Finding near-replicas of documents on the web

Narayanan Shivakumar, Hector Garcia-Molina
Department of Computer Science
Stanford, CA 94305.
{*shiva, hector*}*@cs.stanford.edu*

Abstract. We consider how to efficiently compute the overlap between all pairs of web documents. This information can be used to improve web crawlers, web archivers and in the presentation of search results, among others. We report statistics on how common replication is on the web, and on the cost of computing the above information for a relatively large subset of the web – about 24 million web pages which corresponds to about 150 Gigabytes of textual information.

1 Introduction

Many documents are being replicated across the world wide web. For instance, there are several copies of JAVA FAQs and Linux manuals on the net. Many of these copies are exactly the same, while in some cases the documents are "near" copies. For instance, documents may be older versions of some popular document, or may be in different formats (e.g., HTML, or Postscript), or may have additional buttons, links and inlined images that make them slightly different from other versions of the document.

Replication also occurs at a higher level than documents. In some cases, entire servers are mirrored across different countries and updated daily, weekly or on a monthly basis. The Linux Documentation Project (LDP) page with Linux manuals is one such instance, with over 180 mirror servers across the world. In some cases servers are exclusively dedicated to maintaining LDP pages and are therefore exact or near-replicas; in other cases there are several servers that replicate LDP manuals along with other manuals (`http://sunsite.unc.edu` contains manuals on LDP, FreeDOS, JAVA, etc.) In this paper, we concentrate on computing pair-wise document overlap; we can similarly solve the pair-wise server overlap problem if we consider the server to be the union of all documents in the site.

We believe it is useful to identify near-replicas of documents and servers for the following applications:

1. **Improved web crawling:** Until recently, web crawlers [2, 10] crawled the "entire" web. However since the web is growing rapidly and changing even faster, current crawlers are concentrating on crawling some subset of the web that has "high importance." For example, crawlers may only choose to visit "hot pages" with a high *backlink* count, i.e., pages that are pointed to by many other pages [6, 11]. After finishing each crawl, these crawlers will recrawl the same pages to acquire updates to the hot pages since the last crawl.
   When a limited number of web pages are to be crawled, a crawler should avoid visiting near-replicas, especially if they can be efficiently identified based on prior crawls. For instance, by identifying the 25 MBs of LDP pages to be near-copies ahead of time, the crawl can avoid crawling and indexing these pages from the 180+ sites – a potential savings of nearly 4.5 GBs for LDP pages alone!
2. **Improved ranking functions in search engines:** If documents A, B and C are considered relevant (in that order) to a user query, search engines currently[1] rank links to the three documents based on relevance, without regard to the overlap between the documents. However, we believe the "interestingness" of B is lower than C, if B overlaps with A significantly, while C is a distinct result. Similar proposals are being considered for "result reordering" based on overlap in other database contexts as well [7, 8].
3. **Archiving applications:** Since companies are trying to "archive" the web [1], it may be useful for them to identify near-copies of documents and compress these in a more efficient manner. Also finding near-copies is useful in services that avoid the common "404 error" (error for "Document not found" on the web) by presenting an alternate near-copy on the web, or a copy from a web archive [1].

1.1 Related work

Manber considered computing pair-wise document overlap in the context of finding similar files in a

[1] Search engines are beginning to cluster exact copies of documents by computing simple checksums.

large file system [13, 14]. Researchers at the DEC SRC Lab are developing a similar tool to "syntactically" cluster the web [4, 5]. Heintze has developed the KOALA system for plagiarism detection [9]. As part of the Stanford Digital Library project, we have developed the COPS [3] and SCAM [15, 16] experimental prototypes for finding intellectual property violations. All the above techniques use variations of the same basic idea – compute "fingerprints" for a set of documents and store into a database. Two documents are defined to have significant overlap, if they share at least a certain number of fingerprints. The above systems have different target applications, and therefore differ in how they compute fingerprints of documents, and how they define the similarity measure between a pair of documents.

In this paper, we do not propose new notions of document similarity: we use the similarity measures we have been using in our SCAM (Stanford Copy Analysis Mechanism) prototype over the past three years [15, 16]. We primarily concentrate on efficiently solving the "clustering" problem of computing overlap between all document-pairs simultaneously. This problem is especially hard when the number of documents is on the order of millions. The techniques we use to solve this problem are different from the techniques adopted by the above work [5, 9, 13, 14] as well as in our prior implementation in SCAM. Our new approach to solving the all-pairs document overlap problem is based on efficiently executing *iceberg* queries [17]. As we will see, our techniques take orders of magnitude less space and time.

2 Computing all-pairs document overlap

Each document is first converted from its native format (e,g., HTML, PostScript) into simple textual information using standard converters on UNIX (such as `ps2ascii`, `html2ascii`). The resulting document is then *chunked* into smaller units, such as words, sequences of words, sentences, paragraphs, or the entire document itself. Each textual chunk is then hashed down to a 32-bit fingerprint and stored into the *FPrint-DocID* file, with the following attributes: (1) `fprint`, which is the hashed fingerprint, and (2) `docID`, the ID of the document (typically a 32-bit integer).

We define the problem of computing all-pairs document overlap as follows. Produce the *Overlap* lookup table with triplets of the form $\langle d_i, d_j, F\rangle$, if documents d_i and d_j share F fingerprints in common. Since we are interested in cases where there is "significant" overlap between documents, we can restrict ourselves to storing tuples $\langle d_i, d_j, F\rangle$, only when $F \geq T$, for some threshold T.

2.1 Sort-based approach

The simplest approach to computing all-pairs overlap is the "sort-based" approach as follows. (1) Sort the *FPrint-DocID* file on the `fprint` attribute. (2) Scan the sorted *FPrint-DocID* file: if f_k is a `fprint`, d_i, d_j are `docID`s, and if $\langle f_k, d_i\rangle$ and $\langle f_k, d_j\rangle$ occur in the sorted *FPrint-DocID* file, produce pair $\langle d_i, d_j\rangle$ into file *DocID-DocID*. (3) Sort the *DocID-DocID* file by both attributes, so that all $\langle d_i, d_j\rangle$ pairs, for documents d_i and d_j, are contiguous. (4) Scan the sorted *DocID-DocID* file: if some pair $\langle d_i, d_j\rangle$ occurs more than T times, the document pair is deemed to have significant overlap, and is stored in the *Overlap* lookup table. The above is roughly the approach followed by earlier work in computing all-pairs overlap, including the GLIMPSE system [13, 14], the DEC prototype [5] and in our earlier implementations of our COPS and SCAM prototypes.

The efficiency of the above implementation depends critically on how often fingerprints occur across documents. This is because Step (2) produces a cross-product of all document pairs that share a fingerprint, and subsequent steps process the resulting cross-product. Hence if fingerprints tend to be common across documents, the output may be too large (as we will report in our experiments). In fact, a significant portion of the output from Step (2) consists of document pairs that share one or two (common) fingerprints in common. These pairs will be discarded only in Step (5) when it is recognized that the document pairs do not share more than T fingerprints – until then, these tuples will be stored and sorted, requiring large amounts of storage as well as processing time.

In general, fingerprints that are computed on sequences of words or lines tend to be common, compared to fingerprints computed on paragraph level chunks. On the other hand, similarity measures that use the latter fingerprints tend to be less effective in finding overlap [3, 15, 16]. In applications where the goal is to identify "approximately similar" documents (as in GLIMPSE [13, 14], and in DEC's prototype [5]), the former class of fingerprints tend to be useful due to their efficiency. On the other hand, in applications (like SCAM and COPS [3, 15]) where the goal is to identify document pairs with smaller overlaps, we choose to use the latter class of fingerprints. Hence we need more efficient procedures to compute the *Overlap* table.

2.2 Probablistic-counting based approach

Roughly, our approach to computing the *Overlap* table for a given chunking (such as lines) is as follows. We first compute document pairs that are exact copies using the above "sort-based" procedure, and remove them from subsequent processing. This step is useful in cases when the number of exact copies is very large, since it reduces the work required by the subsequent, less efficient steps. We then chunk up the document and compute fingerprints for the document, and solve the pair-wise overlap problem using "probabilistic counting."

We can implement the above conceptual steps in a variety of ways, depending on our main memory resources. For the rest of the discussion, we assume the following about our main memory: (1) We can maintain the set of `docIDs` in main memory, along with a small constant number of bytes (about ten) for each `docID`. This assumption was almost valid in our case and some of our data structures were in fact allocated in virtual memory. However the paging overhead incurred was very small, and we continue to make this assumption in the rest of our discussion. (2) We do not have the main memory to maintain a counter for each $\langle d_i, d_j \rangle$, document pair that share a chunk. (If we did, we can easily count the number of chunks each document pair shares.) We compute the *Overlap* table as follows under the above assumptions of our main memory.

1. *Produce fingerprints:* We compute one fingerprint for each document based on the entire document, and store into *FPrint-DocID-Exact*. We also compute fingerprints based on the chosen chunk, and store into *FPrint-DocID-Chunk*. Sort the *FPrint-DocID-Exact* file, so that `docIDs` of documents with the same fingerprint are located contiguously.
2. *Remove exact copies:* Scan the sorted *FPrint-DocID-Exact* file: for each d_i that contains fingerprint f_k, maintain in main memory the smallest `docID` d_j that also shares f_k. (The document with smallest `docID` is chosen as the "central" document for a set of documents with the same signature.) Finally remove the signatures of non-center documents from *FPrint-DocID-Chunk*, by scanning *FPrint-DocID-Chunk* and producing *FPrint-DocID-Chunk'*.
3. *Computing non-exact overlap:* We use "coarse counting" or "probabilistic counting" as our primary basis for computing the *Overlap* table. Coarse counting is a technique often used for query size estimation, for computing the number of distinct targets in a relation, for mining association rules and for other applications. The simplest form of coarse counting uses an array $A[1..m]$ of m counters and a hash function h_1. The *CoarseCount* algorithm works as follows: Initialize all m entries of A to zero. Then perform a linear scan of the sorted *FPrint-DocID-Chunk'* file. For each d_i and d_j that share some fingerprint f_k increment the counter $A[h_1(d_i, d_j)]$ by one. After completing this *hashing-scan*, compute a bitmap array $BITMAP_1[1..m]$ by scanning through array A, and setting $BITMAP_1[i]$ if bucket i is *heavy*, i.e. if $A[i] \geq T$. We compute $BITMAP_1$ since it is much smaller than A, and maintains all the information required in the next phase. After $BITMAP_1$ is computed, we reclaim the memory allocated to A. We then compute the *Overlap* table by performing a *candidate-selection* scan of *FPrint-DocID-Chunk'*: for each pair d_i, d_j sharing f_k and with $BITMAP_1[h_1(d_i, d_j)]$ set to one, we add the pair $\langle d_i, d_j \rangle$ to a candidate list F. Finally we remove the false-positives by performing one more scan of the sorted *FPrint-DocID-Chunk'*, and explictly counting the number of fingerprints the document pairs in F share, thereby producing *Overlap*. The candidate-selection scan in this simple coarse-counting algorithm may compute a large F, due to false-positives. We however can progressively remove false-postives using *sampling* techniques, as well as *multiple hash-functions* [17]. In practice, we have seen that by using a small number of hash functions and a small pilot sample, we can remove a large fraction of false-positives for many real-world data sets [17].
The above procedure allows us to avoid explicitly storing the cross-product produced in Step (2) of the "sort-based" algorithm, and the sorting required in the subsequent steps. In fact in many cases, the above approach would finish computing the *Overlap* table, even before Step (2) in the "sort-based" approach terminates [17].

3 Experiments

We used 150 GBs of web data crawled by the Stanford BackRub web crawler [12] for our experiments. This corresponds to approximately 24 million web pages crawled primarily from domains located in the United States of America. We ran our experiments on a SUN UltraSPARC with dual processors,

	Measures / Signatures	Entire document	Four lines	Two lines
Space	Fingerprints	800 MBs	2.4 GBs	4.6 GBs
	Server Map	750 MBs	750 MBs	750 MBs
	URL Map	1.5 GBs	1.5 GBs	1.5 GBs
Time	Read "pagefeed" and compute fingerprints	44 hrs	44 hrs	44 hrs
	Sort *FPrint-DocID*	52 mins	163 mins	7.14 hrs
	Compute all-pairs overlap	45 mins	85 mins	2.42 hrs

Table 1. Storage and time costs for computing *Overlap*.

256 MBs of RAM and 1.4 GBs of swap space, running SunOS 5.5.1. We computed fingerprints of each document for three different chunkings: (1) one fingerprint for the entire document, (2) one fingerprint for every four lines of text, and (3) one fingerprint for every two lines of text. We computed the corresponding *FPrint-DocID* file with these fingerprints, along with a 32-bit integer for the URL and Server. We also maintained a "URL Map" to keep track of the textual URL along with the corresponding numerical identifier. Similarly we maintained a "Server Map." We report the costs in storing the signatures and maps, as well as the time breakdown in computing the *Overlap* table in Table 1, with $T = 15$ and $T = 25$ for the "four line" and "two line" fingerprints respectively. Note that the time to compute *Overlap* for the "four line" and "two line" fingerprints, is the additional time required to compute *Overlap*, after computing the document-pairs that are exact replicas.

In Figure 1 we report the number of replicas for each document for the three chunkings. For instance, about 64% of the 24 million web pages have one exact (darkest bar) replica (itself); about 18% of pages have an additional replica – that is, there are about $\frac{1}{2} * \frac{18}{100} * 24 * 10^6$ distinct pages that have one replica among the other $\frac{1}{2} * \frac{18}{100} * 24 * 10^6$ pages. Similarly, about 5% of pages have between 10 and 100 replicas. As expected, the percentage of pages with more than one replica increases when we relax the notion of similarity from 36% (100 − 64%) for exact replicas, to about 48% (100 − 52%) for "two-line" chunks.

We were initially very surprised by the high degree of replication implied by Figure 1. To understand what data was replicated, we manually examined several highly replicated pages, and identified the following common reasons for such high replication:

1. **Server aliasing:** Several servers have multiple aliases, and therefore different URLs lead to the same page. For instance, `www.webcrawler.com` has many aliases including `wc4.webcrawler.com` and `webcrawler.com`. Hence the BackRub crawler that currently retrieves pages based on URLs, does not in fact recognize the server aliases. Note these are not really copies, but the crawler retrieves these URLs multiple times since the textual URLs appear different. We believe checking the IP address of the machine before crawling may help in a few cases. However in many cases, machines have multiple IP addresses. So in general, detecting server aliasing is a hard problem.
2. **URL aliasing:** Many documents on a server in fact have multiple URLs due to links on the local file system. For instance, the URLs `http://www-db.stanford.edu/~ widom` and `http://www-db.stanford.edu/people/widom.html` in fact correspond to the same HTML document, with a UNIX link on the local filesystem on the www-db machine from the latter to the former.
3. **Replication of popular documents:** As we mentioned in the introduction, there are several manuals and FAQs that are replicated in many servers across the world. Since many manuals contain several hundreds or thousands of HTML pages, when they are replicated across 10 – 100 sites, they constitute a significant fraction of web pages. In Table 2 we report the five most popular set of pages that were replicated in our data, along with an approximate count of the number of distinct servers they were replicated at.

The knowledge gained from our copy analysis can be used to significantly reduce the amount of work done by a crawler. For example, if a page has 10 copies, on its next crawl the system need only visit and index one of those pages. Using our sample data and assuming we only avoid visiting exact copies, the number of pages to be visited can be reduced by 22%. This is about 33 GB (out of our 150 GB total) of

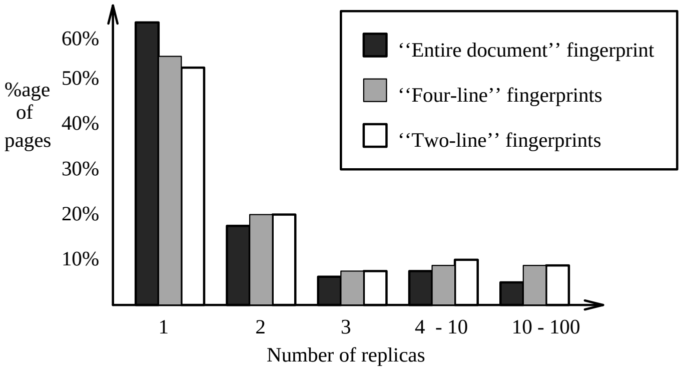

Fig. 1. Document replication on 24 million web pages.

data that does not have to be retrieved, indexed, and stored! If we avoid visiting documents that share 15 or more 4-line chunks, the visited pages can be cut by 29%, and if we avoid documents that overlap by 25 or more 2-line chunks the savings are 33%. In the later case, this is almost 50 GBs less! One can see that the extra effort of finding the near-duplicates (as opposed to exact duplicates) pays off, since we can avoid visiting and indexing 50 - 33 or 17 GB less of data.

4 Conclusion and Future Work

Web documents are being replicated across servers. In many cases, these documents are *near-copies* of other documents. We considered the problem of efficiently identifying all document pairs on the web that have significant overlap. Such overlap information is useful in a variety of applications including web crawling, improving presentation of search results, as well as in web archiving. We report on the degree of replication among 24 million web documents crawled by the Stanford BackRub crawler.

When we started this research, we were interested in knowing how much of the web was replicated. We later realized that our view of the web was very biased by the set of pages crawled by our specific web crawler. Indeed this is the case for any other commercial or research web crawler. We now believe that the techniques presented in this paper serve as a good "debugger" to evaluate web crawlers. For instance the BackRub crawler currently uses one among many possible heuristics to choose the set of pages to crawl. The statistics we compute however show that the current heuristic crawls pages with multiple replicas, several times. Based on the statistics we computed, the authors of the crawler are currently considering options to improve their current heuristic. One proposal they are considering to avoid server aliasing problems, is to use IP addresses of machines, as well as IP addresses of domains (since a machine might have multiple IP addresses) while choosing which pages to crawl. Indeed the degree of replication after the next crawl based on some improved heuristic may be substantially different!

We would also like to raise a more philosophical question, based on our statistics: does the web really need so many replicas of popular pages? For instance, even if the TUCOWS site is popular, we wonder if the number of accesses across the sites, justifies such a high degree of replication. Also it is especially disturbing that the highly replicated Microsoft IIS manuals, and JAVA 1.0.2 API documentation are in fact ancient, and subsumed by future versions of the manuals. Perhaps the HTTP protocol should start including additional meta-information about documents, so that each page can be optionally tagged with information on what is the "primary" document from which the page was copied. Such information will certainly be useful to crawlers, archivers as well as for notifying users that "more up-to-date" copies can be found at an alternate site.

References

1. Alexa. Alexa technology. `http://www.alexa.com`.
2. AltaVista. Altavista search engine. `http://altavista.digital.com`.
3. S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, May 1995.

Rank	Description	Approximate number of server replicas
1	TUCOWS WinSock utilities http://www.tucows.com	100
2	Microsoft Internet Information Server 2.0 Manuals http://acctserver.cob.vt.edu/iisadmin/htmldocs/	90
3	UNIX Help Pages http://www.ucs.ed.ac.uk/~ unixhelp/	75
4	RedHat Linux Manual Version 4.2 http://www.redhat.com/support/docs/rhl	55
5	JAVA 1.0.2 API Documentation http://dnaugler.cs.semo.edu/tutorials/API/	50

Table 2. Popular web pages.

4. A. Broder. On the resemblance and containment of documents. Technical report, DIGITAL Systems Research Center Tech. Report, 1997.
5. A. Broder, S.C. Glassman, and M. S. Manasse. Syntactic Clustering of the Web. In *Sixth International World Wide Web Conference*, April 1997.
6. J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through url ordering. Technical report, Stanford DBGroup Tech. Report, November 1997.
7. D. Florescu, D. Koller, and A. Levy. Using probabilistic information in data integration. In *Proceedings of 23rd Conference on Very Large Databases (VLDB'97)*, pages 216 – 225, August 1997.
8. L. Gravano and H. Garcia-Molina. Merging ranks from heterogeneous internet sources. In *Proceedings of 23rd Conference on Very Large Databases (VLDB'97)*, pages 196–205, August 1997.
9. N. Heintze. Scalable document fingerprinting. In *Proceedings of Second USENIX Workshop on Electronic Commerce*, November 1996.
10. Infoseek. Infoseek search engine. `http://www.infoseek.com`.
11. J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of 9th ACM-SIAM Symposium on Discrete Algorithms, (SODA'98)*, 1998.
12. S. Brin L. Page. Google search engine/ backrub web crawler. `http://google.stanford.edu`.
13. U. Manber. Finding similar files in a large file system. Technical Report TR 93-33, University of Arizona, Tuscon, Arizona, October 1993.
14. U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the winter USENIX Conference*, January 1994.
15. N. Shivakumar and H. Garcia-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, June 1995.
16. N. Shivakumar and H. Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.
17. N. Shivakumar and H. Garcia-Molina. Computing iceberg queries effciently. Technical report, Stanford DBGroup Technical Report, October 1997.