September 10, 2001

SRC Research Report 174

Towards Web-scale Web Archaeology

Shun-Tak A. Leung
Sharon E. Perl
Raymie Stata
Janet L. Wiener

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.compaq.com/SRC/

Towards Web-scale Web Archaeology

Shun-Tak A. Leung
Sharon E. Perl
Raymie Stata
Janet L. Wiener

September 10, 2001

Abstract

Web-scale Web research is difficult. Information on the Web is vast in quantity, unorganized and uncatalogued, and available only over a network with varying reliability. Thus, Web data is difficult to collect, to store, and to manipulate efficiently.

Despite these difficulties, we believe performing Web research at Web-scale is important. We have built a suite of tools that allow us to experiment on collections that are an order of magnitude or more larger than are typically cited in the literature. Two key components of our current tool suite are a fast, extensible Web crawler and a highly tuned, in-memory database of connectivity information. A Web page repository that supports easy access to and storage for billions of documents would allow us to study larger data sets and to study how the Web evolves over time.

1 Introduction

Web archaeology is the study of a particular artifact: the *content* of the World Wide Web. The content of the Web is the product of millions of people working alone and in many disparate organizations; this uncoordinated nature makes the Web a fascinating artifact to study. In addition, the Web is growing as a medium for commercial enterprise, so understanding the Web is crucial to many businesses. At the same time, the Web is the largest information system built to date; its scale makes the study of the Web a significant engineering challenge. To paraphrase the Hitchhiker's Guide to the Galaxy (with apologies to Douglas Adams),

> *The Web is big. Really big. You just won't believe how vastly hugely mindbogglingly big it is. I mean, you may think it's a long way between Yahoo and your Web site, but that's just peanuts to the Web.*

Over the past four years, Compaq Research has had a significant program in the area of Web archaeology. Compaq Research built the AltaVista [21] and Speech-Bot [23] search engines and has published numerous papers about tools and applications [2, 3, 4, 5, 6, 8, 9, 11, 12, 13, 22]. This paper describes what we have learned about doing Web archaeology based on the authors' own work and that of our colleagues at Compaq Research. It also describes the tools that we use, how they fit together, and the problems we have encountered in their deployment.

The next section describes our approach to Web archaeology and some of the tools that support it. Section 3 describes the problems we have encountered when trying to do Web research on a large scale. Section 4 presents an important new tool that we believe enhances the tool suite. Section 5 concludes.

2 Our Approach to Web Archaeology

Over the years, our approach to Web archaeology has acquired a distinct style that we believe, in hindsight, is defined by the following characteristics. First, we emphasize doing Web-scale research, *i.e.*, experimentation on very large data sets. Second, we focus on doing research through *features*, which are carefully defined values computed from the contents of pages.

2.1 Web scale research

Very often, Web studies download a few million pages for study, frequently all collected from a single organization to avoid the use of expensive Internet bandwidth (see, *e.g.,* [1, 10]). Our approach is to study hundreds of millions of pages collected from millions of Web sites. There are a number of reasons why studying only a few million pages leads to unsatisfactory results.

First, due to our initial connection to AltaVista, much of our work in Web archaeology has been about Web search. In this context, it is particularly important to use large data sets. As the document sets scale, it becomes increasingly difficult to maintain precision because an increasing number of irrelevant documents get ranked highly for accidental reasons. This precision was an issue, for example, in the Topic Distillation algorithm of Bharat and Henzinger [6], who added special mechanisms to prevent so-called "topic drift." The problem of precision is made worse by spammers, who try to mislead search engines. Robustness to spam is an important attribute of ranking algorithms, an attribute that is not well tested with small data sets.

Second, data sets taken from the Web sites of a single organization are particularly suspect because they are unusually free of spam and of other types of documents that unexpectedly mislead search algorithms. In addition, empirically, we have found that 80% of hyperlinks on the Web do not cross host boundaries, which can make the Web appear to be better connected than it really is. For example, the study reported in *Nature* [1], which examined the connectivity of Web pages within the single domain `nd.edu` (the University of Notre Dame), concluded that the Web was much more strongly connected than the "Bow tie" study [9], which studied a much broader data set.

Third, large data sets not only test the functionality of algorithms better than smaller sets do, they also test scalability. In the context of the Web, the utility of a ranking algorithm that cannot scale beyond a few million documents is limited. Broder *et al.* [8], for example, reported on the use of *shingleprints* to find near-duplicate pages in a large collection. From the start, shingleprints were designed to leverage random sampling as a tool for scalability. Yet, as successful as the original scheme was, major improvements beyond the original algorithms were ultimately required to keep up with the scaling goals of AltaVista [17]. Similarly, our connectivity database [5] has undergone two major improvements to support scalability [20].

For these reasons, we believe that it is important to study data sets that are as large as possible. In the context of the Web, this will always be a relative term, something toward which to continually strive. As a first approximation, larger data sets are more likely to be representative than smaller ones. We routinely study data sets consisting of hundreds of millions of documents, and we aim to scale up to multi-billion document collections.

2.2 Features

Web archaeology involves answering questions like “How strongly connected is the Web graph?” and “How can we discover pages that are near-duplicates of one another?” Answers to such questions can often be found by studying particular features of documents. A feature is a value or set of values computed from the contents of a document. Examples of features we and our colleagues have studied include links [5], which capture connectivity, shingleprints [8], which capture syntactic similarities, and term vectors [22], which capture semantic similarity. The first question above can be answered by looking at links; the second has at least one answer in terms of shingleprints. As the case of shingleprints illustrates, cleverly-defined features can result in algorithms that are both functionally robust and efficient to implement. Also, while feature definitions are often motivated by a particular question, the best features are those that can be used to answer many questions.

Our inclination to think about Web archaeology in terms of document features has influenced the tools we have built. In particular, we have put significant effort into building *feature databases*, which provide access to particular features extracted from large collections of documents.

We have built four feature databases to date: the URL database, the link database, the host database, and the term vector database. The URL database maps URLs to *URL ids*, which are densely allocated integer identifiers. These URL ids are used as keys into the other three databases. One of these other databases is the link database, which maps URL ids to *outlinks* (the set of URLs pointed to by the URL) and *inlinks* (the set of URLs pointing to the URL). This link database has proven invaluable in building a number of applications (*e.g.*, finding related Web pages [11], and studying properties of the Web Graph [9]). Another feature database is the host database, which maps URL ids to *host ids*, and host ids to a set of URLs. Each host id indicates a unique host name in the URL. The last feature database is the term vector database, which maps URL ids to term vectors.

Feature databases are crucial to performing Web-scale research. Features can usually be stored in a fraction of the storage required by the entire page. For example, while a typical HTML page is 10–15 kilobytes, the link database stores both the inlinks and outlinks of a typical page in under 50 bytes (about 1.5 bytes per link per direction). This compactness allows us to study data sets containing many more documents than we can store on disk. But even if disk space were not an issue, feature databases support Web-scale research by providing *fast* access to features. Our link database, which has been designed to fit in main memory (on large machines!), allows standard graph algorithms to run quickly even on very large data sets; for example, we can compute the strongly-connected components

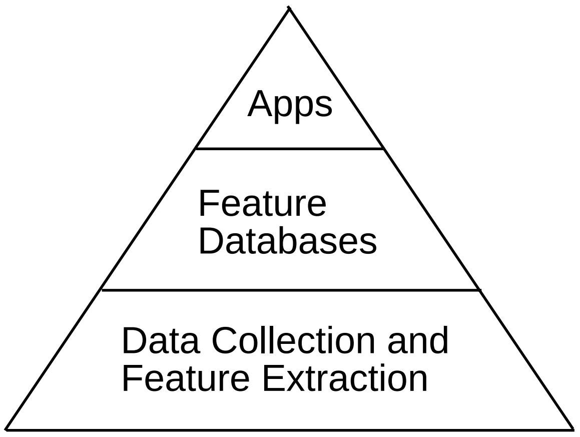

Figure 1: Layers of Web archaeology software

of a 6-billion link database in under half an hour.

2.3 Tools for Web Archaeology

Figure 1 depicts the relationships among the tools we have built for Web archaeology. On top are the applications that generate the Web archaeology results. These applications are built on top of feature databases that support efficient manipulation of large collections of features. The feature databases in turn are built on top of tools that pull data from the Web.

We employ two tools for collecting data from the Web, Compaq's Web Language [16] and Mercator [14, 18]. Compaq's Web Language is a Web scripting language that provides primitives for manipulating HTML parse trees, supports cookies, and interacts with HTML forms; it is well-suited to extracting pages and features from interactive Web sites. However, Compaq's Web Language does not download pages in high volume.

Mercator, on the other hand, is an extensible, scalable crawler capable of downloading tens of millions of documents in a day. One aspect of Mercator's extensibility is its ability to run user-defined feature extractors to customize the data that is saved during a crawl. For example, it can be configured to save an entire document, or just the document's links, its shingleprints, or its term vectors. Mercator is also extensible in its ability to handle new Web protocols with little additional programming. If, for example, we decide we want to download streaming multimedia files during a Mercator crawl, we can easily plug in new modules that handle the multimedia protocols, such as MMS (Windows Media), RTSP (RealMedia and

QuickTime streaming protocol) and PNM (RealMedia legacy protocol).

We use Mercator to download the bulk of our pages, and Compaq's Web Language for specialized crawls requiring intricate interaction with servers.

Although the layering in Figure 1 was not devised *a priori*, it does reflect the characteristics of our research style described in the previous subsections. Of course, the feature layer reflects our focus on features. The pyramid shape represents the data reduction needed to perform Web-scale experimentation: as one moves up the pyramid, one deals with terabytes at the data collection layer, tens or hundreds of gigabytes at the feature layer, and finally human-graspable information at the application layer.

3 Challenges of Web-scale Crawling

In the process of building and using our Web archaeology tools we have faced several significant challenges, most of them pertaining to data collection and storage.

Cost. Web-scale research is costly. To run efficiently, we run applications on large-memory machines, typically with 8GB or more of main memory. While the cost of such machines is dropping, they are several times more expensive than PC workstations. Disk storage is another expense: our cluster currently has over 5 terabytes of disk space. Finally, high-volume crawling requires on the order of 100 megabits/second or more of Internet connectivity, which costs between $300–$500 per megabit/second per month.

Internet bandwidth. Paying for connectivity is just one of the problems associated with good Internet bandwidth. In many organizations, the networks to which machines are connected are separated from border routers by several other routers and perhaps a firewall. Often, one or more of these intermediaries crash when suddenly exposed to the sustained, intense traffic generated by a crawler. We have eliminated these intermediaries, creating a special subnet outside our corporate firewall connected to dedicated routers. DNS (Domain Name Service) performance can also be a problem with crawling [14].

Content quality. It is difficult to ensure that a high-volume crawl is downloading quality content. The first difficulty is to define a metric for quality that is suitable for a broad range of research. Once a suitable metric is defined, maximizing it is also difficult. Dynamically-generated content is particularly vexing in this regard. The difficulties include:

- **Low quality dynamic pages**. *Spider traps* are page-generating scripts that generate large volumes of relatively uninteresting pages. For example, state information, such as session identifiers embedded in URLs, can defeat a crawler's tests for previously visited documents, yielding infinite copies of what is essentially the same page. Calendar programs that allow one to follow "next month" links into the year 3050 also yield relatively uninteresting information. There are even some intentionally-created traps (such as `http://www.die.net`).

- **Hidden, high quality dynamic pages**. At the other end of the spectrum are high quality dynamic pages that are hidden behind passwords or forms. For example, the content of many newspapers is password protected, while many product support pages are accessed through query forms that cannot be filled in by the typical Web crawler.

In our case, the breadth-first nature of our crawler fetches pages in an order roughly correlated with PageRank [7, 19], which is a generally accepted metric of quality. Our breadth-first crawl together with heuristics for normalizing URLs prevents us from going too deeply into uninteresting data. We currently make no attempt to access information that is hidden behind passwords or forms.

Avoiding downtime. Once the crawler is saturating its available bandwidth, the next challenge is to keep it running. In our case, storage management has been the central obstacle to keeping a crawl running. In our current configuration, the crawling machines deposit the data they collect onto their local disks. The machines that will ultimately hold that data periodically wake up and pull the data from those disks (different machines take different data). If something goes wrong that prevents data from being pulled from a crawler, the crawler's disks fill up and the crawl stops.

Mercator supports checkpoints, which have been crucial to organizing our data-movement software and to maintaining reliability in the face of environmental failures. Periodically (in our case, every six hours), the crawler writes its current state to disk, including its sets of URLs seen and URLs yet to be crawled, plus any features that have been extracted. If something goes wrong between checkpoints, the crawler falls back to the state at the previous checkpoint and continues forward from there.

Our data movement software synchronizes itself to these checkpoints, moving data only after it has been committed by the crawler. This synchronization ensures that we do not get duplicates after a restart. Also, we have a number of programs that check the environment for failures, for example, by measuring the amount of disk space left on the crawling machines and the connectivity between the crawlers and the Internet. As soon as a problem is spotted, the crawler is made to checkpoint immediately, and an operator is notified.

E-mail. A final operational issue when crawling is dealing with e-mail. No matter how polite a crawler, it will stand out in the log of the Web sites it visits.

Most Web masters do not care, but when crawling millions of sites, one is bound to get mail. Some Web masters are curious: “What are you up to?”, while others are suspicious: “What are you *up* to?”. Some are desperate, as was a government lawyer who accidentally posted confidential information and wanted it expunged from our collection, or the merchant whose shopping cart application crashed after our crawler added over 800 bottles of some vitamin to its “cart.”

The HTTP standard recommends that a robot include an e-mail address in its HTTP request. Mail to the address that we provide results in an automated reply of a standard response. This response describes the purpose of our crawls and contains instructions for using robots.txt to turn our crawler away. Our automatic response seems to satisfy most e-mail inquiries, giving us more time to respond promptly and usefully to the rest.

4 Evolving the Framework

We have been fairly successful doing Web archaeology with the three-level pyramid model described in Section 2, but experience has shown us a number of shortcomings of the model.

One of the biggest problems is the synchronization of data collection and feature extraction. Crawling a large number of documents takes weeks to months. It’s also expensive, both in terms of our bandwidth and in terms of the cycles and bandwidth at the crawled sites. Thus, it becomes difficult to experiment with new features or modify the definitions of old ones. The more we make our tools available to other researchers, the more prominent this problem becomes, because different researchers have different requirements as to how the downloaded documents are processed.

Another problem is ease of access to data and management of storage. We have, at various times in the past, had large numbers of downloaded documents saved in files. Like most researchers, we aggregate large collections of documents into files of around ten thousand documents. Working with these files is tedious. One needs to write parsers to extract individual documents from the aggregate files, and then (often) parse the HTTP headers and the HTML. To make it all run in a reasonable amount of time, one has to be fairly careful about how things are coded and how the file system gets driven. This overhead inhibits the spontaneity and increases the latency of research.

Finally, the three-level pyramid model does not support well the study of the dynamic nature of the Web, that is, the study of how features change over time. Although the state of the art has been to study snapshots of the Web, we believe that more dynamic studies will yield interesting results.

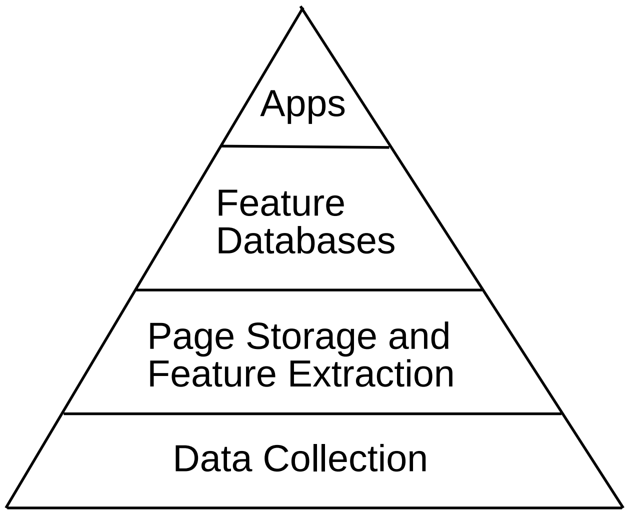

Figure 2: Adding the data storage layer.

Our solution to all of these problems is to introduce a new layer into the pyramid for the storage of raw documents, as shown in Figure 2. We also move the bulk of feature extraction from the data collection layer to the storage layer. By storing the raw data, we are able to compute new kinds of features on demand, addressing the first problem. To address the second problem, an appropriate data storage layer tool hides the problems of storage management, and provides easy access to the data via a high-level query language, allowing researchers to focus more on defining interesting features and less on low-level coding. Finally, to address the third problem, the data storage layer can support storage and retrieval of multiple versions (or *instances*, in HTTP parlance) of documents. We can then study how documents change over time.

We are currently building a storage layer tool, called the Web-in-a-box (WIB). The WIB interfaces to the feature databases above it (and also is directly accessible to the application layer, if desired), and it interfaces to the data collection tools below (in our case, primarily the Mercator crawler).

4.1 Goals of the WIB

To support Web archaeology research well, we believe a storage layer tool should have the following features:

- A simple query language that makes the data accessible to non-programmers;

- The ability to handle historical data, to study the evolution of the Web over time;
- An extensible interface that supports extracting new document features as they are designed by the users;
- The ability to run repeatable experiments and small experiments.

The first phase of the WIB project, which provides the interface to feature databases and applications, largely deals with these issues. A prototype of this system is complete and its salient features are described below.

The second phase of the WIB project is concerned with the interface to the data collection layer of tools and with building an affordable infrastructure for the storage containing raw documents and indices on it. Because we are interested in studying multi-billion document collections, this storage system must be capable of storing hundreds of terabytes of data.

The WIB project is quite similar at a high level to the WebBase project at Stanford [15]. Both systems are targeted at maintaining large, shared repositories of Web pages. We differ in the particular aspects of the problem that we emphasize, in the details of our solutions, and in the scale that we are aiming to support.

4.2 The WIB Data Model

To feature databases and application programs, the WIB looks, abstractly, like a relational database with a single table. The rows of the table represent document instances (HTTP responses for particular URLs at particular times). The columns of the database are the features of the documents. These include features that are easily extracted from the HTTP request and response (*e.g.*, the URL, download date, document type, and document contents), as well as features defined by the users of the WIB.

Users may add columns to the WIB by writing special-purpose feature extraction code, called *analyzers*. In this aspect, the WIB is unlike traditional databases because its set of columns is extensible. Analyzers are pieces of Java code written to conform to a well-defined interface. A given analyzer invocation computes one or more features for a single document (row in the table) and returns its results back to the WIB evaluation engine for storage and access. WIB analyzers are very similar to (and inspired by) analyzer modules in Mercator. They are different in that the WIB analyzer writer need not be concerned with implementing checkpointing code nor with managing the storage for analyzer output. The results of calling an analyzer currently are used for evaluating a single query and then discarded. One could also imagine storing them and adding them to indices.

```
select name, content where
    match(name, "^http://") && match(name, ".ram\$");
select mmlinks where
    match(http_content_type, "^text/");
```

Figure 3: Example WIB queries

Some examples of analyzers that we've written for the WIB include link extractors (to extract links of various kinds from the text of HTML pages), term vector analyzers (to compute the term vectors of HTML pages), and a header analyzer to extract the header values from an HTTP response. In the same way, we could write a shingleprint analyzer, or an analyzer that tries to figure out in what human language a page is written.

4.3 The WIB Query Language

Information in the WIB is retrieved via a high-level query language. The language supports simple projections, that is, the selection of data from one or more of its columns. A query specifies a criterion for selecting rows and the names of the columns to return for rows that match the criterion. The selection criterion for a row is a boolean expression whose base-level components are relational expressions and pattern-matching expressions on column values. Both built-in and user-defined columns may appear in the selection criterion.

Figure 3 shows two example queries in the WIB query language. The first query selects the documents whose name (*i.e.*, URL) starts with `http://` and ends in `.ram`. For each selected document, the WIB outputs its name and content (the HTTP response in its entirety). In this query, all of the columns are built-in. The second query selects documents whose content-type header has `text/` as a prefix and outputs the `mmlinks` column for each of those documents. `mmlinks` is computed by a user-defined analyzer to be the multimedia links in the document (*i.e.*, those links starting with `pnm:`, `rtsp:`, or `mms:`). `http_content_type` is also computed by a user-defined analyzer that parses the header from the "content" column for the document.

To support small Web archaeology experiments, the WIB supports a form of querying that returns a pseudo-random sample of the results matching a selection criterion. The user specifies two additional parameters to the query: the probability of choosing a document matching the criterion, and a seed for a pseudo-random number generator. If the user knows roughly how many documents match the query (which can be obtained by asking the WIB to count the results of the query),

```
[samplingProb=0.1, samplingSeed=789]
   select name, content where
      (match(name, ".html$")
      && insertionDate < Date("2000/11/01"));
```

Figure 4: Example sampling repeatable sampling query.

then the user can choose a selection probability that will produce a result set of approximately the desired size. We believe this sampling capability is very important for an extremely large data collection, where a full query could take many hours or days to execute and/or generate more output than the user wants to process. It allows users to debug new algorithms or quickly approximate results.

The WIB supports repeatable sampling, that is, sampling that produces the same set of result documents. Repeatable sampling is useful for comparing different algorithms on the same data set, for debugging, and for revalidating earlier results. When a query is sampled, the system outputs a date and a seed along with the samples. To reproduce the samples, the user supplies these parameters. The seed controls a pseudo-random number generator. The date is used to eliminate documents that may have been added to the collection since execution of the query.

Figure 4 shows an example sampling query that is repeatable. The elements in square brackets are the values of the sampling parameters. In this case, the probability of selecting any document from the set of documents matching the selection criterion is 0.1 and the seed for the pseudo-random number generator is 789. The set of documents from which to choose the sample is all documents whose name ends in `.html` that were inserted into the WIB on or before November 1, 2000.

Note that the WIB query language intentionally does not support the creation of multiple tables, or any form of join operation. We believe that the WIB is the wrong place to do joins: the data in it is too massive and unstructured. The WIB should instead be used to extract features to be put into a specialized database such as the Connectivity Database, or into a general purpose database such as a SQL database, where joins can be performed efficiently. At some point in the future we may support adding additional columns to the WIB whose values come from external computations. These columns would effectively allow someone to cache the results of an expensive join computation in the WIB. For example, after the Connectivity Server computes the inlinks of a document, they could be stored in a new column for the document and later retrieved by queries.

The WIB query language also does not, in general, support aggregation operations. An aggregation operation is one where the results of multiple rows are summarized through some operation (*e.g.*, to compute an average value, a histogram, or some other statistical function). Aggregation complicates the processing of documents, and it is something that users can implement outside of the WIB if it is desired. To help users pick probabilities for random sampling, we could support approximate counting of the number of documents that satisfy a query, which we can do efficiently within the WIB. At some future time we may also support a wider range of aggregation operations within the WIB if it seems desirable based on more experience.

4.4 Current State

As of this writing, the first phase of the WIB system exists as a command-line program that supports the full WIB query language and the extensible feature-extraction facilities. The program does no storage management. The program is given a query and a set of archive files to which to apply the query. Each archive file contains a sequence of documents, and is about 100 megabytes in size. Output from the query is written to a directory in the file system. Archive files typically contain results from Web crawls, performed outside of the WIB. In addition to providing a nice query interface for processing archive files, this version also supports random sampling through precomputed indices. The indices allow the system to efficiently skip over documents that are not in the sample.

5 Conclusions

This paper discusses the difficulties of studying the content of the Web and describes some of the tools and techniques we have developed to tackle this problem.

In addition to the tools discussed here, we have often wished for a scalable storage cluster designed to support data-parallel operations on write-once data sets. Such a cluster would be an ideal infrastructure for the page repository described in Section 4.3.

A Web page repository that supports easy access to and storage for billions of documents would allow us to study larger data sets. A repository that also contains multiple instances of pages, obtained by continuous crawling of the web, would let us study how the Web evolves over time. Mercator has mechanisms to support continuous crawling, but we'd have to develop policies for those mechanisms, to decide, for example, which web pages to revisit and how often. We would also then want to update our feature databases to support the continuous influx of data. Even without these extensions, however, our infrastructure has proven useful for tackling Web archaeology experiments at extremely large scale.

References

[1] R. Albert, H. Jeong and A.-L. Barabasi. Diameter of the World Wide Web. *Nature*, 401 (1999) pages 130-131.

[2] K. Bharat and A. Broder. A technique for measuring the relative size and overlap of public Web search engines. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, pages 379-388. Elsevier Science, April 1998.

[3] K. Bharat and A. Broder. Mirror, mirror on the Web: A study of host pairs with replicated content. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, pages 501-512. Elsevier Science, May 1999.

[4] K. Bharat, A. Broder, J. Dean, and M. Henzinger. A comparison of techniques to find mirrored hosts on the WWW. In *1999 ACM Digital Library Workshop on Organizing Web Space (WOWS)*, Berkeley, California, August 1999.

[5] K. Bharat, A. Broder, M. Henzinger, P. Kumar and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, pages 469-477. Elsevier Science, April 1998.

[6] K. Bharat and M. R. Henzinger. Improved algorithms for topic distillation in hyperlinked environments. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Australia, pages 104-111. ACM Press, August 1998.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, pages 107-117. Elsevier Science, April 1998.

[8] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic Clustering of the Web. In *Proceedings of the 6th International World Wide Web Conference*, Santa Clara, California, pages 391-404. Elsevier Science, April 1997.

[9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopolan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the Web. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, The Netherlands, pages 309-320. Elsevier Science, May 2000.

[10] J. Cho, H. Garcia-Molina and L. Page. Efficient crawling through URL ordering. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, pages 161-172. Elsevier Science, April 1998.

[11] J. Dean, M. R. Henzinger. Finding related pages in the World Wide Web. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, pages 389-401. Elsevier Science, May 1999.

[12] M. Henzinger, A. Heydon, M. Mitzenmacher and M. Najork. On near-uniform URL sampling. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, The Netherlands, pages 295-308. Elsevier Science, May 2000.

[13] M. Henzinger, A. Heydon, M. Mitzenmacher, and M. Najork. Measuring index quality using random walks on the Web. In *Proceedings of the 8th International World Wide Web Conference*, Toronto, Canada, pages 213-225. Elsevier Science, May 1999.

[14] A. Heydon and M. Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, 2(4), pages 219-229. Baltzer Science Publishers, December 1999.

[15] J. Hirai, S. Raghavan, H. Garcia-Molina, A. Paepcke. WebBase: a repository of Web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, The Netherlands, pages 277-293. Elsevier Science, May 2000.

[16] T. Kistler and J. Marais. WebL–A programming language for the Web. In *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, pages 259-270. Elsevier Science, April 1998.

[17] M. Manasse, personal communication.

[18] M. Najork and A. Heydon. High-Performance Web Crawling. Chapter 2 in J. Abello, P. Pardalos, M. Resende (editors), *Handbook of Massive Data Sets*, Kluwer Academic Publishers, 2001.

[19] M. Najork and J. L. Wiener. Breadth-first search crawling yields high-quality pages. In *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, pages 114–118. ACM Press, May 2001.

[20] K. Randall, R. Stata, R. Wickremesinghe, J. Wiener. The Link Database: Fast access to very large Web Graphs. Research Report 175, Compaq Systems Research Center, Palo Alto, CA, September 2001.

[21] E. J. Ray, D. S. Ray, and R. Seltzer. *The AltaVista Search Revolution*. Osborne McGraw-Hill, Berkeley, California, 1998.

[22] R. Stata, K. Bharat, and F. Maghoul. The Term Vector Database: Fast access to indexing terms for Web pages. In *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, The Netherlands, pages 247-255. Elsevier Science, May 2000.

[23] J.-M. Van Thong, D. Goddeau, A. Litvinova, B. Logan, P. Moreno and M. Swain. SpeechBot: A speech recognition based audio indexing system for the Web. In *International Conference on Computer-Assisted Information Retrieval*, Recherche d'Informations Assistee par Ordinateur (RIAO), Paris, France, April 2000.