Approximate Nearest-Neighbour Search with Inverted Signature Slice Lists

Timothy Chappell[1], Shlomo Geva[2], and Guido Zuccon[3]

[1] Queensland University of Technology, timothy.chappell@qut.edu.au
[2] Queensland University of Technology, s.geva@qut.edu.au
[3] Queensland University of Technology, g.zuccon@qut.edu.au

Abstract. In this paper we present an original approach for finding approximate nearest neighbours in collections of locality-sensitive hashes. The paper demonstrates that this approach makes high-performance nearest-neighbour searching feasible on Web-scale collections and commodity hardware with minimal degradation in search quality.

Keywords: Locality-sensitive hashing, Hamming distance, Clustering

1 Introduction

To determine the similarity between two documents in term vector space for nearest neighbour search, a cosine similarity calculation or similar measure must be performed for every term they share. To rank all documents in a collection by their distance from a given document, this must be repeated for all documents, rendering this operation infeasible for large collections with large vocabularies.

Locality-sensitive hashing ameliorates this issue by reducing the dimensionality of the term vector space in which these documents are stored and by representing these document vectors as binary strings. This allows expensive vector space similarity calculations to be replaced with cheaper Hamming distance calculations [16] that preserve pairwise relationships between document vectors.

Hashing is capable of reducing the costs of the individual document similarity computations; however, in Web-scale collections of hundreds of millions of documents, reducing the per-document processing time is not sufficient to make nearest-neighbour searching feasible. Efficient methods of computing document similarity are necessary for tasks such as near-duplicate detection (the discovery of pairs of documents that differ only marginally), e.g. for the purposes of removing redundant results while web crawling and plagiarism detection [12,13].

In this paper we consider the problem of performing efficient near-duplicate detection using document signatures, i.e. locality-sensitive hashes used to represent documents for the purpose of searching. Faloutsos and Christodoulakis [4] pioneered the use of superimposed coding signatures with an approach similar to Bloom filters, where signatures would be created directly from the filters and compared for similarity by masking them against other filters and counting the bits that remained. While this approach was shown to be inferior to the inverted

file approach for ad hoc retrieval [18], recent work has since shown improvements on that original approach [7], leading to effectiveness comparable to inverted file approaches.

We introduce a novel approach for efficient near-duplicate detection that involves the generation of posting lists associated with a particular signature collection, making it possible to rapidly identify signatures that are close to a given search signature and discard those that are farther away. Our approach is empirically validated on ClueWeb09[4], a standard, publicly available, information retrieval collection. These experiments show that our approach is capable of performing near-duplicate detection on web-scale collections such as ClueWeb09 (500 million English-language documents) in 50 milliseconds on a commodity desktop PC costing under $10,000.

2 Locality-sensitive hashing

A hash function takes an arbitrary input object and produces a binary string (hash) of a fixed length. A standard property of conventional hash functions is that the same input will always produce the same hash, while a different input is almost certain to produce a vastly different hash. These binary strings can be much smaller than the original inputs, so comparing them for equality can be much faster. This makes them useful for applications such as verifying that a large file was transmitted correctly without needing to retransmit the entire file.

A frequently valued property of hash functions is the *avalanche effect*, where similar (but not identical) inputs produce entirely different hashes [6]. This is valued as it makes malicious attacks that rely on producing a certain hash more difficult. It also means that visual inspection of the hashes of two similar inputs will make it clear that there is a difference. By contrast, the locality-sensitive hash exhibits the reverse of this property: when a locality-sensitive hash function receives two slightly different inputs, the resultant hashes will be either identical or highly similar. This makes locality-sensitive hashing appropriate when it is desirable to match inputs that are similar.

For instance, when creating a collection of documents by crawling the Web it may be desirable to eliminate duplicate pages, as they contain no additional information and will consume extra space [2,12]. Because comparing a newly-downloaded web page to every web page downloaded so far could be very expensive, it may be desirable to hash them to make these comparisons faster. However, in the context of building a web collection, two pages that only differ in title or metadata are still essentially duplicates. With a locality-sensitive hash function, these two almost-identical web pages will have identical or almost-identical hashes, making it possible to detect these when comparing hashes.

This approach can be extended to the more general problem of determining object similarity. The similarity between two locality-sensitive hashes determines how similar two objects are: hashes are used as a proxy for computing similarity

[4] `http://www.lemurproject.org/clueweb09.php/`, last visited January 18, 2015.

using the Hamming distance [8] (the number of bits that differ between the two strings).

3 Related Work

Creating document signatures that can be compared for similarity with a Hamming distance calculation is a well-established use of locality-sensitive hashing.

Broder's Minhash [1] is one example of a locality-sensitive hashing algorithm that has been used successfully in the AltaVista search engine [2] for the purpose of discarding duplicate documents. Simhash [15], a more recent locality-sensitive hashing approach, has also been successfully used in this area. The main limiting factor in the scalability of these approaches is that, although Hamming distance computations can be performed extremely quickly, the execution time required to perform these computations over millions of signatures can quickly add up when dealing with web-scale collections.

Lin and Faloutsos [11] introduced frame-sliced signature files to improve on the performance of signature files without compromising on insertion speed the way Faloutsos' earlier bit-sliced signature files[5] did. In frame-sliced signature files, rather than each term setting bits throughout the signature, the bits set by each term are all set entirely within a randomly chosen frame in the signature. The signatures are then stored vertically frame-wise, requiring only the lists of frames corresponding to the frame positions used by terms in the search query to be processed.

Other attempts have been made to work around the scalability problems inherent to these approaches. Broder [2] found that storing the min hashes of each item in sorted order made searching for near duplicates an $\mathrm{O}(n \log n)$ task as opposed to an $\mathrm{O}(n^2)$ task. Recent work by Sood and Loguinov [17] makes use of the probabilistic nature of Simhash [15] to perform fuzzier searches without needing to scan the entire collection. In the field of image searching, Chum and Matas [3] use an inverted file approach to optimise the generation of Minhash document signatures for large image collections. We distinguish our approach from that used by Chun and Matas by using the inverted files directly to make searching the already-generated signatures more efficient.

4 Corpus filtering approaches

One way to avoid calculating Hamming distances for the entire collection is to remove from consideration signatures that are unlikely to be close to the search signature early on. One example is to use signatures small enough such that two documents that are similar enough to count as duplicates produce the same hash. The documents that correspond to each hash can then be stored in a list associated with that hash, immediately filtering out all the documents that do not have a matching signature.

This approach could be highly efficient, but is limited by the hashing function only supporting one level of discrimination, namely the exact match, which needs

to be tuned to balance the frequency of type I and II errors. This tuning can only be applied per-collection, not per-document, as the search signature must be tuned with the same parameters. The inability to discriminate also prevents it from being used for k-nearest-neighbour searching as the threshold cannot be dynamically tuned for k.

5 Inverted signature slice lists

The approach we propose in this paper, the **inverted signature slice list**, is similar to inverted files [14], but applied to the binary signature, not the original document. The document signature is subdivided into *bit slices*, each of a fixed length. The value of each bit slice and its position are then used to index into an array of lists. The list associated with this slice's value and position provide constant-time lookup of this signature and any others that share the same bit slice. Building these lists from a collection of signatures is very time-efficient because record lengths are fixed and text parsing is unnecessary.

Once the lists have been generated, searching is simply a matter of slicing the search signature and looking up the documents that share slices (both exact matches and close matches). The number of times a given signature appears in these lists and the quality of those occurrences (exact matches being more valuable than near matches) give an indication of how close the document signature is to the search signature. The top-k results can then be extracted from close candidates.

5.1 List generation

The document signatures that comprise a signature collection are fixed-length signatures created as the output of a locality-sensitive hash function applied over all the documents in the original document collection. Document signatures are binary strings of a length that is fixed per collection. Shorter signatures require less storage space and are faster to process. Longer signatures can produce results of a higher quality due to minimising feature crosstalk, as one effect of the dimensionality reduction is that document features are all compressed and intermingled in the signature representation.

Typical signatures used for near-duplicate detection are short (32 or 64 bits long) while those used for image and document similarity comparisons are longer (e.g. Kulis and Grauman use 300-bit signatures [10]).

Signature slicing Generating the posting lists involves reading each signature in the collection, dividing that signature up into slices and adding its id to the lists associated with each slice. This process is very similar to the construction of a typical inverted file, but with two key differences:

- The position of the slice is stored along with the content of the slice to make up the corresponding term. For example, if a slice `00110011` makes up the

first 8 bits of a signature, and the (identical) slice `00110011` makes up the last 8 bits of a signature, the two slices have no relation to one another and hence correspond to entirely different inverted lists.

– While inverted files make use of an associative container for looking up terms, it is simpler and more efficient to use the slice's value directly as an array index. For instance, the slice `00110011` has its value (51 in decimal) used as an index into an array large enough to store all 256 possible slices.

In the proposed approach one of these arrays is created for every possible slice position. If signatures are 64 bits wide, there are 8 possible positions this slice could appear in, hence a total of 8 arrays capable of storing up to 256 slices. This can potentially represent a significant waste of memory if the collection is too small to cover most of the indices; as such, it is important to tune the slice width to suitably match the collection size.

Increasing the slice width reduces the load on any particular [value, position] pair by half; as there are more possible values of each slice, each slice value would cover less of the collection. and hence represents the most effective way of improving search performance.

The most efficient slice width for a particular collection may not necessarily be a power of 2. Furthermore, it may not divide evenly into the signature size. In those cases, when w-bit slices divide unevenly into the n-bit signature, $(w-1)$-bit slices may be included alongside the w-bit slices for some positions to ensure that the slices remain largely uniform in width and that they cover the entire signature. For instance, a 63-bit signature with 32-bit slices may have slice position 0 covered by a 32-bit slice and slice position 1 covered by a 31-bit slice. This means the corresponding table for that slice width may be jagged, with certain columns shorter than others. This has negligible implications for performance; uneven slice widths prove to work just as well in practice as even ones.

Storage considerations The slice lists only need to be generated once for each collection. After generation, the lists can be stored on disk and loaded into memory by the search tool. To minimise loading times, we store the slice lists in a block that can be loaded into memory and used as-is.

The amount of disk space (and, when searching, memory) consumed by the posting lists file is influenced by slice width, the number of slices per signature and the collection size. Low slice widths result in a smaller table structure, but more signatures being referenced in each list. Higher slice widths increase the size of the table, spreading the signature references across more lists.

A reference to every signature in the collection must appear in each column of the table (as every signature will match at least one pattern for every slice position). When the slice width is too small, increasing the slice width can actually reduce the disk space required to store the posting lists. As the slice width continues to increase, however, the amount of space taken up by the supporting structure will also increase, overwhelming the benefits from reducing the number

of entries in the posting lists. As a result, for a given collection size and signature size there is a slice width for optimal memory consumption; increasing or decreasing that slice width will increase the amount of memory needed to store the file.

Slice list generation has little impact on the overall computational time efficiency of our approach. For example, creating the 26-bit slice lists for the English-language subset of ClueWeb09 (approximately 500 million signatures) on a 2.40GHz Intel Xeon computer took under 3 hours single-threaded (using 1024-bit signatures). Generation can be trivially parallelised by having each thread build slice lists for different subsets of the collection and merging them at the end.

5.2 List searching

Searching the slice lists is a more complicated process than indexing them because the search component is responsible for handling slices that do not match the query slices exactly. Initially, the query signature is divided into slices in an identical fashion to the indexed signatures. This may mean uneven slice widths if the desired slice width does not divide evenly into the signature size, in which case it is important that the query signature is sliced in the exact same way.

Neighbourhood expansion The [value, position] pair associated with each slice is looked up in the array of posting lists, as done when indexing. Unlike indexing though, we expand the Hamming neighbourhood of each search and bring in similar signatures, under the assumption that even very similar signatures may not match any of the slices exactly. As an example, the 16-bit signature with two 8-bit slices `10110011 01010001` does not have any slice that exactly matches the search signature `00110011 01010101`, even though there are only 2 different bits and this may well be considered similar enough to match.

To expand the Hamming neighbourhood, after consulting the `[00110011, 0]` list looking for candidate documents to consider, we also consult every other possible slice value within a certain Hamming distance from the original query. For example, to perform a 1-bit Hamming expansion, we would include not only `00110011` but also the 8 other possible slice values that exist one bit away. This includes `10110011` from the example earlier, so this signature would be picked up, as would any other signature that contains a slice within a Hamming distance of 1 from the respective slice in the search signature.

We can continue expanding the Hamming neighbourhood of our search signature by bringing in slices that are farther away. This allows less precise matches to be made at the cost of additional search time. The number of posting lists that must be considered at each expansion is the binomial coefficient of the Hamming distance and the slice width, making the total number of posting lists considered the sum of all Hamming distances up to that point, or $\sum_{i=0}^{h} \binom{i}{w}$ where w is the slice width and h is the Hamming distance to expand the neighbourhood.

To illustrate the interaction between Hamming neighbourhood expansion and slice width, consider two documents with 24-bit signatures, one just different

```
1: for all slice position ∈ query signature do
2:     query value ← query signature[slice position]
3:     for all v ∈ values with 0-n bits set do
4:         distance ← popcount(query value ⊕ v)
5:         similarity ← slice width − distance
6:         signature ← list[query value ⊕ v, slice position]
7:         score[signature] ← score[signature] + similarity
8:     end for
9: end for
```

Fig. 1: Pseudo-code algorithm for list searching.

enough from the other to have 2 bits that differ (their Hamming distance is 2). This signature could be sliced up in a number of ways; e.g., into 8 or 12-bit slices. If 12-bit slices are used, there is a $\frac{12}{23}$ probability that both differing bits will end up in different slices and an $\frac{11}{23}$ probability that they end up in the same slice. In the latter case, there is no need to expand the neighbourhood as one of the slices will match exactly. In the former case, a 1-bit expansion is necessary.

With 8-bit slices, there will always be at least one slice that is identical between the two signatures. As such, while neighbourhood expansion is unnecessary for the identification of all signatures a Hamming distance of 2 away when using 8-bit slices, 12-bit slices can only be expected to identify $\frac{11}{23}$ of them without expanding the neighbourhood.

It should be noted that 12-bit slices will have posting lists $\frac{1}{16}$ of the length of 8-bit slices, meaning that moving to a 1-bit neighbourhood expansion (and hence needing to process $13\times$ the number of posting lists) would still improve performance over using 8-bit slices and no neighbourhood expansion.

In summary, while increasing the slice width does trade search accuracy for an increase in retrieval speed, the trade-off is sufficiently worthwhile that even expanding the Hamming neighbourhood to fully counteract the reduced search accuracy is often a more attractive option than leaving the slice width the same. However, given that the improvement in retrieval speed plateaus after the search table reaches a collection-dependent level of sparsity, retrieval time efficiency can only be increased up to a point while maintaining a given level of search accuracy.

Hamming distance estimation Processing these lists up to the desired neighbourhood expansion allows the search tool to not only obtain a subset of the collection containing most of the close signatures, but also to use this same information for calculating optimistic and pessimistic Hamming distances. This can make it possible to cull the subset further before calculating true Hamming distances. Algorithm 1 shows the approach we use, with approximate Hamming distance similarity referred to as *score*. After processing the posting lists, the highest-scoring signatures are likely to be the signatures with the lowest Hamming distances from the query signature.

To illustrate this, consider the case of 32-bit signatures and four 8-bit slices before neighbourhood expansion. After consulting the posting lists for all slices,

the potential range of each signature's Hamming distance can be calculated. A signature that appears in all 4 slices is one that has exactly matched the search signature and as a result has a Hamming distance of 0. A signature that appears in none of the slices cannot have a Hamming distance of less than 4 as it would appear in at least one slice otherwise. Therefore, its optimistic Hamming distance can be calculated as 4 (a case in which every slice had 1 bit differing from the search signature) and its pessimistic distance calculated at 32 (a case in which no slice had any bits in common with the search signature.) In the same way, a signature that appears in 3 of the slices has an optimistic Hamming distance of 1 (if the slice the signature did not appear in had 1 bit that differed from the respective slice in the search signature) and a pessimistic Hamming distance of 8 (that same slice containing all differing bits.)

The range between optimistic and pessimistic Hamming distances can be narrowed through neighbourhood expansion. In the previous example, one signature did not appear in any of the slices and hence could have had a Hamming distance of anything from 4 to 32. On expanding the neighbourhood by 1 bit, if the signature still never appears in any of the slices, the possible range of values its Hamming distance could occupy is reduced to 8-32.

Expanding the Hamming neighbourhood increases the quality of these estimations at the expense of more search time, but also reducing the subset of signatures that fall within the desired range, allowing these signatures to be skipped when calculating true distances later. Based on user requirements, the signature size, slice width, neighbourhood expansion and heuristics for discarding signatures based on their optimistic and/or pessimistic Hamming distances can be tuned to produce the desired trade-offs between performance, memory usage and quality of results.

6 Evaluation

Search accuracy and retrieval time are the most important factors when judging the efficacy of any search approach. Tuning parameters for the inverted signature slice list approach involves making speed-accuracy trade-offs. To judge whether certain trade-offs are worthwhile or not, it is necessary to be able to judge the correctness of the results returned.

Experiments are conducted on a subset of 500 million English-language documents from the ClueWeb09 Category A. We have used 1024-bit TOPSIG [7] signatures; while signature width has an impact on search quality this impact has been explored elsewhere [7] and is not the topic of our research, which is more concerned between the comparative quality between ISSL searches and searches of the raw signatures. As the inverted signature slice list approach is designed to retrieve the signatures with the closest Hamming distances to the query, we are using an exhaustive Hamming distance search that retrieves the closest results without fail as an approach to compare against. By definition, the closer the results retrieved by this approach are to the exhaustive results, the more correct they are.

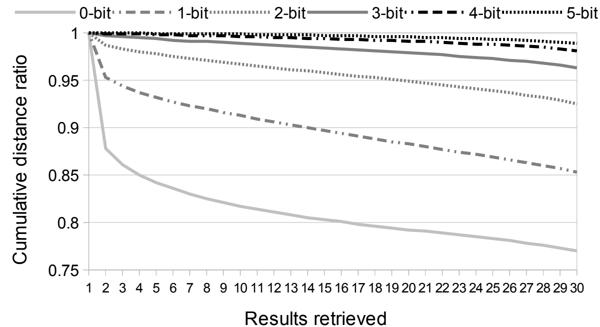

Fig. 2: The impact of neighbourhood expansion (0-bit meaning no expansion) on cumulative distance ratio.

Making a search quality judgement therefore requires a quantitative way of analysing one set of search results in terms of how closely it matches a second set of search results. We introduce the *cumulative distance ratio* metric, which is akin to a graded relevance metric designed for evaluating lists of Hamming distances. This metric considers two lists of equal length; one a list of the signatures returned by some retrieval method, the other being the definitive list of closest signatures (obtained using an exhaustive Hamming distance search with every signature in the collection). It ought to be remembered that, as the inverted signature slice list approach is only concerned with returning the top-k nearest neighbours, here we are measuring its accuracy compared to the definitive list of top-k nearest neighbours.

The distance ratio at position p is calculated as the ratio between the cumulative sums of the Hamming distances of the retrieved documents up to position p: $DR(p) = \frac{\sum_{i=1}^{p} T(i)}{\sum_{i=1}^{p} D(i)}$, where $D(i)$ is the Hamming distance of the ith result from the algorithm being evaluated and $T(i)$ is the Hamming distance of the ith closest signature. For the purposes of calculating the distance ratio, we let $0 \div 0 = 1$: this can be a common occurrence as it happens every time there is an exact duplicate in the collection (Hamming distance of 0) and the search algorithm finds it. From this, we can calculate the cumulative distance ratio $CDR(p) = \sum_{i=1}^{p} DR(i)/p$.

6.1 Hamming neighbourhood expansion

Expanding the Hamming neighbourhood, as described earlier, causes more posting lists to be consulted for each search. This increases the pool of candidates and hence search quality at the cost of increased retrieval time. As Figure 2 shows, only a few bits of neighbourhood expansion are needed to greatly improve search quality and expanding beyond that not only provides increasingly diminishing returns but also comes with a substantial impact to performance (search time:

i	j	Search time	CDR@10
0	0	0.040ms	0.817
1	0	0.112ms	0.869
	1	0.193ms	0.913
2	0	0.568ms	0.896
	1	0.703ms	0.951
	2	1.399ms	0.967

i	j	Search time	CDR@10
3	0	2.242ms	0.911
	1	2.452ms	0.967
	2	3.251ms	0.985
	3	5.080ms	0.989
4	0	7.011ms	0.913
	1	7.258ms	0.971
	2	8.517ms	0.99
	3	11.483ms	0.995
	4	12.744ms	0.996

Table 1: Searching a 1 million document subset of Wikipedia (1024-bit signatures, 16-bit slices, 20 threads, $k = 30$) with the smaller candidate threshold. (i = distance beyond which to stop considering posting lists. j = distance beyond which to stop extending the list of candidate signatures)

3-bit = 5.084ms, 4-bit = 12.534, 5-bit = 27.865, 8-bit = 130.887ms). This is due to the number of posting lists increasing binomially while the number of close signatures remaining in the collection is soon depleted, causing the cumulative distance ratio to quickly plateau.

6.2 Slicing optimisations

One optimisation we have implemented to gain some of the benefits from an expanded Hamming neighbourhood (specifically, the more precise Hamming ranges of the signatures found early on) is to define an earlier Hamming range, beyond which any signatures only seen for the first time will not be considered.

In other words, when processing posting lists beyond this Hamming distance, any documents that are seen and have already accrued score from earlier posting lists will have their score increased as normal. However, signatures that have not yet been seen and do not yet have a score will be ignored. This allows expensive write operations for signatures with a low likelihood of being close enough to the search query to be elided, saving that processing time as well as the processing time required to analyse the score table at the end and extract the top results.

Table 1 demonstrates that this can provide strong improvements in efficiency, but with a corresponding drop in search accuracy that may not be worthwhile under other circumstances.

While the most effective slice width for a given collection size will depend on a number of factors, including available memory, a good rule of thumb is to increase the slice width by one bit each time the collection doubles in size. Doubling the size of the collection will result in the average posting list length doubling in size too, which will make lookups far slower. Increasing the slice width, on the other hand, will cause the average posting list length to halve, the two effectively cancelling each other out.

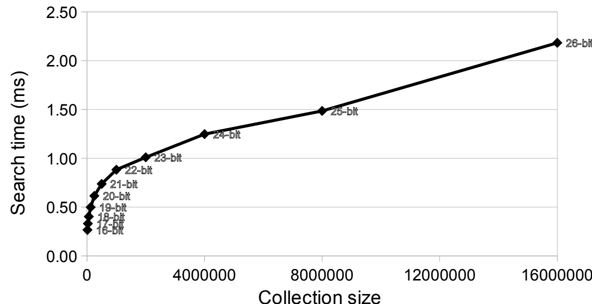

Fig. 3: Keeping the slice width in line with collection growth to reduce the corresponding growth in search times

Slice width	Search time	Memory (MB)	CDR@10
23-bit	199.738ms	180029.43	0.925
24-bit	112.783ms	177417.01	0.915
25-bit	66.753ms	176260.59	0.902
26-bit	56.955ms	177346.38	0.894
Exhaustive	2843.619ms	92266.03	1.000

Table 2: Searching ClueWeb09 (500 million documents). 3-bit Hamming expansion, 20 threads.

Figure 3 captures the most significant aspect of the inverted slice signature lists approach. Note that each point on the curve corresponds to a different slice width, and a successive doubling of the collection size. As the collection size is increased 1024-fold along the x-axis, the search time is only increased by less than 10-fold. This is what makes it possible to search the English ClueWeb09 for top-k nearest in about 57ms. By comparison, an exhaustive signature search takes about 2.8s (see Table 2); we achieve approximately a 50-fold speedup with the inverted signature slice list approach.

7 Conclusion

We have presented an approach to improving the speed of nearest-neighbour signature searching without a considerable loss to search fidelity. While it is difficult to make direct comparisons to other systems, most of which have been designed for different purposes and for which publicly available code and/or data are not provided, none of the systems we have surveyed [3,9,12] work on web-scale collections with (high-end) consumer-level hardware. The field of prior research in this area seems largely divided into two camps: groups using consumer-level hardware searching non-web-scale collections (hundreds or thousands of documents or low millions) [3,9]; and groups searching web-scale collections with highly efficient

networks of Hadoop clusters [12]. The former are working in an entirely different problem space while the latter are difficult to benchmark against, particularly if the code and computational platforms are not available.

We consider here that 50-millisecond search of a 500 million document collection on consumer-level hardware is a compelling justification for the modest loss of precision. The effective use of inverted signature slice lists may be limited to certain applications (near-duplicate detection, clustering etc.), in those situations they can provide great performance improvements over exhaustive approaches. The implementation described in this paper is available under an open-source license and distributed at `http://www.topsig.org`.

References

1. A Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997*, pages 21–29. IEEE, 1997.
2. A Broder. Identifying and filtering near-duplicate documents. In *Combinatorial pattern matching*, pages 1–10. Springer, 2000.
3. O Chum and J Matas. Fast computation of min-hash signatures for image collections. In *CVPR'12*, pages 3077–3084, 2012.
4. C Faloutsos and S Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *TOIS*, 2(4):267–288, 1984.
5. Christos Faloutsos and Raphael Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *VLDB*, volume 88, pages 280–293, 1988.
6. H Feistel. Cryptography and computer privacy. *Sci. Am.*, 228:15–23, 1973.
7. S Geva and C De Vries. Topsig: topology preserving document signatures. In *CIKM'11*, pages 333–338, 2011.
8. R Hamming. Error detecting and error correcting codes. *Bell System Tech J*, 29(2):147–160, 1950.
9. Q Jiang and M Sun. Semi-supervised simhash for efficient document similarity search. In *ACL'11*, pages 93–101, 2011.
10. B Kulis and K Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV'09*, pages 2130–2137, 2009.
11. Zheng Lin and Christos Faloutsos. Frame-sliced signature files. *Knowledge and Data Engineering, IEEE Transactions on*, 4(3):281–289, 1992.
12. G Manku, A Jain, and A Das Sarma. Detecting near-duplicates for web crawling. In *WWW'07*, pages 141–150, 2007.
13. M Potthast and B Stein. New issues in near-duplicate detection. In *Data Analysis, Machine Learning and Applications*, pages 601–609. Springer, 2008.
14. C. J. Van Rijsbergen. *Information Retrieval*. Butterworth, 1979.
15. C Sadowski and G Levin. Simhash: Hash-based similarity detection. Technical report, Google Tech Rep., 2007.
16. M Slaney and M Casey. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine*, 25(2):128–131, 2008.
17. S Sood and D Loguinov. Probabilistic near-duplicate detection using simhash. In *CIKM'11*, pages 1117–1126, 2011.
18. J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *TODS*, 23(4):453–490, 1998.