Detecting the Origin of Text Segments Efficiently

Ossama
Abdel-Hamid[*]
Cairo University
Giza, Egypt

Behshad Behzadi
Google Inc
Zurich, Switzerland

Stefan Christoph
Google Inc
Zurich, Switzerland

Monika Henzinger
Google Inc & EPFL
Lausanne, Switzerland

o.abdelhamid@fci-cu.edu.eg, {behshad, stefanchr, monika}@google.com

ABSTRACT

In the origin detection problem an algorithm is given a set S of documents, ordered by creation time, and a query document D. It needs to output for every consecutive sequence of k alphanumeric terms in D the earliest document in S in which the sequence appeared (if such a document exists). Algorithms for the origin detection problem can, for example, be used to detect the "origin" of text segments in D and thus to detect novel content in D. They can also find the document from which the author of D has copied the most (or show that D is mostly original.)

We propose novel algorithms for this problem and evaluate them together with a large number of previously published algorithms. Our results show that (1) detecting the origin of text segments efficiently can be done with very high accuracy even when the space used is less than 1% of the size of the documents in S, (2) the precision degrades smoothly with the amount of available space, (3) various estimation techniques can be used to increase the performance of the algorithms.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Measurement, Experimentation

Keywords

Document Overlap, Shingling

1. INTRODUCTION

According to web search engines the web consists by now of tens of billions of pages. Not surprisingly most of these pages do not contain novel information but replicate in some way the content of other pages. It is estimated [8, 16, 1] that roughly 20-40% of the pages are exact duplicates. The remaining pages with replication can be roughly categorized into three types [3]: (1) *Near-duplicate pages,* i.e., pages that are almost identical to the original page, except for minor changes, like the insertion or replacement of a few words. (2)

Partial replication, where one or more paragraphs are copied but overall the new document differs significantly from the original document. For example, if a document contains paragraphs from multiple sources then it is a partial replicate of all these documents, but a near-duplicate for none of them. (3) *Semantic duplication,* where pages contain (almost) the same content, but different words. Most attention in the past has been given to finding near-duplicate pages [4, 6, 10, 11, 12, 16, 17]. Recently, attention has shifted towards detecting partial replication [7, 15, 14], but none of the prior work focuses on the origin detection problem.

There are various reasons for this shift. (a) Paragraph replication is very common in blogs. With the increasing popularity of blogs there has also been increasing interest in trying to identify the origin of these copies, for example, in order to point readers of a blog post to the blog post(s) on which the copied text originated. (b) Spammers have become more sophisticated. While earlier spammers copied whole documents and made only minor changes, spammers nowadays are more likely to copy just the most relevant paragraphs (in combination with other spamming techniques) to attract search engine clicks.[1] Detecting partial replication is thus useful for generating a list of potential spammers [9]. (c) As search engines become more sophisticated they try to return better search snippets to the users. One improvement for snippets can be to extract the snippets from the part of the web page that is novel, i.e., that is not copied from an earlier page. (d) Another application would be to build a browser that marks the novel (i.e. not copied) content of the web page it displays. It could also annotate the copied content with the URL of the page where the content originated. A user who repeatedly visits the same page can thus quickly identify which parts of the page have changed. This would be especially interesting for frequently updated pages, like new versions of a news page on a hot topic.

The above applications all assume that we know the correct order in which pages are created. For blogs this is straightforward as blog posts usually have a timestamp. However, for general web pages determining the right creation time is not easy. Web search engines usually keep track of the first time that they have encountered a URL or a new version at a known URL. However, there is no guarantee that the search engine visits a URL soon after its content has changed. Thus it is possible that the content has been partially replicated and that the search engine crawls

---

[*]Work done while visiting Google Inc.

---

[1]A study by Beaza-Yates et al. [1, 2] shows that authors of web pages use search engines to find content that they then copy and paste into their pages.

the page that contains the copy before the origin. In this paper we do not address the problem of correctly ordering web pages (see instead the recent work by Bendersky and Croft [3] on this problem). Here we assume that such an ordering is given and we study how we can efficiently detect partial replication and its origin given such an ordering.

There are two challenges when building a system for detecting partial replication. One is the amount of data to be handled. Clearly, comparing every pair of documents is practically impossible. To avoid this one could build an index of all text fragments found in all documents. However, the size of such an index is proportional to the size of all the documents, which is in the order of hundreds of Terabytes for the whole web. Thus we use a different approach based on selection and hashing that uses a small fraction, e.g., 1%, of the space required for the full index. The second challenge comes from the fact that many documents have to be served very shortly after they were created. News and blogs are two categories of documents of this type. This requires that the system needs to be able to respond in real-time or very close to real time.

Based on these two challenges we are focusing on *space-efficient* techniques for finding the origin of copied text segments in a stream of documents *in real-time*. We design new techniques for addressing them and discuss various design decisions. We then compare our techniques with previously published techniques for this or related problems. The results show that it is indeed possible to significantly decrease the amount of space needed while still generating high-quality results. As in prior art we call a sequence of k consecutive terms in a document a *shingle* [5, 6]. In this paper we focus on the following two problems. (1) Determine all terms in the document that do not belong to a copied shingle. Any maximal sequence of such novel terms is considered to be a novel segment in the document. (2) Determine the document D' where most of the content of a given document D originated. If $D = D'$, then more content is unique to D than copied from any single other source.

Our system uses the following architecture: Every shingle in the document is fingerprinted [13]. Then a *selection algorithm* determines which shingles to store in a hash table. The hash table stores for each shingle some additional information. For selected shingles for which there is a *hit* in the hash table the additional information is sent back to the algorithm. Based on this information we use various *estimation techniques* to determine which shingles are copied and from which document most of the content originated. Since we limit the size of the hash table to be much smaller than the size of all the documents, shingles frequently have to be evicted from the hash table. Thus, we also explore the impact of various *eviction algorithms*.

In summary, this paper contains the following contributions: (1) We present a new shingle selection algorithm called *Hailstorm* for selecting shingles to send to the hash table. We compare it with other existing shingle selection algorithms and show that it outperforms the existing ones. (2) We present a new eviction strategy for the items in the hash table which helps in keeping non-redundant and more useful shingles *("lucky shingles")* for detecting future copying. We compare this new eviction strategy with various known strategies, like LRU. (3) We suggest a new technique called *bridging* for estimating the origin of all selected shingles in a document even though only information about a very small number of shingles in the document is available. (4) We perform an extensive analysis of different algorithms on two real datasets and show that (1),(2) and (3) together provide the best solution for our problem.

The paper is organized as follows: In Section 2 we define formally the problem we are studying and present our system architecture as well as the different algorithms that we are evaluating. In Section 3, we describe our experimental setup and the results on two real-live datasets.

2. ALGORITHMS

We study the following *origin detection problem.* Its input consists of three parts: (1) It contains a set S of sequences of tokens. For example, a sequence of tokens can be a document, consisting of a sequence of terms, or a DNA sequence, consisting of a sequence of nucleotide bases. For simplicity, we refer to a sequence of tokens in S as a *document*, even though all of our techniques apply to the more general setting. Each document in S has a unique timestamp or, more generally, the documents are totally ordered with no ties. We refer to documents that are smaller in this order as being *earlier* and to documents that are larger in this order as being *later.* (2) The "query" part of the input consists of an additional sequence D of tokens, referred to as *query document D.* (3) The last part of the input is a parameter k. We refer to a sub-sequence of k consecutive tokens of a document as a *k-shingle* or *shingle.* The algorithm outputs for *every* shingle s in D the earliest document D' of S that contains s. We call D' the *origin* of shingle s. If $D' \neq D$, we call s a *copied* shingle, otherwise s is a *new* shingle. Note that shingles can overlap in up to $k - 1$ tokens, i.e., they correspond to sliding windows of the document. We say a token is *covered* by a shingle s if the token is one of the tokens in the sub-sequence that forms s.

The origin detection problem can be used to find novel content in documents as follows: Assume that an earlier version D_0 of a document D_1 belongs to S and that D_1 is the query document. If a subsequence T of D_1 is covered only by new shingles then this part of the document did not belong to D_0, otherwise the shingles would have been detected as being copied. Thus T is a novel text segment of D_1.

One algorithm to solve the origin detection problem is to convert every document in S into a set of shingles and to build an inverted index data structure for these shingles. This is the k-gram approach taken by Seo and Croft [15]. As they point out it requires a large amount of space, since every document of n tokens is represented by $n-k+1$ shingles. Seo and Croft studied four different ways of selecting a subset of the shingles reducing the average number of shingles per document by a factor of 6. Their goal was, however, *not* to solve the origin detection problem, but to study how well these selection techniques work to detect partial overlap, considerable overlap, or almost complete overlap between documents.

We want to build a system whose space usage is much smaller. Thus we use the following two-step approach. (1) In the *preprocessing step* we hash the shingles that are selected from the documents in S together with per-shingle information into a fixed-size bucketed hash table. When a bucket becomes full, we evict one of its shingles. (2) In the subsequent *query step* we hash the shingles selected from the query document D into the hash table. When there is a "hit" for a shingle s in the hash table we retrieve the infor-

mation stored for s. We then use the information collected from *all* "hits" to determine (or estimate) the origin of every selected shingle of D. Note that *both* steps require to select and hash shingles for a given document. The query step needs to additionally determine the origin of selected shingles. Thus, we partition our system into three algorithmic components, namely the *selection phase*, the *hashing phase*, and the *estimation phase.* In the selection phase a document D is converted into a sequence of shingles and a subset of shingles is selected. In the hashing phase each selected shingle is hashed into a hash table (in the order of selection), the hash table is updated, and potentially information about the shingle is returned. In the estimation phase the origin of every selected shingles is determined. The preprocessing step applies the selection and the hashing phase to *each* document in S. The query step applies the selection phase, the hashing phase, and in addition the estimation phase to the query document D.

To summarize, our main algorithmic differences to the work by Seo and Croft [15] are: (a) We assume that a fixed amount of space is given for solving the problem and we study how different algorithms work for different memory sizes. Note that fixing the amount of space is often preferable in the deployment of systems as increasing the size of the input set S does not lead to an increased space requirement, but instead to a (hopefully smooth) degradation of the output quality. To deal with the fixed space we experiment with various eviction strategies. (b) We study more shingle selection algorithms, namely all the algorithms that Seo and Croft evaluated (including *Winnowing* [14]), and additionally some new ones that we specifically designed to work well for our problem setup. (c) We add an "estimation phase" where we use information about other shingles in D to estimate the origin of shingles missing in the hash table.

2.1 Selection Phase

In the selection phase a given document is converted into a set of selected shingles. For two of the selection algorithms described below (Hash-breaking and DCT) a given document is actually converted into a set of fingerprints that do *not* correspond to shingles in the document as defined above. However, to simplify our notation we call these fingerprints as well *selected shingles*. For the other algorithms the set of selected shingles is created as follows: (i) All the shingles of D are generated and each shingle is converted into a 62-bit *fingerprint*[2]. (ii) Next a subset of shingles is selected, often based on their fingerprints. In the following we do not distinguish between a shingle and its fingerprint, i.e., we assume the internal representation of a shingle is its fingerprint. We describe next all our selection algorithms in detail.

All. Our baseline algorithm selects all shingles.

Every l-th (lth). For a given integer l select every l-th shingle, i.e., select only the shingles that start at position $0, l, 2l, 3l, \ldots$ in the document. This approach selects a $1/l$-fraction of the shingles. We used $l = 4, 6,$ and 8.

Modulo l (M-l). For a given integer l select a shingle iff its value is divisible by l. With Rabin's fingerprints the probability for a shingle to be divisible by l is $1/l$. Thus, this

approach expects to select a $1/l$-fraction of the shingles. We experimented with $l = 2, 3, \ldots, 8$.

Winnowing w [14] (W-w). Winnowing uses a second window size w and for each consecutive sequence of w shingles it outputs the shingle with the smallest fingerprint value. If there is no unique smallest shingle, the right-most smallest shingle is selected. Schleimer et al [14] showed that the expected number of shingles selected is a $\frac{2}{w+1}$-fraction of all the shingles. We experimented with $w = 5, 6, \ldots, 9$.

Revised Hash-breaking [15] (Hb-p). The original hash-breaking algorithm [5] first applies a hash function h to each token and then breaks the document into non-overlapping *(text) segments* at the tokens whose hash value is divisible by some fixed integer p. Then it fingerprints all the tokens that are contained in the segment. The expected number of fingerprints is a $1/p$-fraction of the shingles for longer documents. Seo and Croft [15] proposed a revised version of hash-breaking where segments of length less than p are ignored, removing "noisy" segments. Recall that we call the fingerprints of the segments *selected shingles* by abuse of notation. We experimented with $p = 3, 4, \ldots, 8$.

DCT [15] (Dct-p). DCT first breaks the document into non-overlapping text segments in the same way as hash-breaking. Next it constructs an integer representation of each text segment as follows: It treats the sequence of fingerprints of the text segments as a discrete time domain signal sequence and after normalizing the values applies a discrete cosine transform to it. The resulting coefficients are quantized and used to construct one 64-bit fingerprint for the text segment. The number of fingerprints equals the number of fingerprints for hash-breaking. The fingerprints produced by DCT are tolerant to small changes in the segment. Recall that we call these fingerprints *selected shingles* by abuse of notation. We experimented with $p = 3, 4, \ldots, 8$.

Furthermore, we propose the following new algorithm:

Hailstorm (Hs). The algorithm first fingerprints every token and then selects a shingle s iff the minimum fingerprint value of all k tokens in s occurs at the first or the last position of s (and potentially also in-between.) Due to the probabilistic properties of Rabin's fingerprints the probability that a shingle is chosen is $\frac{2}{k}$ if all tokens in the shingle are different.

Winnowing fulfills this *locality* property [14]: Whether a shingle s is selected or not only depends on the shingles in the same windows as s and *not* on other shingles. Algorithms Modulo and Hailstorm fulfill an even stronger property, called *context-freeness:* Whether a shingle is selected or not only depends on the shingle itself and *not* on any other shingle in the document, i.e., any given shingle s is either selected in all documents containing it or is never selected. Furthermore, Winnowing guarantees that in each sequence of $w + k - 1$ tokens at least one shingle is selected, i.e., it is not possible that no shingle is selected in an arbitrarily long sequence of tokens. Note that this could happen with Algorithm Modulo. Hailstorm gives an even stronger guarantee: Lemma 1 proves that for every token at least one shingle covering the token is selected. Thus, all tokens are covered by the selected shingles. We call this *total coverage.* Algorithm Every l-th also has this property as long as $l \leq k$. However, Hailstorm is the only shingle-based selection algorithm that is both context-free and guarantees total coverage.

LEMMA 1. *In every document D, any token (except for the first or last $k-1$ tokens) is covered by at least one k-shingle selected by Algorithm Hailstorm.*

---

[2]Converting a string of characters into a single integer such that the probability of a collision is small is called *fingerprinting.* We use Rabin's fingerprints [13] throughout the paper.

PROOF. For a position p in D consider the interval of $2k-1$ tokens centered around p and let x be the position of the smallest token in this interval. If $x \geq p$ (resp. $x \leq p$) then the k-shingle terminating (resp. starting) at x is selected and it covers p.   □

All selected shingles are sent to the hash table. Thus, the amount of data sent to the hash table varies between different algorithms and different parameter settings.

For a real-time system the amount of data sent to the hash table (e.g. via rpc calls) dominates its running time. Thus we want to reduce the amount of data sent as much as possible without hurting the quality of the result. Note that there is a certain redundancy in the data sent to the hash table. Specifically, a shingle might be selected even though all its tokens are already covered by other selected shingles. Thus, we also propose and evaluate a "no complete overlap" version of the above algorithms that does not send these redundant shingles. It first selects all shingles, but before sending them to the hash table it makes a second pass over them and discards the selected shingles all of whose tokens are covered by selected, not-discarded shingles. This significantly reduces the data sent to the hash tables. It also reduces evictions from the hash tables and usually improves the quality of the results. Applying this technique to the algorithms M-p, W-w, and Hs creates the algorithms NM-p, NW-w, and NHs. Note that *Every 8-th* is a no-complete-overlap variant of *Every-4th* and *All*. There is no overlap in HB-p and DCT-p.

2.2 Hashing Phase

Every selected shingle s is hashed into a fixed-sized hash table. The hash table is split into buckets, each storing the information for 64 shingles (see Section 3.5 for a discussion of bucket sizes).

For each shingle in the hash table its value plus some additional information described below are stored in the bucket. Each selected shingle s is hashed into one bucket, which is then searched to see whether it contains s. If s is found, part of the information stored with s in the hash table is updated, another part is sent back to be used in the estimation phase. When s is not found in its bucket, it is now inserted into the bucket together with all the necessary information, which was sent together with s to the hash table. If the bucket is already storing 64 shingles, then one of the shingles in the bucket is evicted to make space for s. We describe below different eviction policies.

The hash table is the main data structure used by the algorithm and dominates its memory usage. We want to understand the impact of different sizes of the hash table on the performance of various algorithms. Thus we experimented with eight different values for the hash table size m between $20MB$ and $5GB$. The space needed to store a shingle s and its information varies between 14 bytes and 18 bytes depending on which eviction algorithm and which estimation algorithm we use. Since the number of shingles per bucket is fixed, the number of buckets in the hash table varies between different algorithms and different hash table sizes. Depending on the number of buckets the first 14 or more bits of a shingle are used to hash a shingle.

The per shingle information can contain up to five parts, namely (1) the fingerprint of s itself, (2) its origin D_s, (3) its offset in D_2 (see below), and (4) information about neighboring selected shingles in D_s, and (5) information for the eviction algorithm. We now explain them in detail. (1) The fingerprint of s consists of 62 bits. Since at least the first 14 bits are encoded in the bucket index (no matter how many buckets there are), we only need 6 bytes to store the shingle itself[3]. (2) We use 8 bytes to store (a fingerprint of) its origin D_s. When a new shingle is inserted into a bucket, its origin is the document that we are currently processing. (3) Some of the algorithms in the estimation phase need the offset of s in D_s. The *offset* of a shingle s in a document D is the number of shingles that were selected in D before s in the shingle order of D. If the number is at least 2^8, we compute its value modulo 2^8, i.e., we use a one byte for the offset. (4) Some algorithms also use the first byte of the fingerprint of the selected shingle immediately preceding s and the first byte of the fingerprint of the selected shingle immediately succeeding s in D_s. Thus for these algorithms we keep these two bytes. (5) For the *Copy-Count* and the *Lucky* eviction algorithm we use one additional byte, while for the *Random* and *LRU* eviction algorithm, no additional byte is used. The use of this byte is explained in detail in the description of the eviction algorithms.

We study three classic eviction algorithms together with *Lucky*, a novel eviction algorithm that we propose:

Random (R). Evict a random shingle from the bucket.

LRU. Each hash table bucket is treated as a queue of shingles. When a new shingle is inserted in the bucket it is placed at the end of the queue. Every time a shingle is searched for and found in the bucket, it is placed at the end of the queue. When a shingle needs to be evicted, the shingle at the front of the queue is evicted.

Copy-Count (CC). When a new shingle is inserted into the bucket, its copy-count is set to one. Every time the shingle is searched for and found in the hash table the copy-count is incremented by 1. When a shingle needs to be evicted, the shingle with the smallest copy-count is evicted. When the counters for 10 shingles in a bucket reach 255, we divide all the counters by two.

Lucky (LS). We keep a 1-byte *lucky score*, which is a weighted variant of the copy count, but with larger increments for "more important" shingles. The shingle with the smallest lucky score is evicted. For increments the idea is as follows: Assume a text segment R of shingles is copied from one document to another. The *bridging* technique in the estimation phase (described below) will allow us to guess that all of R was copied if we find the first and last shingle of R in the hash table. Thus we want the first and the last shingle of R to stay in the hash table and, hence, we increment the lucky score of them by more than 1. We also give some higher lucky score to the first and last shingle in the document since they have a good chance of becoming the first, resp., last shingle of some copied segment. Finally, we also slightly increase the lucky score of every y-th selected shingle in the document. Thus no matter which segment of the document is copied by later documents, there is a chance that a selected shingle not "far away" from the boundary of the copied segment is stored in the hash table. Now we describe the lucky algorithm more formally. We first hash

---

[3]When we need more than 14 bits to encode the bucket index, we could have stored fewer than 6 bytes per shingle. However, in the best case this would have increased the number of shingles stored in the hash table by at most 8%, which would have had no measurable effect on the performance (as our data shows).

all selected shingles of a document D to the hash table. Based on the shingle information stored in the hash table and various heuristics (described in the estimation phase) we determine all maximal sub-sequences of selected shingles of D that have been copied from the same origin. They are called *copied block*. We then update the lucky score of the selected shingles of D that are in the hash table or are inserted into it using the following rules: (a) The lucky score of every new shingle is set to 1, the lucky score of every other shingle is incremented by 1. (b) We additionally increase the lucky score of the first and the last shingles of a copied block by $\lfloor \sqrt{b-2} \rfloor$, where b is the size of the block. (c) If a shingle is the first or the last shingle of its document its lucky score is additionally incremented by 3. (d) For every y-th selected shingle in D the lucky score is additionally increased by 1. We set $y = 7$. (e) Whenever the average lucky score of all shingles in a bucket reaches some fixed limit, we divide all the lucky score fields by 2. This is necessary to make sure that the numbers do not become too large. After some preliminary experiments we set the limit to 11.

2.3 Estimation Phase

The input to the estimation algorithm is the information retrieved from the hash table (if available) for *all* selected shingles from the query document D. The estimation algorithm outputs (a guess of) the origin for *each* selected shingle (whether it was found in the hash table or not) and nothing for not-selected shingles. We present four estimation algorithms. The baseline algorithm *No Bridging (NB)* only outputs origins for selected shingles that are *found* in the hash table. For them it reports their origin stored in the hash table. For all the other selected shingles it reports the current document as its origin. The other three estimation algorithms use additional information retrieved from the hash table to guess the origin of selected shingles that were not found in the hash table. If guessing is not possible, they also output the current document as origin.

Our second algorithm is called *Expansion (E)*. It uses the information about neighboring selected shingles. When a selected shingle s from D is found in the hash table, the algorithm compares the first byte of the immediately preceding and succeeding selected shingle of s in D with the corresponding information stored in the hash table. Whenever a match is found, the origin of the shingle s' with the match is set to be the origin of s if s' was not itself found in the hash table. Thus it is possible that we set the origin for up to two additional selected shingles even if they are not in the hash table.

The *Bridging Algorithm (B)* uses the offset of each shingle. Let s and s' be two selected shingles of D such that (a) the offset of s in D is less than the offset of s', (b) for both shingles the same origin was stored in the hash table, (c) the difference of their offsets in D is equal to the difference of their offset (in the origin) stored in the hash table and it is smaller than a given limit T, and (d) none of the shingles that occur after s and before s' in D fulfill all (a), (b) and (c). When all these conditions hold, we assume that the whole block between s and s' was copied from the (common) origin of s and s' and thus we label all selected shingles between s and s' as having the same origin as s and s'. We say that we are *bridging* between s and s'. Note that the endpoint of one bridge can be the start point of the next bridge. Thus two bridges of length 50 can "bridge" the same

shingles as one bridge of length 100. We set $T = 30$, but we also experimented with different values of T, see Section 3. Given all selected shingles of D, their offsets and their origins (if any), *all* bridges of D can be found in time linear in the number of selected shingles using a hashing-based approach.
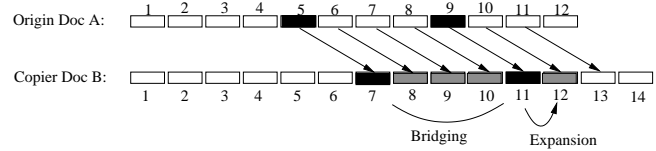

Figure 1: Bridging and Expansion

Figure 1 illustrates the bridging and expansion concepts. It shows the shingles of two documents A and B. Document B is created later and copies 7 shingles from A ($A_5...A_{11}$, indicated by arrows). Suppose that while processing document B we only find the two black copied shingles in the hash table: B_7 as a copy of A_5 and B_{11} as a copy of A_9. If any of the shingles B_8 to B_{10} are selected, bridging would predict correctly that they are copied from the same origin A, even though they are not in the hash table. Furthermore, B_8, B_{10} and B_{12} could also be detected as copies based on expansion. Copied shingle B_{13} remains undiscovered.

Our fourth variant is called *Bridging with Expansion (BE)*. It uses the offset *and* the information about neighboring selected shingles. We bridge between two shingle s and s' iff conditions (a) - (d) from bridging *and* these two additional conditions hold: (e) the first byte of the selected shingle immediately succeeding s in D matches the corresponding information in the hash table and (f) the first byte of the selected shingle immediately preceding s' in D matches the corresponding information in the hash table. Additionally BE applies the expansion techniques. Note that the shingle origins that the refined bridging heuristic and the expansion heuristic output never contradict each other.

3. EVALUATION

3.1 Description of the data sets

We evaluated our algorithms on two data sets, namely on a collection of 8.6 million blogs, called the *blogs data set,* and on a collection of 1.3 million German web pages from the .ch domain, called the *Swiss data set*. The blogs data set is ordered according to the time when the blog was posted. These blogs were collected over 10 days in January 2006, at a rate of 866,000 blogs per day. The Swiss data set consists of web pages crawled in August 2008 and is in crawl order. All these documents contain at least 45 tokens, i.e., alphanumeric sequences. We omit smaller documents because we observed that frequently their main content is not their text but a picture or a video that is contained or can be downloaded from the document. The documents in the blogs data set consist only of the actual user posts and do not contain the frame information that appears around the posts. In the Swiss data set frames were not removed, but all the HTML meta data was removed.

We experimented with shingle size $k = 6$ and $k = 8$ and achieved slightly better results for 8. Thus we only report results for $k = 8$. This corresponds to phrases of 8 terms and thus allows for a fairly fine-grain origin detection.

Recall that in the origin detection problem there is an initial set S of documents and a query document D We describe now how we constructed for each of the above data sets an initial set S and a *set* of query documents. To determine the origin of every shingle we first ran our system without shingle selection and with enough memory to hold all shingles. For each query document D, the *most copied origin* is a document D' that is the origin for the largest number of shingles in D. If the number of shingles in D with most copied origin D' is 10% larger than the number of shingles for any other individual origin, then we call D' the *dominant origin*. One of our evaluation metrics is to measure for how many documents we correctly identify the dominant origin. The metric uses the dominant origin and *not* the most copied origin, since the most copied origin is not necessarily unique and the dominant origin is more stable against small changes in the algorithms. Not all documents have a dominant origin but in our data sets most do, namely 94% of the documents in the blogs data set and 92% in the Swiss data set.

With this metric in mind we constructed for each of the two data sets the initial set S and the set of query documents as follows. (1) We determined the last 100k documents *that have a dominant origin.* They form the *set of query documents.* (2) The initial set S is formed by all documents in the data set (no matter whether they have a dominant origin or not) that occur before *all* of the query documents.

data set	# of docu-ments	# of shingles	avg # of shingles per doc	with dom. origin	avg size of cop. blocks	shingle copy ratio
blogs	8,666,731	1.71 bil.	197	94%	17	0.36
Swiss	1,360,393	0.78 bil.	570	92%	13	0.16

Table 1: Various statistics of our data sets.

Table 1 gives a few statistics about the two data sets, showing that the average number of shingles per document is almost a factor of three larger for the Swiss data set than for the blogs data set. This confirms the intuition that blogs posts are usually small. The table also gives the *shingle copy ratio*, the ratio of the number of copied shingles by the total number of shingles. It is 36% for the blogs data set and 16% in the Swiss data set. Recall that we call a maximum sequences of selected shingles with the same origin a *copied block.* In the blogs data set the average number of tokens in a copied block is 17, in the Swiss data set it is 13. [4] All this data shows that in blogs more and longer text segments are copied than on the web in general.

As mentioned before we evaluated the algorithms on eight different hash table sizes m between 5GB and 20MB. Table 2 shows for each size the percentage of all shingles that fit into the hash table at one time. It varies between 34% for 5GB and 0.1% for 20MB on the blogs data set and between 58% and 0.2% on the Swiss data set. Thus $m = 100MB$ corresponds to storing about 1% of the space required for all documents in S.

---

[4]These numbers are sightly inflated by the fact that we cannot identify text segments with less than $k = 8$ tokens as copied blocks.

m	5000	2000	1000	500	200	100	50	20
Blogs	34.2	13.7	6.8	3.3	1.4	0.7	0.3	0.1
Swiss	57.5	23.0	11.5	5.8	2.3	1.2	0.6	0.2

Table 2: Different hash table sizes m in MB and the percentage of all shingles that fit into the hash table at one time (in percent) for both datasets.

3.2 Evaluation metric

We used three metrics to evaluate our algorithms. The first metric, *selected shingles ratio (ssr),* is the percentage of all shingles that are selected. Since only the selected shingles are sent to the hash table, *ssr* measures the data transfer to the hash table. We use *ssr* as a measure of the total traffic between the hash table and the algorithm even though it only measures the amount of data sent *to* the hash table and not the amount of data returned *from* the hash table because the latter is negligible in comparison to the former. The reason is that (a) the "hit ratio" in the hash table is only 2-3%, i.e., we return data for only 2-3% of the shingles that are sent to the hash table and (2) the data returned per shingle consists of up to 11 bytes (its origin and up to three additional bytes) and thus it is not much larger than the data sent to the hash table which contains as a minimum the 8-byte fingerprint of the shingle. Since we want to build a real-time system it is important that *ssr* is small enough: The time for transferring the data by far dominates the running time of the system. Unlike the metrics below *ssr* does not measure the *quality* of the output. Thus any algorithms with small enough *ssr* to allow real-time processing is suitable, i.e., we selected as "best" algorithm the algorithm with highest-quality output out of all algorithms with sufficiently small *ssr*. After some experimentation with our system we decided that *ssr* should be no larger than 25%. However, this threshold can be different for different system.

Our other two metrics measure the *result quality* of the various algorithms. The *dominant origin (DO)* metric measures the percentage of query documents for which the algorithm correctly identifies the dominant origin. This models the application where we point the reader of a blog to the blog from which most of the copying happened. If most of the shingles of a document are not found in the hash table, then the document itself is its own dominant origin. For 62% of the query documents in the blogs data set and for 69% in the Swiss data the query document is its own dominant origin. Thus a trivial baseline algorithm that always outputs the document itself as its dominant origin would achieve a DO value of 62%, resp. 69%. For evaluating an algorithm we computed the DO performance for eight different hash table sizes m, but for space reasons we usually omit a few. We also give the *average DO* score, i.e. the average DO value over *all eight* different values of m.

The third metric measures how well we can determine "new" content in a document. We call a token in a query document D a *fresh* token if *none* of the shingles it belongs to are found in the hash table when D is processed. A token that is not fresh is called *old.* After determining the origin of every selected shingle in D the estimation algorithm makes a second pass over D and outputs either *fresh* or *old* for every token. Note that *fresh* does not mean that the token has never occurred in any of the preceding documents, only that it has not be seen *in this context* in the preceding

documents. Using a system with no selection and sufficient memory to store all shingles we determined the correct label for each token. For the query documents in the blogs data set 57% of the tokens are fresh, for the Swiss data set 61% of the tokens are fresh[5]. The *token freshness (TF)* metric measures the percentage of tokens out of all tokens that are correctly labeled as fresh or old. Thus, a trivial baseline that outputs fresh for every token would achieve a TF value of 57%, resp. 61%, on the two data sets. We also give the *average TF* score, i.e. the TF metric averaged over *all eight* different hash table sizes.

To simplify the comparison of algorithms we also compute the arithmetic mean of the average DO score and the average TF score of an algorithm and call it the *overall score*.

3.3 Overview of the experiments

We ran two versions of each selection algorithm, one with random eviction and no estimation, i.e. the baseline algorithm NB, and one with lucky eviction and BE estimation. Based on the results for the selection algorithms, we chose the best performing selection algorithm and All as a "no-selection" baseline. For these two selection algorithms we then experimented with the four different eviction algorithms without any bridging or expansion. In a third set of experiments we then evaluated the same two selection algorithms and the best two eviction algorithms with the four different estimation algorithms.

3.4 Comparison of the selection algorithms

We have seven types of selection algorithms, some with various choices of parameters and some with a no-complete-overlap variant. This results in 42 different selection algorithms. Table 3 lists the *ssr* value in the blogs data set[6] for each of the 33 selection variants with sufficiently small *ssr* value (i.e. *ssr* ≤ 25%) which we evaluated in our further experiments. We also evaluated Algorithm All as a baseline algorithm, even though its *ssr* is 100%. Table 3 shows how changing the parameter for each algorithm changes its *ssr* value. It also shows that the no-complete-overlap versions have a significantly smaller *ssr* value than their original counterparts, e.g., the *ssr* of W8 is 23.5%, while the *ssr* of NW8 is only 17%.

Every l-th			Modulo l			
4th	6th	8th	NM2	NM3	M4	NM4
25%	17%	13%	15%	15%	25%	14%
Modulo l						
M5	NM5	M6	NM6	M7	NM7	M8
20%	14%	16%	12.5%	14%	11%	12%
Winnowing w						
NW5	NW6	NW7	W8	NW8	W9	NW9
16%	17%	16.5%	23.5%	17%	21%	16%
Dct p (same ssr as HB-p)					Hailstorm	M-l
Dct3	Dct4	Dct5	Dct6	Dct8	NHs	NM8
20%	15%	12%	10%	8%	17%	10%

Table 3: Percentage of shingles sent to the hash table by the selection algorithms in the blogs data set.

We ran each of these 33 selection algorithms together with two versions for the eviction and estimate phases, namely

---

[5]We believe that the fact that frames are not removed from the Swiss data set is responsible for this relatively small percentage of fresh tokens in the Swiss data set.

[6]The *ssr* values for the Swiss data set are almost identical.

with random eviction and no estimation (*Version A*), and with lucky eviction and BE estimation (*Version B*). We chose these two versions as they represent the two extreme cases of eviction and estimation. Version A assumes no further intelligence is used, while Version B is our best combination of eviction and estimation algorithms (see below.) Running all algorithms with these two versions resulted in 66 average DO and 66 average TF scores. We then picked in each type of selection algorithm the variant that performed best, based on the *overall score*. We chose according to the overall score as we are interested in both metrics and there is no algorithm that performs best in both metrics. Interestingly, for each type of algorithm the same parameter choices performed best on *both* data sets. These are NHs, 4th, NW8, NM3, Hb3, and Dct8. We call them the *(overall) best performing selection algorithms* and report their DO performance on the blogs data set in Table 4 and 5.

m	All	NHs	4th	NW8	NM3	Hb3	Dct8
5000	83.2	**98.8**	89.5	98.2	96.7	96.9	90.0
2000	79.8	**97.3**	83.2	96.2	95.7	95.6	90.0
1000	77.2	84.6	78.5	84.3	85.1	84.6	**89.5**
500	75.7	79.8	76.5	79.5	79.4	79.3	**82.8**
200	74.3	77.5	75.1	77.3	77.3	77.1	**77.6**
100	73.6	75.8	74.0	75.7	75.7	75.6	**76.3**
50	73.2	74.7	73.4	74.6	74.6	74.6	**75.3**
20	72.9	73.7	72.8	73.6	73.7	73.6	**74.0**
AvgDO	76.2	**82.8**	77.9	82.4	82.3	82.2	81.9

Table 4: Blogs data set: the DO score (in %) of the best performing selection algorithms for different hash table sizes (in MB) when combined with random eviction and no estimation. The maximum in each row is highlighted.

m	All	NHs	4th	NW8	NM3	Hb3	Dct8
5000	**99.2**	98.5	88.9	98.0	96.3	96.8	85.1
500	90.6	**93.7**	86.0	92.9	92.0	91.8	85.1
50	79.7	**84.3**	79.6	83.5	83.1	82.5	82.8
AvgDO	88.0	**91.0**	84.0	90.2	89.3	89.2	84.1

Table 5: Blogs data set: the DO score (in %) of the best performing selection algorithms for different hash table sizes (in MB) when combined with lucky eviction and BE estimation.

Based on this data we can draw four conclusions *for the blogs data set*: (1) The DO performance gradually decreases with m (more or less smoothly depending on the algorithm). (2) Version B clearly outperforms Version A. (3) The no-complete-overlap version of an algorithm usual has a higher DO performance than the original version. The only exception is Every-4th that slightly outperforms Every-8th. (4) Smart selection clearly helps. Even though some selection algorithms select many more shingles, they actually perform worse in the DO metric. The most extreme case is Algorithm All which is the worst algorithm in combination with Version A, even though it selects all of its shingles. However, also Algorithm W5 (not shown in the table) with an ssr value of 34% performs quite poorly with an average DO value of 80.0% in combination with Version A. (5) Out of the overall best performing Algorithms NHs has the highest average DO score in both versions. Additionally, for almost all memory sizes its performance is close to the top performance for that memory size, i.e., it adjusts very well to different hash

table sizes. [7] (6) Algorithm Dct8 performs well in Version A, but not in Version B. While it performs well for small m, its performance does not increase with m as soon as $m \geq$ 200MB resulting is a very poor performance for $m =$5GB/ Other variants of Dct exhibit a similar behavior.

Swiss data set. Conclusions (1)-(5) hold also for the Swiss data set (details omitted for space reasons) with the following slight adjustments: The performance of NHs is slightly lower (80.5%) for Version A and slightly higher (92.0%) for Version B. In Version A, NM7 outperforms NHs with a DO score of 81.7%, mostly because it performs very well for $m = 500$MB. In Version B, Algorithm NHs has the top DO score. Conclusion (6) does not hold on the Swiss data set: All types of Dct performed poorly for all memory sizes. Thus we conclude that the results of DCT for our problem setting are inconclusive and deserve further study.

Next we analyze the token freshness metric. Table 6 shows the results for the overall best performing algorithms for the blogs data set. We draw the following conclusions from them and the omitted data for the Swiss data set: (1) As with the DO metric the token freshness decreases gradually as m decreases. (2) Version B slightly outperforms Version A. (3) Algorithm All performs best. Algorithm NHs performs second best, it is the algorithm with highest average TF score for all algorithms with $ssr \leq 25\%$.

m	All	NHs	4th	NW8	NM3	Hb3	Dct8
Version A: Random Eviction + No Estimation							
5000	**97.7**	93.7	89.7	93.4	91.6	90.7	87.5
500	**86.9**	86.2	82.7	86.2	84.9	84.8	84.4
50	**78.2**	77.2	75.0	77.3	76.5	77.1	77.3
AvgTF	**85.7**	84.5	81.1	84.3	83.2	83.1	82.5
Version B: Lucky Eviction + BE Bridging							
5000	**98.4**	93.6	89.7	93.5	91.7	90.8	86.2
500	**90.5**	89.6	85.1	89.4	88.1	88.1	85.9
50	81.4	81.6	77.9	81.4	80.5	81.6	**82.7**
AvgTF	**87.9**	87.2	83.2	87.0	85.8	86.0	84.6

Table 6: Blogs data set: the TF score (in %) of the best performing selection algorithms for different hash table sizes (in MB).

Data& Version	All	NHs	4th	NW8	NM3	Hb3	Dct8
Blogs A	81.0	**83.6**	79.5	83.4	82.7	82.6	82.2
Blogs B	87.9	**89.1**	83.6	88.7	87.6	87.6	84.3
Swiss A	80.9	81.6	76.0	81.4	**81.8**	81.1	79.3
Swiss B	90.6	**88.0**	79.2	87.6	87.7	86.7	59.7

Table 7: The overall score, i.e., the mean of the average DO and the average TF metric (in percent) on both data sets and both versions for the best performing selection algorithms. The maximum in each row (ignoring All) is highlighted.

Table 7 gives the overall score for the best performing algorithms and for the baseline algorithm All for both data sets and both versions. Ignoring All (which has an ssr value of 100%) NHs performs best for three combinations and is close to the best algorithm (NM8) in the fourth combination.

Thus we use NHs in our experimentation for the eviction and the estimation phase. Algorithm NW8 is the second best algorithm.[8] It is interesting to note that the best performing algorithms, i.e., the algorithms with highest overall score are all "no-overlap" algorithms. The corresponding "overlap" algorithms select more shingles, which always results in a higher TF score and in most cases results also in a lower DO score. For example, Hs with Version B on the blogs data set has an average DO score of 89.8% and a TF score of 87.6%, resulting in an overall score of 88.7%, while the corresponding numbers of NHs are 91.0%, 87.2%, and 89.1%.

3.5 Comparison of eviction algorithms

For the best performing selection algorithm NHs and for the baseline algorithm All we compared the performance of all four eviction algorithms *without* any estimation, i.e. with the estimation algorithm NB. Table 8 gives the results for the DO metric and the TF metric on the blogs data set. It shows that copy count gives the best DO performance when combined with either NHs or All. For TF the picture is mixed. With Algorithm All lucky eviction works best, with NHs LRU is either best or close to best.

m	All				NHs			
	R	LRU	CC	LS	R	LRU	CC	LS
Dominant Origin Metric								
5000	83.2	93.0	**96.1**	95.0	98.8	98.8	98.8	98.8
500	75.7	77.6	**80.9**	80.7	79.8	88.2	**93.1**	87.5
50	73.2	73.4	72.6	**73.5**	74.7	75.9	**78.5**	77.8
Avg	76.2	78.9	**81.1**	80.4	82.8	86.0	**88.3**	86.7
Token Freshness Metric								
5000	97.6	96.1	96.8	**98.6**	93.5	93.5	93.5	93.5
500	86.9	83.8	84.1	**89.6**	86.2	88.1	**88.8**	86.0
50	78.2	76.7	71.8	**79.8**	77.2	**77.9**	75.1	77.1
Avg	85.7	83.7	81.8	**87.7**	84.3	**85.3**	84.2	84.4
Overall Score								
	80.7	81.3	81.4	**84.1**	83.5	85.6	86.2	**85.5**

Table 8: Blogs data set: the DO and TF score (in %) of all eviction algorithms for the selection algorithms All and NHs with no estimation algorithm for different hash table sizes (in MB).

When we average the DO and the TF performance CC performs best with NHs, and LS performs best with All. It is not surprising that CC performs well when no bridging is used, since lucky eviction was specifically designed to work with bridging and is not expected to work very well without bridging. Algorithms CC and LS are also the best two eviction algorithms for the Swiss data set. Thus, we decided to use the two best eviction algorithms CC and LS for the experiments with different bridging variants.

Each bucket in the hash table keeps 64 shingles. To study the impact of this choice we varied the number of shingles per bucket to be 4, 8, 16, 32, 128, and 256, *without* changing the amount of data we store per shingle. Thus the total number of shingles stored in the hash table did not change, only the bucketing and thus the evictions changed. As a result we only saw a small impact on the DO and the TF metric, less than 0.5 percentage points.

3.6 Comparison of estimation algorithms

---

[7]Algorithm NM8 (not shown) has the highest DO score in both data sets, namely 84.9% in Version A and 91.0% (tied with NHs) in Version B. However its TF score is much lower (75.9% resp. 77.4%) so that it has a lower overall score.

[8]The conclusions we draw in Sections 3.5 and 3.6 would not change if we used NW8 instead of NHs.

For the best performing selection algorithm NHs and the "no-selection" baseline algorithm All combined with the two best performing eviction algorithms CC and LS we experimented with all four different estimation algorithms. Table 9 and Table 10 give the results on the blogs data set. From this data we draw the following conclusions: (1) When used with lucky eviction both expansion and bridging give a considerable improvement when used alone, but they work best in combination. (2) When using CC eviction all estimation algorithms perform roughly the same. (3) Algorithm BE even helps when $m = 5GB$, i.e., when 34-58% of the shingles can be kept. When combined with the "no-selection" algorithm All it achieves an almost perfect result of 98.4% DO score and 99.2% TF score. The results for the Swiss data set (omitted) confirm these conclusions.

m	NHs + CC				NHs + LS			
	NB	E	B	BE	NB	E	B	BE
Dominant Origin Metric								
5000	**98.8**	**98.8**	97.4	98.5	**98.8**	**98.8**	97.4	98.5
500	93.1	93.1	92.1	92.7	87.5	92.3	92.6	**93.7**
50	78.5	78.8	77.9	79.2	77.8	82.4	83.0	**84.3**
Avg	88.3	88.7	87.6	88.4	86.7	89.7	89.7	**91.0**
Token Freshness Metric								
5000	93.5	93.5	**93.6**	**93.6**	93.5	93.5	**93.6**	93.6
500	88.8	88.6	89.3	88.6	86.0	88.4	89.5	**89.6**
50	75.1	75.5	76.4	75.8	77.1	80.0	81.5	**81.6**
Avg	84.2	84.2	85.0	84.5	84.4	86.2	87.1	**87.2**
Overall Score								
	86.2	86.4	86.3	86.5	85.5	87.9	88.4	**89.1**

Table 9: Blogs data set: the DO and TF score (in %) of all bridging algorithms with the selection algorithm NHs and the eviction algorithms CC and LS, for different hash table sizes m (in MB).

m	ALL + CC				ALL + LS			
	NB	E	B	BE	NB	E	B	BE
Dominant Origin Metric								
5000	96.8	96.6	96.6	96.4	**98.6**	98.5	98.4	98.4
500	84.1	84.0	84.4	83.8	89.6	**90.5**	90.3	**90.5**
50	71.8	71.9	72.0	71.9	79.8	81.3	81.2	**81.4**
Avg	81.8	81.8	81.9	81.7	87.7	88.5	88.4	**88.6**
Token Freshness Metric								
5000	96.1	96.6	95.2	96.1	95.0	**99.5**	97.9	99.2
500	80.9	82.9	82.1	82.3	80.7	90.2	89.2	**90.6**
50	72.6	75.5	74.4	75.3	73.4	78.9	78.2	**79.7**
Avg	81.0	82.7	81.5	82.3	80.3	87.7	86.6	**87.9**
Overall Score								
	81.4	82.2	81.7	82.0	84.0	88.1	87.5	**88.3**

Table 10: Blogs data set: the DO and TF score (in %) of all bridging algorithms with the selection algorithm All and the eviction algorithms CC and LS, for different hash table sizes m (in MB).

To understand the performance of BE estimation better we varied the "length" of bridges, i.e. parameter T that limits how many selected shingles a bridge can span. The tables above use $t = 30$. We first report the results for the blogs data set. For Algorithm All with lucky eviction and BE estimation the average DO score (averaged over all values of m) is 87.4% for $T = 10$ and it is continuously increasing with T up to 88.3% for $T = 1000$. The reason is that increasing T leads to a small improvement for all hash table sizes, except for $m = 5GB$ where the DO score is unchanged.

The average TF metric shows the same behavior. For the selection algorithm NHs we see improvements of the same magnitude on the blogs data set when increasing T, but only for $m \leq 500MB$. On the Swiss data set, however, there is barely any change when increasing T for All and only an improvement of up to 0.35 percentage points for NHs. This results in a small improvement in the overall score when increasing the length of bridges. At first this seems surprising since the average size of a copied block is less than 20 tokens (see Table 1.) However, the distribution of the number of tokens in copied blocks has a very long tail and the larger we set T the more of these long copied blocks are bridged.

We also studied for different hash table sizes how many of the shingles are labeled with an origin that is *not* the default origin *due to bridging and expansion*. More precisely, we measured for how many selected shingles did the estimation algorithm NB give the default origin while BE did *not* give the default origin. We call such a shingle an *extra-labeled* shingle. Table 11 shows for the selection algorithm NHs on the blogs data set that the number of extra-labeled shingles increases from 6694 for m=5GB to 133,737 for m=20MB. It also shows that the percentage of extra-labeled shingles with correct origin (i.e. the *accuracy*) increases from 10.5% for m=5GB to 98.6% for m=20MB. Thus as m decreases many more shingles are labeled due to bridging and expansion and more and more of them are correctly labeled. The reason is that for m=5GB most of the copied selected shingles are found in the hash table and thus when bridging is applied it is mostly too aggressive. The situation is completely reversed for m=20MB. Barely any copied selected shingles are found in the hash table and bridging and expansion are a useful technique for restoring the missing information.

m	number of extra-labeled shingles	Accuracy
5000	6694	10.5
2000	8139	30.8
1000	33003	84.8
500	122614	96.6
200	163484	98.3
100	155107	98.6
50	154093	98.8
20	133737	98.6

Table 11: Blogs data set and selection algorithm NHs: The number of extra-labeled shingles and the accuracy (in percent) of the origin of extra-labeled shingles for different hash table sizes (in MB).

3.7 Overall evaluation

The performance of NHs with lucky eviction and BE estimation is very strong: On the blogs, resp. Swiss data set its DO score varies from 98.5%, resp. 97.8% for $m = 5GB$ (34% of the data) to 79.7%, resp. 80.9% for $m = 20MB$ (0.1% of the data). Its TF score decreases from 93.4%, resp. 89.0% (Swiss) for $m = 5GB$ to 78.1%, resp. 76.7% for $m = 20MB$. Both metrics are dropping quite smoothly. That is exactly the kind of behavior one likes to see in real systems. The average DO score is 90.9% on the blogs data set and 91.0% for the Swiss data set. Algorithm All (when combined with lucky eviction and expansion-only estimation) has a lower DO score than NHs on both data sets for all memory sizes higher DO score except for $m = 5GB$, resulting in a lower average DO score of 87.5% on the blogs data set and 90.4%

on the Swiss data set. Thus, intelligent shingle selection leads to a clear gain in the DO metric.

For the TF metric the results are reversed. Algorithm All with lucky eviction and BE estimation has a higher average TF score than Algorithm NHs. It is 88.3% on the blogs data set and 90.8% on the Swiss data set. Algorithm NHs with the same eviction and estimation algorithms achieves an average TF score of 87.2% on the blogs data set and 83.9% on the Swiss data set. Altogether NHs has the higher overall score for the blogs data set and All has the higher overall score for the Swiss data set (see Table 7).
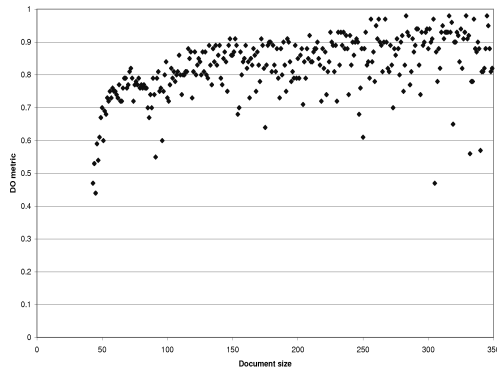

Figure 2: For $m =$20MB the DO metric averaged for all documents of a fixed size for the selection algorithm NHs with lucky eviction and BE estimation.

We also studied how well the best performing algorithm works for different document lengths. Let the *document length* be the number of shingles in a document. We determined the DO metric for $m = 20$MB (where the DO values are the lowest) on the blogs data set and computed for each document length n the mean of this DO score for all documents of length n. Figure 2 shows a plot of these mean DO scores for all document lengths between 38 (i.e., 45 tokens, our shortest document length) to 350, covering 85% of the documents in the blogs data set. It shows a clear increase in DO score as length increases. This is not surprising as for longer documents more shingles are selected and thus more data is available for the determination of the dominant origin.

4. CONCLUSION

We have shown that it is possible to determine the dominant origin of a document and the novelty of its tokens with an accuracy of low 80% to high 90%, depending on the amount of available space. To identify the best algorithm we compared 33 variants of selection algorithms with 4 different eviction algorithms and 4 different estimation algorithms, resulting in $33 \cdot 4 \cdot 4 = 528$ different variations of real-time algorithms. Using space for 34% of the tokens the best algorithm can determine the dominant origin of documents in a collection of 8.6 million blogs with 99% accuracy and can label tokens as fresh or old with 99% precision. Its performance degrades smoothly with the available space. When the space is reduced to 1.4% of the tokens, it determines the dominant origin for every document with 90% accuracy and can label tokens as fresh or old with 86% precision.

Our system consists of three phases, namely a selection phase, an eviction phase, and an estimation phase. The best performing selection algorithm is a new selection algorithm, called Hailstorm. It slightly outperforms Winnowing for all memory sizes and for all combinations of eviction and estimation algorithms. We also present a new eviction policy (lucky eviction) and new estimation techniques (bridging and expansion). When lucky eviction is used in combination with bridging and expansion, the performance of the system is significantly improved. The strong performance and moderate space requirements of our best algorithm suggest that real-time origin detection on web-scale is possible.

5. REFERENCES

[1] R. A. Baeza-Yates, Á. R. P. Jr., and N. Ziviani. Understanding content reuse on the web: Static and dynamic analyses. In *WEBKDD*, pages 227–246, 2006.

[2] R. A. Baeza-Yates, Á. R. P. Jr., and N. Ziviani. Genealogical trees on the web: a search engine user perspective. In *WWW*, pages 367–376, 2008.

[3] M. Bendersky and W. B. Croft. Find text reuse on the web. In *WSDM*, 2009.

[4] Y. Bernstein and J. Zobel. Redundant documents and search effectiveness. In *CIKM*, pages 736–743, 2005.

[5] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. of ACM SIGMOD'95*, pages 398–409, 1995.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proc, of 6th WWW*, pages 1157–1166, 1997.

[7] H.-C. Chang, J.-H. Wang, and C.-Y. Chiu. Finding event-relevant content from the web using a near-duplicate detection approach. In *Web Intelligence*, pages 291–294, 2007.

[8] D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB*, pages 37–45, 2003.

[9] D. Fetterly, M. Manasse, and M. Najork. Detecting phrase-level duplication on the world wide web. In *SIGIR '05*, pages 170–177. ACM, 2005.

[10] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR'06*, pages 284–291, 2006.

[11] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. Am. Soc. Inf. Sci. Technol.*, 54(3):203–215, 2003.

[12] U. Manber. Finding similar files in a large file system. In *USENIX Winter*, pages 1–10, 1994.

[13] M. O. Rabin. Fingerprinting by random polynomials. *Technidal report, Harvard University*, TR-15-81, 1981.

[14] A. A. Saul Schleimer, Daniel S. Wilkerson. Winnowing: local algorithms for document fingerprinting. In *Proc. of. ACM SIGMOD'03*, pages 76–85, 2003.

[15] J. Seo and W. B. Croft. Local text reuse detection. In *SIGIR '08*, pages 571–578. ACM, 2008.

[16] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents and servers on the web. In *WebDB*, pages 204–212, 1998.

[17] J. Zobel and Y. Bernstein. The case of the duplicate documents measurement, search, and science. In *APWeb*, pages 26–39, 2006.