a single copy of $\mathcal{H}$, it storage requirement for $t = d$ is:

$$\Omega' = (2^t + 2^d)\frac{b}{8} = \frac{2nb}{8}, \tag{22}$$

which is equivalent to 2 permuted tables in BPHS. For $h = 3$ and the minimum four tables in BPHS, our approach is at least twice as efficient. Furthermore, for a fixed number of additional tables (i.e., just one), PSM's latency (21) scales as $\Theta(k(d) \log k(d))$, where $d = \log_2 n$. Since $k(d)$ is $O(d^h)$, we obtain that its overall complexity for processing all hashes in $\mathcal{H}$ is $O(n \log^h(n) \log\log n)$. This compares favorably to $\Theta(n^2)$ of BPHS.

7.4 Batch PSM

Given a batch $\mathcal{Q}$ of new pages, PSM loads chunks of $m = |\mathcal{Q}|$ fingerprints from the file, sets up a hash table for each element of the chunk, and performs lookups for all $x \in \mathcal{Q}$. Note that after the first match, PSM$_F$ removes x from $\mathcal{Q}$, which prevents its being checked against subsequent chunks. This allows PSM$_F$ to become much faster as it progresses through the file.

In the worst case, PSM's I/O delay is the same as (16), which in our tests is again much smaller than the CPU latency, where the latter can be broken down into three parts – producing the permutations for m hashes, performing m lookups in each of n/m chunks, and finally sorting the hashes in $\mathcal{Q}$ before writing them to disk:

$$\tau'_{CPU} = m\delta_V \log_2 k + n\tau'(\log m) + m\delta_S \log_2 m, \tag{23}$$

where δ_S is the mean latency of moving hashes while sorting.

As important result of this analysis is how batch size m affects rate r in (18). For a fixed number of tables T, BPHS's τ_{CPU} scales as $\Theta(m)$ and keeps r virtually unchanged. On the other hand, ignoring the VOSH and sorting overhead, PSM's τ'_{CPU} scales as $\Theta(\log^h(m) \log\log m)$. This increases the overall rate r slightly slower than linear, i.e., as $\Theta(m/\log^h(m) \log\log m)$, but nevertheless significantly faster than in BPHS. We re-examine this issue and confirm this result in the next section.

8. EXPERIMENTS

We next describe our dataset $\mathcal{D}$, examine whether `simhash` indeed approximates cosine similarity on $\mathcal{D}$, select the optimal h, and evaluate PSM in comparison to BPHS.

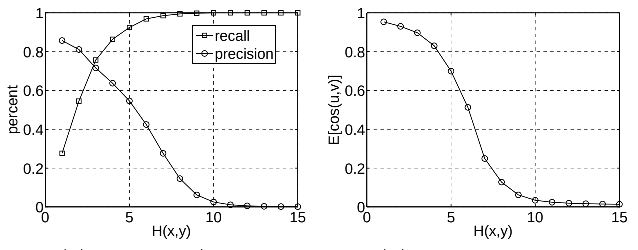
8.1 Dataset

All our experiments involve a set of 100M web pages crawled by IRLbot [15] in April 2008. We process the collection by removing pages that have size less than 5 KB, contain no URLs, have an exact duplicate (identified using a standard hash function), or consist of non-English words, all of which shrinks $\mathcal{D}$ to a total of 70M pages.

We parse each remaining page, removing stop-words and stemming all text outside of HTML tags. We then create feature vectors with weights $t_i(u)$ being the normalize TF-IDF score of each word i on page u. For calculating `simhash` fingerprints, we use the 64-bit MurmurHash function [1].

8.2 Usability of Simhash

While [16] has shown in small-scale manual verification that $h = 3$ and $b = 64$ produce good results on Google's



(a) precision/recall (b) expected cosine

Figure 6: Comparison of simhash against cosine similarity at different Hamming distances.

dataset with 8B pages, we aim to verify that the same parameters work well for our $\mathcal{D}$, but using the similarity measure of the feature-vector space and at a much larger scale. To our knowledge this has not been done before.

We randomly sample 100M document pairs (out of a total of 4.9 quadrillion) and place them into set Υ. We then define set $N_{\cos}$ to contain all pairs whose cosine similarity is above threshold $\theta = 0.9$, i.e., $N_{\cos} = \{(u, v) \in \Upsilon : \cos(u, v) \geq \theta\}$. We also define sets N_h (for $h = 1, 2, \ldots, 64$) to contain all sampled pairs of documents within Hamming distance h of each other, i.e., $N_h = \{(x, y) \in \Upsilon : H(x, y) \leq h\}$.

Then, *precision* at distance h is defined as the fraction of pages in N_h that belong to $N_{\cos}$:

$$P(h) = \frac{|N_h \cap N_{\cos}|}{|N_h|} \tag{24}$$

and *recall* at distance h is the fraction of pages in $N_{\cos}$ that belong to N_h:

$$R(h) = \frac{|N_h \cap N_{\cos}|}{|N_{\cos}|} \tag{25}$$

Figure 6(a) plots both metrics as a function of h. Observe that small h produces a high rate of false-negatives, but keeps the false positive rate low. Large values of h offer the opposite condition – many false positives, but few false negatives. Taking the point where both curves intersect, we arrive at $h = 3$ as a sensible balance for this tradeoff.

Figure 6(b) shows the expected cosine $E[\cos(u, v)]$ between document pairs at different Hamming distances. The correlation between the two is very clear, with $H(x, y) \in [1, 3]$ producing $E[\cos(u, v)]$ that ranges from 0.95 down to 0.89, which confirms the applicability of `simhash` as a substitute for $s(u, v)$ on this dataset.

8.3 Implementation Details

We implemented both PSM and BPHS [16] in Visual Studio C++, using similar optimizations and running them on the same hardware, which consisted of a desktop machine with the AMD Phenom II X6 CPU (2.8 GHz), 16 MB of RAM, 5 TB of disk space, and Windows Server 2008 R2.

Since PSM is an approximation to an exact `simhash` search, its *relative recall* R' is computed against the matches found by BPHS. Thus, PSM's *total recall* against cosine similarity is $R(h)R'$, where $R(h)$ is given in (25) and plotted in Figure 6(a). It should be noted that PSM's relative precision is 100% against BPHS and is not a factor in our comparison.

Relative recall R' for our experiments is defined slightly differently for PSM$_A$ and PSM$_F$. In the former case, we take the number of matching pairs found by PSM$_A$ and normalize

Method	Tables	Time (sec)	Queries/sec
BPHS$_A$	4	65	154K
	10	53	189K
PSM$_A$	1.06	15.5	645K
	1.125	14.4	694K
	1.25	14.4	694K
	1.5	13.5	741K
	2	13.1	765K
BPHS$_F$	4	54	185K
	10	26	385K
PSM$_F$	1.06	6.9	1.4M
	1.125	6.7	1.5K
	1.25	6.4	1.56M
	1.5	6.26	1.6M
	2	6.25	1.6M

Table 2: Comparison of PSM to BPHS (online mode, 10M queries, 60M existing hashes).

it by that found by BPHS$_A$. In the latter case, we record the number of `simhashes` for which PSM$_F$ found at least one similar pair and divide it by the same number in BPHS$_F$.

All our experiments use $k(t)$ such that R' achieves recall 95%, unless otherwise specified. As discussed earlier, we divide the full dataset of 70M fingerprints by random sampling into two parts: 10M hashes are used as queries and the remaining 60M are used as the existing collection $\mathcal{H}$.

We experimented with replacing the binary search in BPHS with extrapolation search suggested in [16]. For $d = 26$, this reduced the number of RAM lookups from 26 to 11, but the overall runtime of the algorithm increased due to the larger number of multiplications/divisions needed to compute each jump. We thus do not include it in our comparison.

8.4 Online Mode

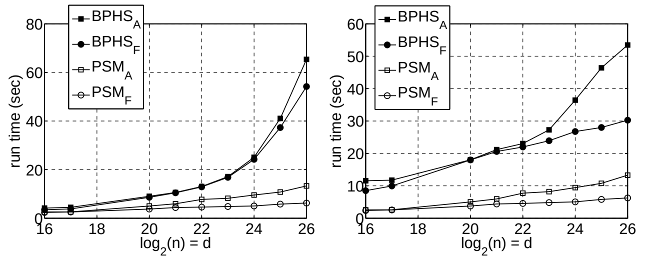
In our experiments, we use two most space-efficient BPHS designs suggested in [16], i.e., $T = 4$ and $T = 10$ tables. To make comparison easier to follow, we convert our RAM overhead to the same notation by dividing Ω' by the size of $\mathcal{H}$, i.e., $nb/8$. Thus, PSM's number of tables becomes:

$$T' = 1 + \frac{1}{2^{d-t}}. \tag{26}$$

By varying t, we can achieve any overhead $T' \in [1, 2]$. We study how this selection impacts the result using the total run-time of each algorithm over the entire set of 10M queries. Table 2 shows our results. The top half of the table focuses on finding *all* matches, where PSM$_A$ is $3.4 - 5$ times faster than BPHS$_A$, while reducing its RAM consumption by a factor of 2-10 depending on the choice of t. In the lower half, PSM$_F$ is even better (i.e., $3.7 - 8.7$ times faster than BPHS$_F$) with the same savings in RAM.

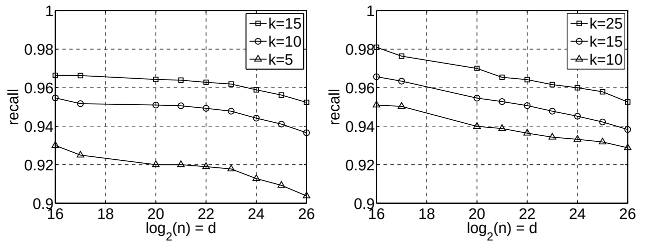
Another interesting observation in the table is that PSM is relatively insensitive to RAM usage. After dropping T' from 2 to 1.06, PSM$_A$ loses only 15% in speed and PSM$_F$ only about 9%. Focusing on the ratio of speed (in thousands of queries/sec) to the number of tables used, PSM$_A$ peaks at 617 with $T' = 1.125$ and PSM$_F$ at 1367 with $T' = 1.06$. The best numbers from BPHS are 38 and 46, respectively.

We next focus on the scalability of each method. In the modeling section, we showed that with a fixed number of tables, BPHS's lookup delay for each query scaled linearly with set size n, which is equivalent to an exponential increase when plotted against $d = \log_2 n$. At the same time, we



(a) $T = 4$ (b) $T = 10$

Figure 7: Scalability with dataset size (online mode, 10M queries, 60M existing hashes).



(a) PSM$_F$ (b) PSM$_A$

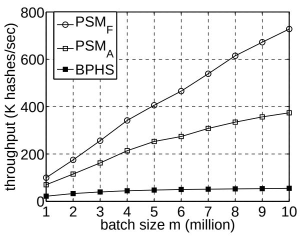
Figure 8: Relative recall in PSM with a fixed number of bit flips (online mode, 10M queries, 60M existing hashes).

showed that PSM's CPU overhead could be crudely upper-bounded by $O(d^h)$, which at least in theory should be significantly better.

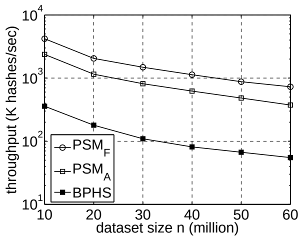
Figure 7 confirms this result in our implementation. Specifically, in part (a) of the figure, we fix $T = 4$ tables for [16] and keep $T' = 2$ in our method. Notice the aggressive increase in delay for both versions of BPHS and an almost linear increase for PSM. In part (b), the BPHS design calls for $g = 2$ and $G = 5$, which means that the exponential term in (14) does not become active until d exceeds $gb/(b+h) = 25.6$. Thus, most of the visible increase in Figure 7(b) is due to the binary search; however, once the dataset becomes substantially larger, these curves will become exponential in d. Note that in part (a) of the figure, the situation was dramatically different because g was 1 and $gb/(b + h)$ was 16.

We finish this section by keeping the number of attempted combinations $k(t)$ constant and examining how recall R' changes with dataset size d. This demonstrates the decay rate of recall as a function of $|\mathcal{H}|$, which might be interesting to applications that intend to keep per-query CPU overhead constant as $n \to \infty$. In these experiments, we scale the number of header bits as $t = d$ and plot the result in Figure 8. First, notice that recall of PSM$_F$ decreases slightly slower than that of PSM$_A$. Second, observe that with just $k = 5$ combinations of bit-flips, the former method achieves 90% recall for all datasets up to 60M pages. The latter technique can maintain 93% recall with just $k = 10$ flips, which implies that by lowering the target R' to 90%, our method can become $2.5 - 3$ times faster than already demonstrated.

In real-time systems with hard memory and performance constraints that are ready to sacrifice a small percentage of recall, these results show that PSM offers a significantly



(a) speed vs m



(b) speed vs n

Figure 9: Throughput in batch mode.

faster and more space-efficient solution than currently available in the literature.

8.5 Batch Mode

In these experiments, the existing dataset of 60M fingerprints is read from disk and matched against a query batch with $m = |\mathcal{Q}| = 10$M fingerprints stored in memory. Since in our system τ_{disk} is at least 20 times smaller than τ_{CPU}, the overall performance is dominated by the throughput of the studied algorithms. While we are not concerned with RAM as much as before, simple calculations show that PSM still uses less RAM than BPHS.

Figure 9(a) shows the effect of batch size m on the processing speed. As discussed earlier, BPHS$_A$ and BPHS$_F$ are the same method in batch mode, which we plot as a single curve in the figure. We use the 4-table design for BPHS since it proved faster than the 10-table version for dataset sizes below 2^{25} (see Figure 7). We now come back to the prediction in the modeling section that BPHS's speed should saturate and remain constant with m, while that of PSM should increase sublinearly, but no worse than $m/\log_2^h(m)\log\log m$.

Figure 9(a) confirms both findings, showing that the processing rate of BPHS stabilizes at 55K/sec, while our technique scales from 99K/sec to 727K/sec for PSM$_F$ and from 70K/sec to 373K/sec for PSM$_A$. At the final batch size (i.e., $m = 10$M), PSM outperforms BPHS by a factor of $7.7 - 14$, which is expected to continue increasing as $m \to \infty$.

It should be noted that even for the largest batch $m =$ 10M, both methods in Figure 9(a) are still 50-70% slower than in online mode. For BPHS, this can be explained by the much larger number of binary searches it performs compared to the online version. For PSM, the model predicts that splitting the dataset into small chunks reduces performance since each new hash must be looked up in n/m hash tables.

We finish the paper by examining how both methods behave when n increases, but m stays fixed. The model shows that the run-time of both techniques should be linear in n, which means that their throughput should be $\Theta(1/n)$. Figure 9(b) confirms this fact and shows that the ratio between the PSM and BPHS curves remains constant at approximately 7 for PSM$_A$ and 14 for PSM$_F$. Combining the various observations, we can conclude that as $n \to \infty$, batch-mode PSM will be able to use larger m and its performance gains over fixed-table BPHS will grow even further.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel way of utilizing `simhash` to find near-duplicates in large collections of documents. We showed that by sacrificing a small percentage of recall the proposed approach consistently outperformed [16] in terms of query speed and space consumption, which it was able to simultaneously lower by a factor that ranged from 2 to 14 in various configurations.

Future work involves analysis of feature-selection techniques for better clustering, improvement of `simhash` recall against cosine similarity, and further overhead reduction in our bit-flipping algorithm.

10. REFERENCES

[1] A. Appleby, "MurmurHash." [Online]. Available: http://sites.google.com/site/murmurhash/.

[2] S. Baluja and M. Covell, "Learning 'Forgiving' Hash Functions: Algorithms and Large Scale Tests," in *Proc. IJCAI*, Jan. 2007, pp. 2663–2669.

[3] M. Bawa, T. Condie, and P. Ganesan, "LSH Forest: SelfTuning Indexes for Similarity Search," in *Proc. WWW*, May 2005, pp. 651–660.

[4] J. L. Bentley, "K-D Trees For Semi-Dynamic Point Sets," in *Proc. ACM Symposium on Computational Geometry (SCG)*, Jun. 1990, pp. 187–197.

[5] A. Broder, "Identifying and Filtering Near-Duplicate Documents," in *Proc. CPM*, Jun. 2000, pp. 1–10.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-Wise Independent Permutations," in *Proc. ACM STOC*, May 1998, pp. 327–336.

[7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic Clustering of the Web," *Computer Networks and ISDN Systems*, vol. 29, no. 8–13, pp. 1157–1166, Sep. 1997.

[8] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in *Proc. ACM STOC*, May 2002, pp. 380–388.

[9] A. Chowdhury, O. Frieder, D. Grossman, and M. C. McCabe, "Collection Statistics for Fast Duplicate Document Detection," *ACM Trans. Inf. Syst.*, vol. 20, no. 2, Apr. 2002.

[10] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang, "Finding Interesting Associations without Support Pruning," in *Proc. IEEE ICDE*, Feb. 2000, pp. 489–500.

[11] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental Clustering for Mining in a Data Warehousing Environment," in *Proc. VLDB*, Aug. 1998, pp. 323–333.

[12] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," in *Proc. ACM SIGMOD*, Jun. 1984, pp. 47–57.

[13] M. R. Henzinger, "Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms," in *Proc. ACM SIGIR*, Aug. 2006, pp. 284–291.

[14] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proc. ACM STOC*, May 1998, pp. 604–613.

[15] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," in *Proc. WWW*, Apr. 2008, pp. 427–436.

[16] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting Near Duplicates for Web Crawling," in *Proc. WWW*, May 2007, pp. 141–149.

[17] M. Minsky and S. Papert, *Perceptrons.* MIT Press, 1969.

[18] R. Salakhutdinov and G. Hinton, "Semantic Hashing," *International Journal of Approximate Reasoning*, vol. 50, no. 7, pp. 969–978, July 2009.

[19] B. Stein, S. M. zu Eissen, and M. Potthast, "Strategies for Retrieving Plagiarized Documents," in *Proc. ACM SIGIR*, Jul. 2007, pp. 825–826.

[20] P. Wegner, "A Technique for Counting Ones in a Binary Computer," *Commun. ACM*, vol. 3, no. 5, p. 322, May 1960.