WebCrawler: Finding What People Want

Brian Pinkerton

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
University of Washington
2000

Program Authorized to Offer Degree:
Department of Computer Science & Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Brian Pinkerton

and have found that it is complete and satisfactory in all respects
and that any and all revisions required by the final
examining committee have been made.

Co-chairs of the Supervisory Committee:

_____

Edward Lazowska

_____

John Zahorjan

Reading Committee:

_____

Edward Lazowska

_____

John Zahorjan

_____

David Notkin

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microfilm and/or (b) printed copies of the manuscript made from microform."

Signature: _____

Date: _____

University of Washington

Abstract

WebCrawler: Finding What People Want

Brian Pinkerton

Chairpersons of the Supervisory Committee:
Professor Edward Lazowska
Professor John Zahorjan
Department of Computer Science & Engineering

WebCrawler, the first comprehensive full-text search engine for the World-Wide Web, has played a fundamental role in making the Web easier to use for millions of people. Its invention and subsequent evolution, spanning a three-year period, helped fuel the Web's growth by creating a new way of navigating hypertext.

Before search engines like WebCrawler, users found Web documents by following hypertext links from one document to another. When the Web was small and its documents shared the same fundamental purpose, users could find documents with relative ease. However, the Web quickly grew to millions of pages making navigation difficult. WebCrawler assists users in their Web navigation by automating the task of link traversal, creating a searchable index of the web, and fulfilling searchers' queries from the index. To use WebCrawler, a user issues a query to a pre-computed index, quickly retrieving a list of documents that match the query.

This dissertation describes WebCrawler's scientific contributions: a method for choosing a subset of the Web to index; an approach to creating a search service that is easy to use; a new way to rank search results that can generate highly effective results for both naive and expert searchers; and an architecture for the service that has effectively handled a three-order-of-magnitude increase in load.

This dissertation also describes how WebCrawler evolved to accommodate the extraordinary growth of the Web. This growth affected WebCrawler not only by increasing the size and scope of its index, but also by increasing the demand for its service. Each of WebCrawler's components had to change to accommodate this growth: the *crawler* had to download more documents, the *full-text index* had to become more efficient at storing and finding those documents, and the *service* had to accommodate heavier demand. Such changes were not only related to scale, however: the evolving nature of the Web meant that functional changes were necessary, too, such as the ability to handle naive queries from searchers.

Table of Contents

List of Figures

Preface

This dissertation chronicles the evolution of a research project — WebCrawler, the World-Wide Web's first comprehensive full-text search engine — from its beginning as a small experiment to its maturation as a commercial product. In one sense, this dissertation adheres to a conventional format: it describes an important problem, details a solution and an implementation, and provides an evaluation of the solution. However, this dissertation is different than most in that the evaluation is based not on the solution's theoretical or laboratory results, but on its actual implementation in a production setting. One difficulty with this approach is that the work was tied to the environment in which it was developed. Indeed, several theoretically promising solutions described in the dissertation were not deployed, even though they may have been feasible in another environment. However, the value of this approach lies precisely in its close tie with the implementation environment, where the success or failure of a particular solution is based on the complex and evolving set of trade-offs that occur in real engineering environments.

Acknowledgments

WebCrawler was a team effort. This dissertation and the work it describes reflect not only the hard work and ideas of a team of people, but also the guidance, input, and patience of others.

John Zahorjan and Ed Lazowska helped guide my wandering thoughts and made me get the message right. Over the years, they have been incredibly patient, understanding, and supportive as my path consistently veered away from the UW. Larry Ruzzo, David Notkin, and Hank Levy provided leadership, insight, and support throughout my long graduate career. Ger van den Engh and Barb Trask taught me genomics, and provided the environment in which WebCrawler was first conceived.

WebCrawler would have remained a small graduate student's project without the hard work and dedication of the WebCrawler Team. Special thanks to: Rob Wilen, for believing in the big vision; Scott Stephenson, for keeping the roof on; Keith Ohlfs, for Spidey; Dennis Gentry, for creative hardware design; Martijn Koster, for an unforgettable programming marathon; Andrew Pinkerton, for an unwavering commitment to non-stop operation; David Powell, for making the long commute; Mustafa Vahanvaty, for seeing order where there was none; Adam Hertz, for making the big vision happen; Kate Greer, for adding some style; and Jim Rheinhold, for watching out for the customers!

Turning a research project into a product was hard, but turning that product into a dissertation proved just as difficult. Thanks to: Lizza Miller, for getting me started; Debi Pinkerton for inspiring me and showing me it wasn't too late to finish my degree; Frankye Jones and Dave Rispoli, for their administrative wizardry; and Tad Pinkerton and Helen Tombropoulos, for their eagle-eyed editing.

Any conceptual mistakes in this dissertation are my own. However, any typographical “features” can be attributed to my faithful editorial assistants, Filaki and Zouzouni, whose attempts at typing kept me laughing. Raven provided similar assistance during WebCrawler’s original programming.

A Ph.D. is a journey that few imagine to be in reach. For preparing me for this journey, making sure I knew it was achievable, and encouraging me along, I would like to thank my parents, Tad and Hannah, and my grandparents, Ken & Mildred Pinkerton and Grant & Harriet Gale.

WebCrawler is now both a product and a dissertation. None of it would have happened without the inspiration, vision, and insight of my wife, Rhea Tombropoulos. She has helped me make the most of this opportunity, and has made these years better than I could have imagined. Thank you.

Chapter 1

Introduction

The widespread adoption of the World-Wide Web (the Web) has created challenges both for society as a whole and for the technology used to build and maintain the Web. On a societal level, the Web is expanding faster than we can comprehend its implications or develop rules for its use. The ubiquitous use of the Web has raised important social concerns in the areas of privacy, censorship, and access to information. On a technical level, the novelty of the Web and the pace of its growth have created challenges not only in the development of new applications that realize the power of the Web, but also in the technology needed to scale applications to accommodate the resulting large data sets and heavy loads. The disciplines of distributed systems, algorithms, and information retrieval have all contributed to and benefited from solutions to these new problems.

This dissertation describes WebCrawler, the Web's first comprehensive full-text search engine. WebCrawler has played a fundamental role in making the Web easier to use for millions of people. Its invention and subsequent evolution, from 1994 to 1997, helped fuel the Web's growth by creating a new way of navigating hypertext: searching. Before search engines, a user who wished to locate information on the Web either had to know the precise address of the documents he sought or had to navigate patiently from link to link in hopes of finding his destination. As the Web grew to encompass millions of sites, with many different purposes, such navigation became impractical and arguably impossible. I built WebCrawler as a Web service to assist users in their Web navigation by automating the task of link traversal and creating a

searchable index of the Web. Conceptually, WebCrawler is a node in the Web graph that contains links to many sites on the Web, shortening the path between searchers and their destinations.

In addition to its impact on the Web, WebCrawler also made a number of contributions to computer science. Some of these contributions arose from new problems and solutions emanating from WebCrawler's novel mission, while others arose when theoretical research on hypertext, information retrieval, and distributed systems fell short in meeting the demands of the Web. WebCrawler broke new ground as the first comprehensive full-text search system for the Web. In developing WebCrawler, I identified several new problems that had to be solved, such as the need to carefully choose a subset of documents to put in a collection and the need to provide users with easy access to the relevant documents in this collection. WebCrawler also stretched the limits of traditional operating systems and techniques, requiring new systems that were at once scalable and fault-tolerant; applying these systems in WebCrawler's operating environment meant simplifying and focusing many of the assumptions of past practice to create the right mix of scalability and fault tolerance.

WebCrawler's contributions overlap three areas of computer science: hypertext, information retrieval, and distributed systems. In the following three sections, I introduce the challenges the Web posed to each of these areas.

1.1 Hypertext and the Web

Today, we think of the Web as a typical hypertext system: a widely distributed collection of documents, related by links occurring in the body of each document. In the course of reading a hypertext document the reader can choose to follow the embedded links to other related documents. This modern notion of hypertext is remarkably similar to concepts foreseen by some of the original hypertext thinkers and to many of the subsequent hypertext systems [Bush 1945, Nelson 1972, Berners-Lee 1992].

The key distinction between the Web and the hypertext systems that preceded it is that the Web is decentralized across administrative boundaries. With the onset of the Web, publishers could easily create and publish documents that anyone could instantly read. Furthermore, the structure of the system — and its

early principles — encouraged publishers to include links that crossed administrative boundaries. Today, documents are located around the world and are created by millions of authors and consumed by even more readers. Figure 1.1 shows this incredible exponential growth: between 1994 — when WebCrawler was started — and 1998, the Web grew three orders of magnitude.

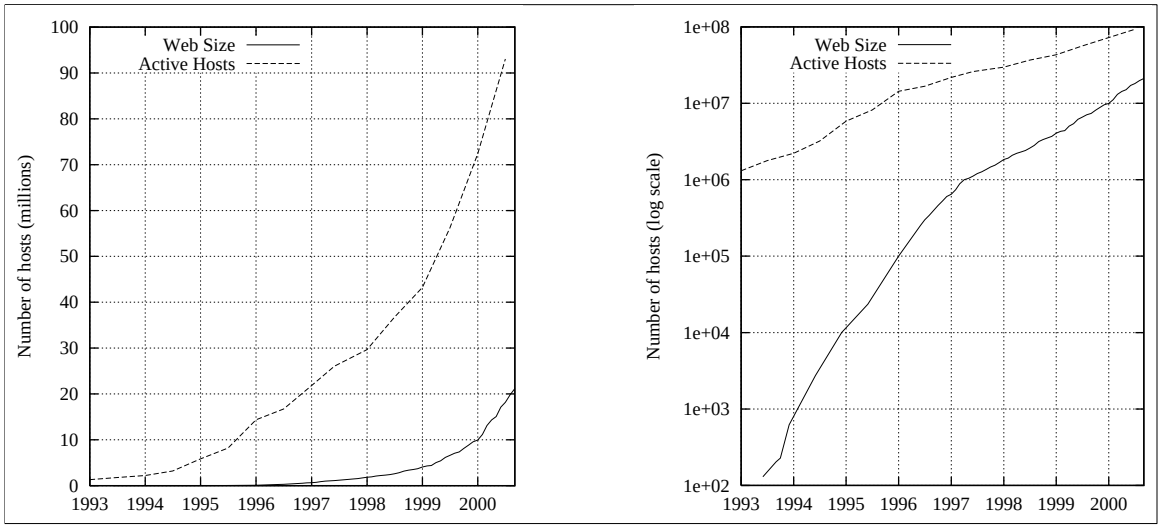

Figure 1.1: Growth of the Internet (number of active hosts) and the Web (number of servers) from 1993 to 2000. These graphs show the same data, represented on a linear scale on the left and on a logarithmic one on the right. These data are aggregated from multiple sources [Gray 1996, Netcraft 2000, ISC 2000].

The large size of the Web, and the resulting diversity, posed unique problems whose solutions were not apparent in past research. In particular, finding resources in such a distributed hypertext was hard — and became progressively more difficult as the system grew. To solve this problem, WebCrawler went beyond hypertext: combining hypertext with search, it reached a new level of sophistication that helped fuel the Web's growth. However, performing this task well meant changing the conventional idea of search originally defined in information retrieval research.

1.2 Information Retrieval

Like the researchers in hypertext but generally independent of them, researchers in information retrieval (IR) have also sought to match users with the information they need. Where hypertext uses browsing to accomplish this task, information retrieval has traditionally used search.

The basic problem in information retrieval is to find a subset of relevant documents from a collection in response to a searcher's query. Typically, the query is expressed as a string of words and a set of constraints on metadata (for example, the query *find articles from 1969 on the Apollo program* could be expressed as *date=1969 and type=article and keywords='apollo program'*). The results of a query are a selection of documents from a larger collection. Collections are typically static bodies of documents. Most of the work in IR has focused on systems for storing and viewing documents, methods for processing queries and determining the relevant results, and user interfaces for querying, viewing, and refining results [Salton 1989]. Because the use of computers and powerful retrieval systems had generally been restricted to a small subset of the world's population, past IR work was targeted primarily at expert users such as research librarians who could take advantage of powerful, complex query systems.

The Web changed many of these assumptions. It accelerated the trend of individual ownership of computer systems, increasing the pool of less sophisticated users. Indeed, today the pool of expert users makes up only a tiny portion of the total number of users of the Web. At the same time, search engines like WebCrawler exposed large searchable databases to large numbers of people. The result was a need for simple, easy-to-use search interfaces. The search engines had to assume that searchers would not possess the knowledge to express complex queries. As a result, WebCrawler contains several new interface ideas and algorithms to make the system more usable to the new average user.

1.3 Distributed Systems

WebCrawler functions as a node in a widely distributed system (the Web) and is, itself, implemented as a distributed system. As such, WebCrawler draws on a long history of relevant systems research in the architecture of distributed systems, load balancing, and resource location.

WebCrawler's focus on distribution satisfies three important requirements: scale, availability, and cost. WebCrawler is a large system: it supports hundreds of queries per second against an index that takes days of processing to create. On top of that, the system must always be available because downtime creates ill will with searchers and damages the WebCrawler business. Finally, WebCrawler must solve these prob-

lems at a reasonable cost. These requirements all point to a distributed system as the solution: a distributed system can scale by adding components, services can be replicated or distributed in such a way that the service is always available, and the components of the system can be chosen to minimize the cost of the whole.

Distributed systems have long been candidates for solving the problems posed by scale, availability, and cost. However, many of the solutions do not allow for the dynamic growth, ease of administration, and scale required for systems on the Web. For instance, a tightly coupled distributed system would be difficult to locate in multiple geographic locations. On the other hand, WebCrawler's unique application allows several simplifying assumptions to be made: occasional failures are permitted, response time does not have to be guaranteed, and consistency is not necessary. WebCrawler had to go beyond the solutions of traditional distributed systems research and use the simplifying assumptions to focus on a simple distributed paradigm that has served as a model for current Web systems.

1.4 Contributions

WebCrawler broke ground in several important ways. In this dissertation, I shall focus on five specific contributions:

1. WebCrawler was the first full-text search engine to broadly catalog the Web. Other engines that came before WebCrawler were useful, but were either more difficult to use or cataloged a smaller subset of the Web.
2. WebCrawler introduced the concept of carefully choosing a subset of the Web to index, rather than approaching that choice on a hit-or-miss basis. Indexing the entire Web is difficult. All search engines, including WebCrawler, catalog a subset of the Web's documents. Although this approach makes scaling the system much easier, it does create a new problem: how to select the subset. WebCrawler accomplishes the task of selecting the subset by crawling a larger subset, and using the metadata from that crawl, along with several heuristics, to select a final subset.
3. WebCrawler allows naive searchers to easily find documents on the Web. Designing an information gathering experience that works for them was a new problem and had not been addressed by past

research. User interface, query format, and query specificity all make a difference to the searcher's experience, and WebCrawler optimizes those for the naive user.

4. WebCrawler creates a more useful relevance ranking by combining traditional full-text methods with metadata found on the Web. The results produced by information retrieval are generally ordered on the basis of the relevance of those results to the searcher's query. WebCrawler takes advantage of additional information, including the structure of the Web, to improve relevance ranking.
5. WebCrawler takes an effective approach to achieving scale, load balancing, and fault tolerance that is less complex than what is predicted by the literature. Much of the work in distributed systems was not useful in building WebCrawler. WebCrawler's constraints as a large system — running on commercial hardware in a non-stop environment — made using past research difficult. WebCrawler is implemented using a simple fault-tolerant method that also simplifies growth.

WebCrawler is still in service today as a successful commercial offering of Excite@Home [WebCrawler 2000]. Although its architecture has evolved over the last three years to accommodate new commercial offerings and the ever-changing Web, WebCrawler's service remains true to the fundamental principles outlined in this dissertation. This speaks clearly to the appropriateness of the design decisions that were driven by the research contributions listed above.

1.5 Guide to this Dissertation

In this dissertation, I shall describe the WebCrawler implementation and the contributions outlined in the previous section. In Chapter 2, I survey research and practice related to WebCrawler, and provide a background for many of the topics discussed later. In Chapter 3, I provide a brief outline of WebCrawler's structure. In Chapters 4 through 6, I detail the implementation and evolution of various components of WebCrawler, and I describe how these fit into — or, in some cases, challenged — the research of the time. In Chapter 4, I describe the implementation and evolution of the Crawler component. In Chapter 5, I describe the implementation of WebCrawler's information retrieval component. In Chapter 6, I cover WebCrawler's

implementation as a distributed system. In Chapter 7, I conclude by discussing WebCrawler contributions to the fields of hypertext, information retrieval, and distributed systems, and I also highlight promising areas for future research.

Chapter 2

Background and Related Work

In a large distributed system like the Web, users find resources by following hypertext links from one document to another. When the system is small and its resources share the same fundamental purpose, users can find resources of interest with relative ease. However, with the Web now encompassing millions of sites with many different purposes, navigation is difficult. WebCrawler, the Web's first comprehensive full-text search engine, is a tool that assists users in their Web navigation by automating the task of link traversal, creating a searchable index of the web, and fulfilling searchers' queries from the index. Conceptually, WebCrawler is a node in the Web graph that contains links to many sites on the net, shortening the path between users and their destinations.

The Web's realization and WebCrawler's implementation span work in several areas of traditional computing, namely hypertext, information retrieval, and distributed systems. While WebCrawler draws heavily from this work, its large scale and widespread use has led to interesting new problems that challenge many of the assumptions made in earlier research. This chapter outlines the work on which WebCrawler is based, relates it to more recent works-in-progress, and points out those areas where the realities of the Web challenge past research.

2.1 Navigating Hypertext

Today, we think of a hypertext system as a distributed collection of documents related by links occurring in the text of the document. In the course of reading a hypertext document, the reader can choose to follow the embedded links to other related documents. This modern notion of hypertext, now synonymous with the Web, is remarkably similar in concept to what some of the original hypertext thinkers foresaw.

In 1945, Vannevar Bush proposed a device, the *memex*, that would free users from the "artificiality of systems of indexing" [Bush 1945]. His central thesis was that the tried-and-true catalogs used by libraries were rigid and did not easily fit with the non-linear nature of human thought and exploration. The memex would allow its users easy access to a large collection of online documents, and facilitate the user's navigation among those documents by means of *trails* — links among documents created by the users of the memex system.

Decades later, Bush's concept of the memex grew into the modern notion of hypertext, where authors and readers could enhance the expressive power of texts by adding links among them [Nelson 1981]. Readers browse among documents, following links from one document to another. In modern hypertext systems, links can be either typed or untyped and can themselves bear information. Doug Engelbart was instrumental in moving theory into practice [van Dam 1988].

Early notions of hypertext, beginning with the memex, envisioned link information as being added on top of existing documents, with the authors of the link data generally being separate from the authors of the underlying text. Indeed, in the memex the user himself was responsible for creating his own set of links, though he could reference sets of links (trails) created by other users [Bush 1945].

With the participatory nature of early hypertext systems, there was little distinction between *author* and *reader:* a reader would construct hypertext trails by creating links as he browsed through documents. In more modern hypertext systems, and in the Web in particular, the two roles are generally distinct: readers do not edit the hypertext as they browse through it, nor do they create persistent trails of their browsing history

as envisioned by Bush. Readers of a hypertext navigate among documents, following links and occasionally consulting hyperlinked indexes of different sets of documents.

Most early hypertext systems were constructed against single collections, often in a single subject domain [Coombs 1990, Frisse 1988]. Andries van Dam's Hypertext Editing System, deployed on an IBM/360, was one of the first functioning hypertext systems put to actual use [van Dam 1969, 1988, DeRose 1999]. Xanadu [Nelson 1988], a hypertext system that was designed but never deployed, was intended to be distributed in nature, with content and links crossing administrative boundaries.

With the World-Wide Web [Berners-Lee 1992], Berners-Lee brought the distributed nature of a hypertext system to a new level with documents that are located around the world created by millions of authors and consumed by even more readers. The scale made possible by this system and the resulting diversity have posed unique issues that were not apparent in past research. In particular, finding resources in such a system is difficult, and WebCrawler was designed to solve just this problem.

2.2 Information Retrieval

At its heart, WebCrawler is a full-text retrieval engine: it takes queries expressed as strings of words and looks them up in a full-text index. The field of full-text indexing and information retrieval (IR) has a long history: many of the fundamental underlying algorithms were first described over 25 years ago [Salton 1975, Salton 1989]. Much of this early work is relevant to Web search engines; however, key differences not only between the Web and typical collections but also between the Web's users and typical searchers challenge many of the assumptions underlying conventional IR work.

The basic problem in IR is to find relevant documents from a collection of documents in response to a searcher's query. Typically, the query is expressed as a string of words and a set of constraints on metadata, and the results are a selection of documents from a large collection. Most of the work in IR has focused on systems for storing and viewing documents, methods for processing queries and determining the relevant results, and user interfaces for querying, viewing, and refining results [Sampath 1985].

Two metrics are generally used to evaluate the performance of these IR systems: *precision* and *recall* [Salton 1989]. Precision is the fraction of documents returned in response to a query that are relevant to that query, while recall is the fraction of relevant documents returned against the total number of relevant documents in the entire collection. These metrics are subjective because what is relevant is completely up to the particular searcher. The Web further complicates their use because the set of documents that makes up the collection is not well defined. Nevertheless, they provide a meaningful way of comparing different algorithms given a set of queries and a standard collection.

One of the most influential models for retrieving documents, the vector-space model [Salton 1975], specifies which documents should be retrieved for a given query, and how those documents should be ranked relative to one another in the results list. Variants on this basic method such as boolean logic have been proposed over time [Lancaster 1979, Salton 1983], but the fundamental algorithm remains the same.

Early research in IR concentrated on problems of improving the lives of experienced users of these systems, mainly professional librarians. Indeed, in the seventies and eighties, the only ones with access to such systems were professionals. These searchers had the time to learn a complex user interface and to express their queries in an advanced form.

Easy access to the Web for millions of people, however, has changed both the profile of the average searcher and also the many assumptions underlying conventional IR. No longer can the typical searcher be considered experienced; indeed, experience with WebCrawler suggests that even boolean queries are generally beyond the average searcher's reach. Time is also an important constraint: where a professional searcher might be willing to spend half an hour researching a particular topic, the average Web searcher might spend five minutes. Furthermore, the professional searcher is likely to be more patient, while the Web searcher typically expects an immediate response. With such a focus on experienced searchers, the utility of conventional IR work to systems used by naive searchers is low. Even systems whose purpose was to simplify the life of searchers by presenting a simpler interface were too complicated [Crouch 1989]. Recently, the Text Retrieval Conference (TREC) has begun to include a track that looks at query process as well as outcome; these efforts may shed light on difficult user interfaces [TREC 2000].

Most systems for querying the Web use some of this fundamental IR work in conjunction with new work focused on the unique characteristics of the Web and its population. In particular, work that focuses on the naive and short queries that searchers issue is highly relevant, particularly since conventional IR research has historically focused on long queries with comparatively expert searchers [Crestani 1998]. For example, Wilkinson explores techniques for increasing the performance of short queries in the TREC-4 collection [Wilkinson 1996]. Hearst explores constraints for increasing the precision of 5-to-10-word queries, considered short by the IR community [Hearst 1996]. To date, these efforts seem to have fallen short for lack of hard data about real searchers [TREC 2000].

In Chapter 5, I shall describe how much of the work on full-text retrieval is directly relevant to WebCrawler. Methods to improve relevance ranking and integrate relevance ranking with boolean retrieval are especially important. The work on short queries is useful but does not come close to solving the problem, since the bulk of WebCrawler's searchers issue short queries. Rather than having a system optimized for long queries with acceptable performance on short queries, WebCrawler is a system optimized for short queries with adequate performance on long queries.

2.3 Combining Information Retrieval and Hypertext

Hypertext is oriented toward a browse-and-explore usage model. But this model does not fit the situation in which the searcher is seeking documents about a particular topic in a large heterogeneous collection; the searcher wants to get quickly to an unknown document that is not apparently linked from his current location. To solve this problem, researchers began to focus on the power of navigating hypertext by using a combination of traditional hypertext navigation and full-text indexing. As early as 1988, researchers realized the limitations of stand-alone hypertext systems and appreciated the power of combining hypertext with traditional information retrieval [Halasz 1988, Frisse 1989].

One of the first systems to combine the power of hypertext and information retrieval, IRIS Intermedia [Meyrowitz 1986, Coombs 1990], layered an information retrieval engine on top of a rich underlying hypertext model. Searchers could locate documents using a query interface that gave a list of resulting doc-

uments for a particular search. Any document in that list could be opened, and normal hypertext browsing would follow. The power of this system was derived from the fact that "The Full Text application provides Intermedia users with a simple tool for breaking structural boundaries and locating information quickly" [Coombs 1990]. Though the authors paid attention to the usability of their systems, these early systems were complex by today's standards. For instance, Coombs mentions the need for phrase searching as an alternative to complicated phrase operators, but does not look for further simplifications that would allow the searcher to ignore syntax if he wished [Coombs 1990]. Moreover, Crouch presents a browsing-based alternative to simplify the query experience, but this does little to move toward a very simple user interface [Crouch 1989].

Research in the area of information retrieval has generally dealt with collections that contain structured metadata and relatively unstructured text. When indexing such a collection, each document is indexed on its own merits, separately from the others. However, in a hypertext collection, one has the opportunity to infer new information about a document based not on its text or metadata, but on the structure of the links in and around the document. For example, Croft used a combination of text analysis and link analysis to extend the set of results to documents that would generally not be considered when judged only by their textual content [Croft 1989]. Kleinberg and Page explore models that abandon traditional relevance ranking in favor of sorting documents by other criteria that indicate their quality [Kleinberg 1998, Brin 1998]. Using link information to infer relevance is related to the practice of citation indexing commonly used in the scientific community [Kleinberg 1998, Small 1973]. Lastly, Dean and Henzinger explore an interesting method for using Web structure information to find documents related to a specific document. This approach should be compared to textual methods for implementing the more-like-this feature found on many full-text systems [Dean 1999].

Other systems focus on the unique characteristics of Web content and the difficulties in indexing it. For instance, Broder explores a clever means for finding and eliminating near-duplicate documents [Broder 1997].

As I shall describe in Chapter 3, WebCrawler combines information retrieval and hypertext. One of WebCrawler's primary goals is to provide a simple interface to searching the Web. Most of the traditional information retrieval interfaces, and even the hypertext ones, are too complex for the average WebCrawler searcher. The trend toward using hypertext structure to aid the retrieval process is a valuable one. WebCrawler uses some variants of structure analysis to improve results and to weed out documents not worth indexing.

2.4 Internet Resource Discovery

Much of the work described in previous sections on searching hypertext has at its core a simplifying assumption: that the collections to be searched are stored in a single system and are homogeneous in structure. For a system to be meaningfully used by many publishers, however, the collection must be distributed, complicating the task of providing a centralized search interface. The Web is the ultimate example of a widely distributed publishing system.

Several early search services provide insight into different methods for structuring a distributed search. These systems can be divided into two models based on the way they perform queries across administrative boundaries: In the distributed model, a searcher issues a query at a central location, which in turn sends the query to some number of remote locations for processing; the remote locations then forward their results back to the central location, which merges the results and presents them to the searcher. In the centralized model, one system maintains a complete index at a central location in the network, retrieving the raw material necessary to build that index from remote locations; the searcher's query is then executed against the single central index.

The distributed model has several advantages. First, because it distributes the collection among many databases, the complexity of searching each database is minimized. To the extent that any worse-than-linear algorithms are used at the core of each search, searching many smaller collections in parallel can result in a dramatic improvement in processing time. Second, each database need not use the same method for searching: it only needs to implement a common interface to the central mechanism. Lastly, building the

remote index is accomplished much more easily: each administrative entity maintains its own index, against its own local content. The index can be as complete and current as each location wishes.

However, the distributed model breaks down under extreme scale, when the number of remote databases is very large. Also, getting agreement on a common retrieval protocol among a large number of administratively diverse locations can be difficult. Furthermore, guaranteeing good response time for searchers is impossible when all components of the system are not under the control of a central entity.

In contrast, the centralized model offers none of the advantages in building and maintaining an index that a distributed model offers. However, the centralized model does offer latency advantages for individual queries: a query executed inside a single administrative domain can be run far more quickly than one distributed to potentially thousands of sites. Furthermore, the system for indexing and searching can evolve much more quickly when under the control of that one entity.

Notable examples of the distributed model are WAIS, Netfind, and Harvest. WAIS [Kahle 1991], used a vector-space retrieval model over a distributed collection of indexes. To execute a query, a central query agent communicated with each remote server using an extension of the Z39.50 protocol. Individual servers saw each query and responded with relevant documents from a local collection. While a useful system, WAIS suffered for lack of content because there was no widespread mechanism in place at the time for viewing content from a remote system. Essentially, a WAIS client had to implement a system for viewing content as well as searching for it.

To combat the content problem, Schwartz created Netfind [Schwartz 1994], a system that applied a distributed search strategy to the problem of finding information about people on the Web. Netfind searched for people using a variety of existing mechanisms, so the content was readily available. Like WAIS, Netfind had no central index and, therefore, few consistency problems between the underlying data and the index. Netfind performed a targeted search, in the sense that not every remote database was queried for every search. Instead, Netfind chose to limit each search to the servers closest to where the person could be found. Extending the idea of query routing, Harvest [Bowman 1995] used the idea of a *digest* to send queries only to those servers most likely to have answers to a particular query. Each server in the system created a digest,

an excerpt of the content available at that site, and then sent the digest to a central location. In processing a searcher's query, the central location would then use an index of the digests from all the sites to route queries to the sites with the most potential for producing meaningful results. While this approach was promising, the task of having each site create a digest turned out to be administratively difficult for most sites. Had the functionality to build digests been part of the popular HTTP servers, then perhaps this practice would have been more widespread.

Aside from WAIS, Netfind, and Harvest, most systems maintained a central index, with queries hitting only that index. The trade-off is simple: either one moves the queries to remote locations, or one brings the content to the central site for indexing. Creating a central index generally takes more resources at the indexing site, but can supply answers to searchers more quickly. Examples of the centralized approach include Archie and Veronica, and all Web search engines. I will first describe Archie and Veronica; and then in the next section Web Search engines.

Archie [Emtage 1991] provided a very useful resource: it indexed a collection of documents (FTP files) that were already being heavily used. Archie required a site to be manually included in the central index, but after that initial step, updates to the collection were automatic. In a manner similar to Archie, a system called Veronica provided for searches of the Gopher system [Machovec 1993]. Participating Gopher sites had their content downloaded and indexed at a central location. Searches were then run against the central index and Gopher links were provided to the resulting content.

These approaches provide a basis for some of the decisions made in WebCrawler. For instance, the difficulty of having administratively diverse sites do extra work to belong to an index, as in Harvest, was too much: WebCrawler needed a crawling-based architecture, described in Chapter 4, to ensure sufficient breadth.

2.5 Web Search Engines

With the emergence of the Web, most of the early Internet resource discovery systems faded in importance. The Web had several advantages over these early systems: it provided uniform addressing

across administrative and protocol boundaries; it made publishing documents easy; and, most importantly, it made browsing documents and following links even easier [Berners-Lee 1992].

However, the Web, essentially a large hypertext system, was missing a feature: a universal way to find information. Early on, two resources provided most of the way-finding for users on the Web: the WWW Virtual Library and the NCSA "What's New" Page. Both of these catalogs were essentially large collections of links, though the Virtual Library was organized into a hierarchy. Since they were constructed by humans their quality was uniformly good, but they tended to be out of date and not very comprehensive [Virtual 2000].

In late 1993, three searchable indexes became available: Jumpstation [Fletcher 1993], the WWW Worm [McBryan 1994], and the RBSE index [Eichmann 1994]. Each provided a searchable interface to an automatically constructed index of Web pages. All three employed a crawler to automatically follow hypertext links in Web pages and retrieve the contents of pages for indexing at a central location. However, Jumpstation and the WWW Worm indexed only the titles of pages, while RBSE limited its search to a small fraction of the total Web.

Noting the difficulties with automatically retrieving Web documents, Koster took a different approach with ALIWEB [Koster 1994]. Instead of having a crawling robot retrieve each individual document from a site, the site would itself provide a special file at a known location that could be retrieved via HTTP and indexed by a central service. In this sense, ALIWEB was much like Harvest, though its details made it easier to implement. In the end, ALIWEB was not widely adopted because it depended on site administrators to collect and publish the information to the special file.

Each of these systems immediately preceded the introduction of WebCrawler. Following WebCrawler's introduction, several other successful searching sites were introduced: Lycos [Mauldin 1997], InfoSeek [InfoSeek 2000], Yahoo [Yahoo 2000], AltaVista [AltaVista 2000], and Excite [Excite 2000] being notable examples. Each of these sites employed catalogs that differed in scope, implementation, and presentation. All were automatically constructed full-text indexes, except Yahoo, which used a human-edited catalog similar to the WWW Virtual Catalog.

2.6 Meta-Searching

With the advent of multiple Internet search engines came the inevitable introduction of the meta-search engine. These systems, typified by SavvySearch [Dreilinger 1997] and MetaCrawler [Selberg 1997], query multiple remote search engines to satisfy a single searcher's request. While these meta engines are at first glance similar to WAIS, they are unique in the sense that they are using servers that have overlapping content.

From a searcher's perspective, meta-search is currently desirable because each search engine, although overlapping in content with others, has some unique documents in its index. Also, each search engine updates documents with different frequencies; thus, some will have a more up-to-date picture of the Web. Meta search engines also synthesize the relevance ranking schemes of these search engines and, as a result, can theoretically produce better results. The drawback to using meta-search engines is the same as mentioned above for the widely distributed retrieval models: searchers encounter a slowdown in performance, though MetaCrawler made heavy use of caching to eliminate the performance problems.

Another issue with meta-search engines for pure Web searching is that they rely on the fact that the search engines contain subsets of the Web. Currently, search engines are doing a much better job of indexing larger parts of the Web [AltaVista 2000, Fast 2000, Google 2000]; consequently, the advantage of meta-search engines will be reduced to a problem of relevance ranking. And if a specific retrieval strategy or set of relevance rankings can be shown to work well in a meta-search engine, there is nothing but time and effort that prevents its implementation in a single search engine.

2.7 Distributed Systems

WebCrawler functions as a node in a widely distributed system (the Web) and is, itself, a distributed system. As such, WebCrawler draws on a long history of relevant systems research in the architecture of distributed systems, load balancing, and resource location.

This focus on distribution occurs for three important reasons: scale, availability, and cost. WebCrawler is a large system: it supports hundreds of queries per second against an index that takes days of processing to create. Once created, the index is read-only, and can easily be distributed. Furthermore, the system must always be available: downtime creates ill will with searchers and damages the WebCrawler business. Finally, because WebCrawler is at the core of a business, minimizing cost is essential.

These constraints all point to a distributed system as the solution: a distributed system can scale by adding components, services can be replicated or distributed in such a way that the service is always available, and the components of the system can be chosen to minimize the cost of the whole.

Three aspects of distributed systems research are directly relevant to WebCrawler. First, the overall architecture of these systems has important implications for their ability to scale and provide fault tolerance at a reasonable cost. Second, distributing work across such a system means that balancing the work among individual nodes is important. Finally, getting users to find these nodes in the first place is a difficult problem, particularly when the nodes are widely distributed.

2.7.1 Architecture of Distributed Systems

The research community has seen many different models for handling distributed computation. Generally, research has focused on providing platforms that were available, reliable, and consistent. However, by targeting all three qualities in a single system, one misses the chance to relax constraints where appropriate and ends up with a system that is more expensive to operate than it would otherwise be.

Early research-oriented models involved distributing the operating system at a low level among the different nodes and providing transparent distribution of processes (as in Locus [Popek 1981] and the V Distributed System [Cheriton 1983]) or of objects (as in Eden [Lazowska 1981]). In the more recent systems such as Amoeba and Emerald, the primitives were faster and reasonable throughput could be obtained [Tanenbaum 1991, Jul 1988]. Overall, these systems were designed to support existing applications in a distributed fashion. As such, their primitives were designed to support a variety of distributed models, though their tendency toward transparency made optimization difficult. More recently, the authors of the Coign

system assert that their system can make the kinds of optimizations usually made by programmers [Hunt 1999].

In contrast, distributed shared-memory (DSM) systems treat networks of machines as an opportunity for sharing memory transparently [Dasgupta 1990, Feeley 1995]. In theory, this page-sharing methodology could be useful to a system like WebCrawler; in practice, however, it turns out to be better to take advantage of some application-level knowledge to more precisely distribute work.

The authors of the Grapevine system exploited opportunities in their application to relax some of the assumptions that made implementing some of the general systems difficult [Birrell 1982]. In particular, complete consistency among distributed replicas of the registration database was not always required, allowing the system to provide a substantial savings during update and lookup operations [Schroeder 1984].

The Networks of Workstations (NOW) effort uses a network of ordinary workstations connected by a fast network to perform tasks. Their focus on low-latency networking and operating system support for efficient distribution of work would be useful to systems like WebCrawler should it ever be available in commercial operating systems [Anderson 1995]. NOW is emblematic of a shift toward providing primitives to implement policies, rather than providing complete solutions in which controlling policy can be difficult.

Inktomi uses a different model of scaling to meet both demand and index size. Where WebCrawler uses multiple servers, each with a complete copy of the index, Inktomi uses many servers each with a fragment of the index. Searches are run in parallel on all servers at once (similar to the WAIS model) and are merged, ranked, and presented to the searcher [Brewer 1998].

Recently, Ninja is very promising. It seeks to provide a platform for building scalable Web services by using many easy-to-configure servers. One of the key aspects of interest in Ninja — and one directly related to scale — is that Ninja's goal is to minimize the administrative overhead associated with running a large cluster of systems [Gribble 1999].

This distributed architecture work is relevant to WebCrawler because of the large scale of the WebCrawler service. Its size and implementation, along with the cost of hardware, dictate that WebCrawler run as a distributed system. However, most of the systems that provide complete transparency are difficult to

work with in a low-latency, high-availability environment. For instance, using process migration to move the state of a task from one part of the system to another would be both slower and less fault-tolerant than the implementation WebCrawler needs. As we shall see in Chapter 6, some of the more recent work could be used in WebCrawler if it were adopted in commercial operating systems.

2.7.2 Load balancing

Load balancing is necessary in systems that have a number of resources available to handle tasks and a variety of tasks to distribute among those resources. Load balancing is applied in parallel systems at the processor level; in distributed systems, at the node level. Eager studied and tested different algorithms for distributing work, and came up with accurate models for assigning work [Eager 1986]. In an empirical setting, Weiping has studied load balancing in Amoeba [Weiping 1995] and concluded that centralized decision-making outperforms distributed algorithms.

However, not all load-balancing occurs in tightly controlled situations. For instance, on the Internet where clients make requests of remote systems, the Domain Name System (DNS) has been used successfully to balance requests among several servers by having name servers reply with a list of addresses for a particular name [Mockapetris 1987]. This list can be used by the client to choose among a set of servers for a particular request. Unlike most of the centralized work, this approach relies on the clients to choose their own algorithms for selecting and contacting servers. However, because most clients implement simple deterministic schemes, the name servers themselves have a reasonable degree of control over the outcome. For instance, the name server can repeat addresses in the list to give one server twice as much traffic as another, or it can drop servers from the list when they become too overloaded.

Recently, attention has focused on detailed methods for tracking the load of the constituent systems and on using the tracking data in the name server to deliver a suitably tailored response to the client [Cisco 2000b, Colajanni 1997, Schemers 1995]. These efforts, though useful, help more with response time than with availability because of the persistence of DNS results.

A different class of systems work at the network level by interposing themselves as TCP multiplexors between the client and servers [Cisco 2000a, Alteon 2000]. Such systems provide better load-balancing control than does DNS because they can immediately detect and handle failures rather than having to wait for the list of addresses to time out in a client's cache. Furthermore, these systems can implement a variety of different algorithms for balancing among the servers, including ones that take into account the server's response time to requests. While these systems work well in local environments where they can easily serve as a single point of contact, they do not work well across geographic locations.

WebCrawler, because of its distributed nature, is a heavy user of load balancing. Traditional techniques that involve scheduling tasks — and not requests — are generally too coarse-grained. WebCrawler employs both wide-area strategies based on DNS and local strategies based on TCP switching. However, as I shall describe in Chapter 6, the naive implementations of both of these led us to some necessary enhancements to cope with scale and failure situations.

2.7.3 Resource Location

In a distributed system, components of the system need to find each other; in the case of a client-server system, the clients need to find the servers. In a load-balancing system, the servers with the least load are the most desirable. However, when the clients are far from the servers and the servers are themselves distributed, the cost of communicating between clients and servers becomes a significant factor in response time. In this case, choosing a server that is close can help to minimize the overall delay.

Several aspects of the DNS work described in the previous section bear on the location of clients and servers and on the communication paths among them rather than on the load of the server. Particularly in cases with low-latency requests, the time to transmit and receive the request can dominate the overall transaction time. A few systems have DNS-based solutions that model the communication cost between the client and server, and return server addresses accordingly [Cisco 2000a, Colajanni 1998].

These DNS-based systems can also be used to solve the availability problem: the name servers can give out only addresses of servers that are known to be responsive. Because of the relatively long lifetime of

the DNS information, however, this method only works well when outages are planned and the name servers can be primed with the information in advance.

Recently, several companies have begun hosting Web content from widely distributed locations (e.g. Akamai, Digital Island, I-Beam, Adero). Although each company's system is different, they all attempt to move content closer to the user in the network, thereby avoiding network bottlenecks in serving the content. Some systems, notably those of Akamai and Digital Island, rewrite content from the source system, substituting links to the original system with links to content at distributed nodes [Akamai 2000]. With this approach, the client simply resolves the name for the (hopefully closer) distributed node, and retrieves embedded content from there.

One important issue with these widely distributed systems is that they are less likely to be useful over time, given the trend toward more and more customized content on the Web. Search results, for instance, are dynamically created in response to a searcher's query, and must be served from a location with enough specialized hardware to run the search.

The recent DNS work is highly relevant to WebCrawler, though its prediction of the communication cost between a client and server is still very rudimentary. In fact, as described in Chapter 6, WebCrawler employs some of the wide-area strategies indicated above, and a few other specific optimizations to improve behavior for certain clients.

Chapter 3

An Overview of WebCrawler

In this brief overview of WebCrawler, I shall first describe my motivation for developing WebCrawler. I shall then describe how WebCrawler is used on the Web. Finally, I will give an overview of WebCrawler's implementation. Chapters 4 to 6 will provide details on each of WebCrawler's main components, as well as the factors that led to the important design decisions.

3.1 WebCrawler

WebCrawler is a Web service that assists users in their Web navigation by automating the task of link traversal, creating a searchable index of the web, and fulfilling searchers' queries from the index. Conceptually, WebCrawler is a node in the Web graph that contains links to many sites on the net, shortening the path between users and their destinations. Such a simplification of the Web experience is important for several reasons: First, WebCrawler saves users time when they search instead of trying to guess at a path of links from page to page. Often, a user will see no obvious connection between the page he is viewing and the page he seeks. For example, he may be viewing a page on one topic and desire a page on a completely different topic, one that is not linked from his current location. In such cases, by jumping to WebCrawler — either using its address or a button on the browser — the searcher can easily locate his destination page. Such time savings is especially important given the increase in the size and scope of the Web: between 1994 and 2000, the Web grew in size by four orders of magnitude [MIDS 2000].

Second, WebCrawler's simplification of the Web experience makes the Web a more friendly and useful tool. Navigating the Web by using keyword searches is often more intuitive than trying to use a Uniform Resource Locator (URL) to identify a Web page directly. If users have a good experience, they are more likely to continue to use the Web, and such repeat usage will continue to fuel the growth of the medium. Arguably, search engines like WebCrawler have contributed to the continued simplicity and growth of the Web.

Finally, WebCrawler is useful because it can provide some context for a searcher's particular query: by issuing a well-formed query, a searcher can find the breadth of information about that particular topic and can use that information to further refine his goal. Searchers frequently issue a broad query which they refine as they learn more about their intended subject.

3.2 The Searcher's Experience

WebCrawler was designed to provide a simple and lightweight experience. While it has evolved in many ways since first deployed in 1994, the straightforward and intuitive user interface has always been WebCrawler's hallmark. The 1995 WebCrawler home page, depicted in Figure 3.1, contains a prominent

Figure 3.1: The WebCrawler search page (left) and results page (right) circa 1995. These pages were generated from a working replica of the WebCrawler service that was archived in September, 1995.

search field into which searchers enter their queries. The queries can be a single word, a series of words, a

phrase, or a boolean expression. After typing the query, the searcher presses the search button or hits return, and WebCrawler executes the query.

After executing the query, WebCrawler responds by replacing the search page with a results page, shown in Figure 3.1, containing links to relevant Web pages. The results are ranked according to their relevance to the query, and each result is identified by title and optionally, a summary of the document. Presented with the results page, the searcher can simply click the title of the page in which he is interested. Searchers commonly use the Back button of their browser to return to a results page so they may explore several of the results.

3.3 WebCrawler's Implementation

WebCrawler's implementation is, from the perspective of a Web user, entirely on the Web; no client-side elements outside the browser are necessary. The service is composed of two fundamental parts: crawling, the process of finding documents and constructing the index; and serving, the process of receiving queries from searchers and using the index to determine the relevant results. This process is illustrated schematically in Figure 3.2.

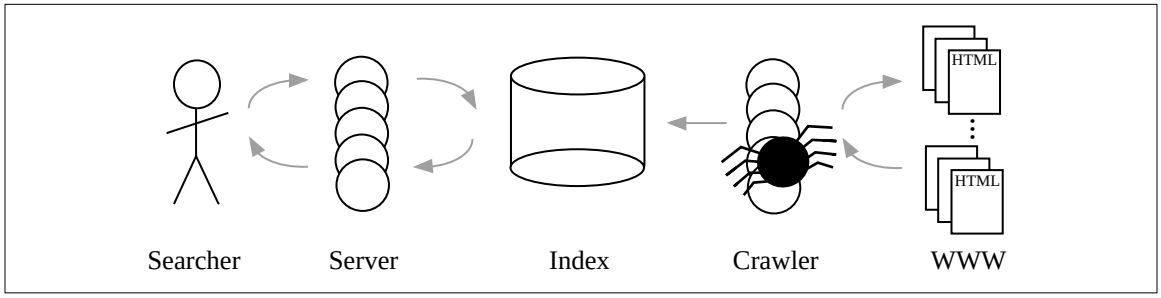

Figure 3.2: WebCrawler's overall architecture. The Crawler retrieves and processes documents from the Web, creating an index that the server uses to answer queries.

Though WebCrawler's implementation has evolved over time, these basic components have remained the same. Only their scope, detail, and relative timing have changed. Here I shall briefly describe an early implementation of WebCrawler, leaving the evolution for Chapters 4 to 6.

3.4 Crawling

Crawling is the means by which WebCrawler collects pages from the Web. The end result of crawling is a collection of Web pages at a central location. Given the continuous expansion of the Web, this crawled collection is guaranteed to be a subset of the Web and, indeed, it may be far smaller than the total size of the Web. By design, WebCrawler aims for a small, manageable collection that is representative of the entire Web.

Crawling proceeds in much the same fashion as a person would if he were trying to build a collection of Web pages manually: WebCrawler starts with a single URL, downloads that page, retrieves the links from that page to others, and repeats the process with each of those pages. Before long, WebCrawler discovers links to most of the pages on the Web, although it takes some time to actually visit each of those pages. In algorithmic terms, WebCrawler is performing a traversal of the Web graph using a modified breadth-first approach.

As pages are retrieved from the Web, WebCrawler extracts the links for further crawling and feeds the contents of the page to the indexer. The indexer takes the full-text of the page and incorporates it into the full-text index. Indexing is a fairly straightforward process, and can easily keep up with the crawling when given the appropriate resources. At first, indexing ran in step with crawling: each document was added to the index as it was retrieved from the Web. Later, indexing became a batch process that ran nightly.

Inherent in the crawling process is the strategy for constructing the collection: which documents are worth indexing? which are not? At first, the entire Web would seem to be worth indexing. However, serving queries against such a large index is expensive and does not necessarily result in the best results all the time. In Chapter 4, I shall describe WebCrawler's collection strategy and its rationale.

Because of the large size of the Web, approximately one billion pages at the end of 1999, implementing the crawler efficiently is difficult. Furthermore, the Web's authors make the task of crawling even more difficult by constantly adding, deleting, modifying, and moving documents. In isolation, each one of

these effects is small, but taken together they present a considerable problem. In Chapter 4, I shall describe the process of crawling, the requirements of the crawler, my solution, and some important lessons.

3.5 Serving

The process of handling queries from searchers and returning results in a timely way involves two main components: a high volume Hypertext Transfer Protocol (HTTP) service, and an index query process to perform the actual work of fulfilling the searcher's request.

Requests are received from the searcher's browser via HTTP. The searcher's HTTP connection is handled by a front-end Web server whose only job is formatting search results for presentation and returning those results in HTML format to the searcher's browser. To handle the actual query, the front-end server sends the search request to a special query server located on its local network. The query server runs the query against the full-text index, and returns results to the front-end for formatting. This two-tier architecture has proven effective at handling traffic as it increased three orders of magnitude. The service architecture and its predecessors are discussed in more detail in Chapter 6.

The index stored on the query server consists of two parts: a full-text index, and a metadata repository for storing page titles, URLs, and summaries. WebCrawler performs a search by first processing the query, looking up the component words in the full-text index, developing a list of results, ranking those results, and then joining the result list with the appropriate data from the metadata repository.

As I will describe in Chapter 5, WebCrawler uses an inverted full-text index with a combination of a vector-space and boolean ranking. Because Web searchers are generally naive, WebCrawler employs several enhancements to improve their queries in particular. These enhancements form WebCrawler's main contribution to IR, and are also described in Chapter 5.

3.6 Summary

Conceptually, WebCrawler is simple: it builds an index of the Web, takes queries from searchers via the Web, and returns relevant documents as results. However, as the following three chapters will illus-

trate, WebCrawler's task is not at all simple: crawling is difficult given the size, dynamic nature, and constraints of the Web; handling a large number queries against a big index is hard; and getting the right results for the searcher is also difficult.

Chapter 4

The Crawler

In this Chapter, I describe the crawler component of WebCrawler and its evolution from a simple page-gathering tool to a large database-based system.

The Crawler is the means by which WebCrawler collects pages from the Web. It operates by iteratively downloading a web page, processing it, and following the links in that page to other Web pages, perhaps on other servers. The end result of crawling is a collection of Web pages, HTML or plain text at a central location. The collection policy implemented in the crawler determines what pages are collected, and which of those pages are indexed. Although at first glance the process of crawling appears simple, many complications occur in practice.

These complications, as well as the changing Web environment, have caused WebCrawler's crawler to evolve. At first, crawling a subset of the documents on the Web using a modified breadth-first traversal was enough. More recently the scale, complexity, and dynamic nature of the Web have changed many of the requirements and made the crawling process more difficult.

Crawling is a step unique to Web indexing, because its document collection is distributed among many different servers. In a more traditional IR system, the documents to be indexed are available locally in a database or file system. However, since Web pages are distributed on millions of servers, we must first bring them to a central location before they can be indexed together in a single collection. Moreover, the

Web's authors frequently add, delete, modify, and move their documents, making the crawler's task even more challenging.

In this chapter, I shall describe the two fundamental crawlers in WebCrawler's history. For each, I shall describe the overall design, its implementation, its collection policy, and the problems I encountered.

4.1 The First Crawler

At first, WebCrawler's crawler was quite simple, primarily because of the scale and content of the Web. In late 1993 and early 1994, the Web was small, limited primarily to research and educational institutions. Indeed, WebCrawler's first index contained pages from only 6000 sites.

4.1.1 Details of the First Crawler

The first crawler, shown schematically in Figure 4.1, had four central pieces: a database in which to store state, a set of agents to retrieve pages from the Web, a dispatcher to coordinate the agents and database, and an indexer to update the full-text index with newly retrieved documents. The indexing component of

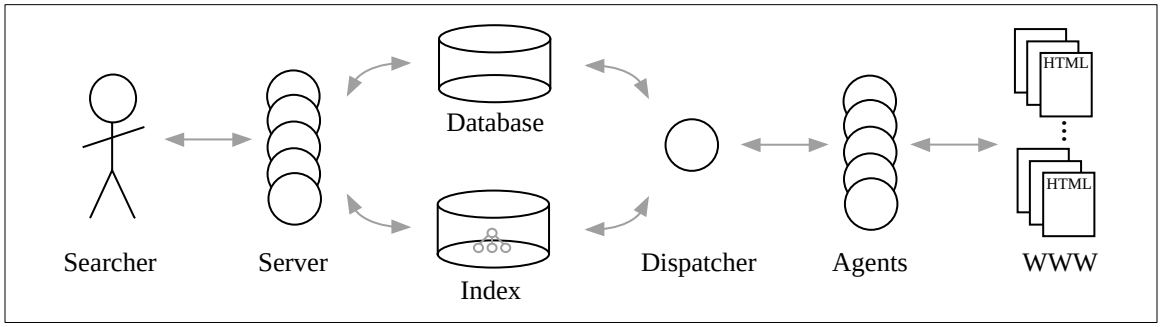

Figure 4.1: Architecture of the first crawler.

the crawler will be described in Chapter 5.

The database was the locus for all of the crawling decisions. Fundamentally, it contained a list of URLs to be crawled, and a list of those that had been crawled. It also stored metadata on those URLs: namely, the title, size, last modification time, and last visit time. The central process dispatcher, reading from the database, would select pages to be crawled and forward those URLs to a set of agents.

As WebCrawler visited pages, the links from the page were extracted and, if necessary, put in the database and the database was updated with the time the page had been visited. In algorithmic terms, the database served as a place to mark which nodes (URLs) had been visited and which had not, so that the nodes would not be re-visited during a traversal of the Web.

The database's schema was very simple: a server table enumerated the different web servers found, while the document table itemized each page. URLs were not stored in their entirety: the protocol (e.g. http, ftp, gopher, etc.) and hostname (or IP address) were stripped off and stored in the server table, leaving only the pathname of the URL to be stored in the document table. Each document, then, contained a pointer to the server on which it could be found. Finally, the database contained a link table that stored link relationships between documents. A link from document A to document B, for instance, resulted in an {A,B} pair being placed in the link table. While not strictly necessary for the process of crawling and indexing, the link table made a number of other analyses and features possible, such as the WebCrawler Top25: a list of the 25 most heavily linked sites on the Web.

In addition to saving space, this database structure allowed WebCrawler to operate its crawling process at the server level as well as the URL level. In the next section, I shall describe WebCrawler's crawling policy, and how WebCrawler used its database structure to achieve that policy.

WebCrawler's agents were very simple: their task was to retrieve a Web page, parse its HTML, and forward that text to the dispatcher process. The agents were implemented by using a thin wrapper on top of the CERN WWW library [W3C 2000] to retrieve the pages with HTTP and to parse the HTML on these pages into plain text. With agents running in separate processes, incorrect behavior by an agent would not cause the entire system to crash, and would ultimately allow the agents to be distributed to other machines.

As an agent completed its retrieval, it would send the full text and link data to the dispatcher, which would index the document synchronously, merging the newly indexed data into the central full-text index. The indexing process was very straightforward: given the title and full-text of the page from the agent, the indexer merely broke the text down into its component words and updated the posting lists for each of those words. I shall discuss the structure of the full-text index in more detail in Chapter 5.

4.1.2 Collection Policy

In a tightly coupled system such as that of the first crawler, the crawler is in charge of the collection policy: determining which URLs are collected and submitted for indexing, and which are not. From the very start, WebCrawler's explicit goal was to collect a subset of all the documents on the Web. There were several reasons for this policy. First, as more documents are added to a collection, the number of false positives increases and searchers will have a harder time distinguishing the relevant documents from the irrelevant ones. Second, WebCrawler was resource-constrained in its early implementation: the crawling, indexing, and serving components all ran on one or two machines. Creating a larger index simply was not feasible. As the Web has grown to over one billion pages, this constraint is even more true, despite the large number of resources that can be devoted to the task today. Last, by indexing in a hypertext domain, WebCrawler is taking advantage of the fact that it can bring a searcher close to his destination, and the searcher can then use the hypertext to navigate to his ultimate destination. Such navigation often increases the effectiveness of the search because the hypertext navigation can provide context for the information at the ultimate location.

Given that WebCrawler will not index the entire Web, the question is which documents should it index. With the goal of enabling searchers to get close to many different destinations in mind, WebCrawler used a modified breadth-first algorithm to target pages on as many different servers as possible. Since all non-trivial pages that it retrieved were included in the index, WebCrawler decided at crawl time which pages to crawl and, therefore, include.

WebCrawler's algorithm was in fact a simple breadth-first traversal. However, the breadth-first selection was done at the level of the server table, not the document table. As such, the actual node-to-node traversal did not represent a breadth-first one, but something harder to define. I chose the server-level traversal for several reasons. First, documents on a particular server are likely to be related with structured, easy-to-follow links among them. Therefore, getting a few representative documents from each server is a straightforward way to achieve WebCrawler's collection policy. Second, visiting servers in a breadth-first

fashion automatically creates delays between subsequent visits to a single server. As I shall discuss later, these delays are appreciated by server administrators. Finally, this breadth-first algorithm was easy to implement. Big mistakes, though they occurred, were less likely than with a complex algorithm.

The breadth-first traversal was implemented by keeping a cursor to the b-tree in which the server objects were stored, and simply advancing the cursor to traverse the list of servers. For each server, a candidate document was chosen and retrieved. The documents were usually chosen on first-in, first-out basis, though the root URL (*/,* or */index.html*) would be crawled preferentially if it was known to exist.

One important feature of this algorithm is that it works as WebCrawler explores the Web graph. If the graph were known ahead of time, more intelligent schemes could be used. For instance, for a particular index size, an algorithm could use the link table to determine the optimal set of pages that minimizes the connectivity to all pages known by WebCrawler.

4.1.3 Implementation Notes

This early version of WebCrawler's crawler was built on top of NEXTSTEP running on Intel-based PCs. It used NEXTSTEP's IndexingKit [NEXT 1992] for a database, in which serialized Objective-C objects were stored in each of the tables. The full-text index was also maintained using the IndexingKit's full-text functionality. Initially, WebCrawler ran on a single 486 with 800MB of disk and 128MB of memory.

4.1.4 Problems and Challenges

This first crawler worked well for six months. However, as the size of the Web increased, the implementation faced four major problems: fault tolerance, scale, politeness, and supervision. The most serious of these problems was fault tolerance. Although the system was basically reliable, the machine running the crawler would occasionally crash and corrupt the database. Although the IndexingKit was designed to tolerate these crashes, it did not. As such, it seemed prudent to engineer the system to anticipate a crash and be able to easily restart afterwards.

In the early days, simply re-crawling after a crash was an acceptable solution: the Web was small, and rebuilding the database from scratch could be accomplished quickly. However, as the Web grew and as server administrators became sensitive to repeated downloads of a particular page, keeping state in the crawler became more important — to the point where losing state from a crash could cause a delay of several days as the crawler ran from scratch.

With this fault tolerance in mind, the first modification to the Crawler was to stop indexing in real-time with the crawling, and to merely dump the raw HTML into the file system as it was retrieved from the Web. This allowed a degree of separation between the indexing and crawling processes, and prohibited failures in one from affecting the stability of the other. Nevertheless, subsequent experience with this crawler also indicated problems with fault tolerance, especially as the database grew large.

As the Web grew toward a million documents, scale also became an issue. Because both crawled and uncrawled documents were stored in the database, the database would eventually store a large fraction of the URLs on the Web. While the architecture of the crawler supported this idea, its implementation did not: the scalability of the IndexingKit simply did not support such a large database. This limitation surfaced in both the crawler database and the size of the full-text index.

The third problem with the early crawler was that, although it was relatively polite to individual servers, it did not obey the Standards for Robot Exclusion [Koster 2000]. These standards were first described by the Braustubl group, an impromptu gathering of search-engine experts at the Third International WWW Conference in Darmstadt. They outlined a process to keep Web robots like WebCrawler from downloading certain URLs on the Web, especially those that are generated by expensive back-end processes. To conform to these standards, a robot would simply download a file — `/robots.txt` — from each server it wished to crawl. This file contained a list of path fragments that all crawlers, or a particular crawler as identified by its UserAgent header, should refrain from downloading and, subsequently, indexing.

One interesting issue that occurred several times during the development of the crawler was one of supervision: when testing a new algorithm live on the Web, I made several mistakes and got the crawler into a tight loop downloading a single page repeatedly or a set of pages from a single site. With the bandwidth

available to the crawler typically being greater than that available to the victim site, this mistake would usually swamp the victim's available bandwidth and render the site unresponsive. The problem was exacerbated by the fact that many HTTP servers of the time did not easily cope with rapid-fire requests. Indeed, this kind of mistake was difficult to distinguish from a denial of service (DOS) attack on the site, something that administrators today watch for diligently. Thus, watching the crawler carefully during development and the early stages of deployment was crucial.

In the early stages of the Web, authors were not very understanding about robots downloading their pages because the utility of search engines was less clear than it is today. Many authors were worried about the load robots imposed on the Internet and on the sites they crawled. However, it is trivial to show that, all other things being equal, search engines actually reduce the load on the Net by creating shortcuts between Web pages for searchers. If every individual searcher had to get to his destination by following every intervening link, the collective load that all these searchers would impose on the Net would be far greater than simply having the search engine crawl it once.

4.2 The Second Crawler

I designed the second crawler to circumvent some of the problems encountered with the first crawler with particular regard to the issues of fault tolerance, scale, and flexibility. Conceptually, the architecture of this new system was close to the old one, but with a few important differences.

This generation crawler still had a database at its core, though this time the database was implemented using a commercial, full-scale database: Oracle. Surrounding the database were several key processes: the main dispatcher process retrieved pages to index from the database, and passed them out to a series of agents. These agents retrieved pages from the Web, and passed their results back to an updater process. The updater was responsible for queuing the pages for indexing and updating the database with the new metadata from the pages' retrieval. The architecture is illustrated in Figure 4.2.

One key aspect of scalability in the new crawler was that the different parts could be distributed on different machines. The database had its own high-performance machine. The dispatcher and agents ran on

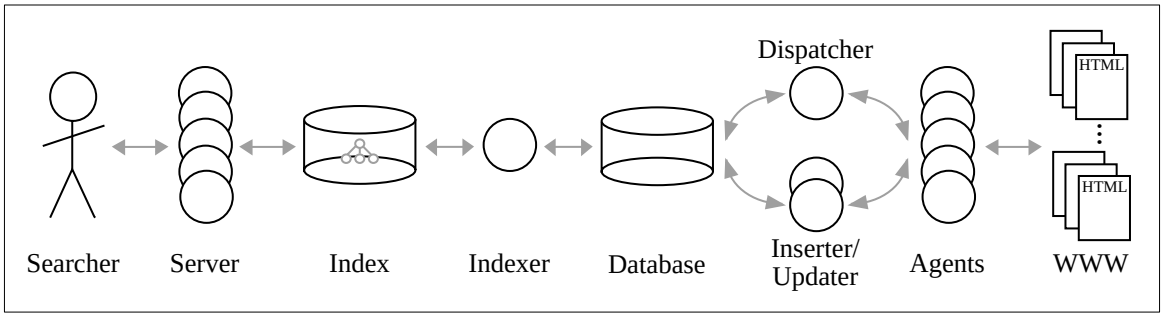

Figure 4.2: Architecture of the second crawler. The database is at the heart of the operation.

one computer, drawing their data over an Oracle client connection to the database. The agents eventually ran on multiple machines, connecting to the dispatcher using NEXTSTEP's distributed object mechanism [NEXT 1992]. The updater also ran on its own machine, receiving data from the agents, updating the database, and placing the full text on a file system shared with the indexer.

Using a commercial database at the core helped ensure that the database would not become corrupted upon a crash of the database server. Careful backups completed the fault tolerance equation. The database schema for this new crawler was nearly identical to that of the original crawler: a server table contained data on the different servers, while the document table contained information on each URL. Finally, a link table stored data on the linking relationships among pages. In practice, the link table was by far the largest table, growing to some 16 million rows at one point.

4.2.1 The Collection Policy

Once again, the goal of WebCrawler was to have a relatively small index with high-quality documents. Because of the distribution of the database and client processes, running a breadth-first traversal in the style of the first crawler was problematic. In fact, as the size of the database grew, WebCrawler began to change from a breadth-first crawl to a more selective approach. Given that WebCrawler could store several million URLs in the database and given that the dispatcher process was remote from the database, stepping a cursor through the server table row-by-row was not feasible. We began to converge on a new strategy for the collection: we would crawl many documents, obtaining metadata and link information for them, and use that data to select the best documents for the collection.

If a human were given the task of choosing a set of pages to put in an index, he would probably look at the breadth of pages available, and choose among those to populate the index. This strategy is exactly the way the new collection policy worked: WebCrawler collected a large number of documents, then selected a subset for the index.

To select documents for the index, WebCrawler chose according to a number of criteria, often manually selected. First, we wanted to maintain some degree of breadth, so we identified the shortest URLs from each server for inclusion on the theory that the URLs with the fewest path components were most likely to be the more general pages on the server. We also identified the more popular pages from the entire Web for inclusion, reasoning that more popular pages were likely to be of higher quality. To determine popularity, we used data from the link table. A page P was more popular than a page Q if it had more links to it from other pages not on its own server. The link table, coupled with the power of the relational database, made these kinds of determinations easy. In fact, one of the great successes of this crawler was the ease with which we could experiment with different selection criteria for the collection.

4.2.2 Implementation Notes

This crawler was implemented on several different systems. The database was an Oracle relational database and ran on an eight processor SGI Challenge XL with approximately 100GB of striped disk storage. The dispatcher, agent, and updater processes ran on a variety of NEXTSTEP PCs. The dispatcher and agents used NEXTSTEP's Distributed Objects [NEXT 1992], and the dispatcher and updater used the DatabaseKit [NEXT 1994].

4.2.3 Problems and Lessons

We learned many lessons with this new system. Architecturally, the second crawler was robust enough to crawl more than a million documents a day, a significant increase in throughput from the previous crawler. Most importantly, the database became a useful foundation for other parts of the service: it pro-

vided metadata to the information retrieval system far beyond what was available from the first crawler, and it also allowed us to build useful tools for guiding the crawling process.

While both crawlers provided ample metadata on the documents they crawled, the first crawler did not put this information in a very general form, meaning that applications of the data had to be anticipated before the data was captured. With the information in a relational database, applications for the data could be decided later and applied when necessary. An example of such an application is the external ranking discussed in Chapter 5. It uses link count information, filtered in a special way, to decide which documents to promote in the search results. Such data was easily obtainable from the relational database, and was not from the custom structures of the first crawler.

Another important application of the database was in directing the crawling process itself. Requests from site administrators could easily be entered into the crawling queue, and content designed to spoof the relevance ranking process could just as easily be removed.

However, from an implementation point of view, the most fundamental lesson was that commercial software packages, particularly those which provide a high level of abstraction, are difficult to make work successfully in a high-performing non-stop system. Although such difficulty does not reflect on the architecture, one could make improvements to the architecture that would allow for less-reliable components.

Parts of the system were unreliable. For instance, using NEXTSTEP's distributed objects provided a meaningful implementation shortcut, because objects and references could be transparently sent across process and machine boundaries. However, in a situation where this communication occurred constantly, 24 hours a day at full load, the system would hang frequently. Switching to a TCP-based protocol fixed this problem.

Though using NEXTSTEP's DatabaseKit [NEXT 1994] provided a high level of abstraction, created scale problems because of its high overhead. In the DatabaseKit, rows from the remote database are manifested in the application as instances of a particular Objective-C class. This abstraction is powerful, because it allows the programmer to write his code in an object-oriented fashion and frees him from having to write code to extract data from the database to instantiate objects. However, the overhead associated with

instantiating an object for each row retrieved from the database and with managing the associated memory allocation turned out to be quite a problem. On the Updater machine, for instance, fully 50% of the application process's CPU would be involved in these steps.

We noticed that certain operations in Oracle were not nearly as fast as they should be. In some cases, doing the right trick with the query optimizer would fix our problem, but in many cases the built-in overhead was simply too high. Locking conflicts inside Oracle made it infeasible to have more than one dispatcher or updater. Oracle uses row-level locking for most operations, yet still acquires more coarsely grained locks occasionally. These locks made running more than one simultaneous updater infeasible. Furthermore, the combination of the DatabaseKit with abstraction provided by Oracle meant that finding these conflicts was difficult and required examination of the SQL generated by the DatabaseKit.

4.2.4 Coping with the Changing Web

Aside from many of the local implementation problems, the Web had begun to change. The rapidly expanding Web, both in size and usage, meant that mirrors of popular sites were becoming common. Indexing a page and one or more of its mirrored copies became a likely occurrence. Such behavior, though, is not desirable because a single query will result in multiple identical pages that are served from different locations. To solve this problem, WebCrawler computed a checksum (a 32-bit CRC) for each page, and would not index a page if it found another page with the same checksum and size already indexed.

However, the checksum approach led to an interesting problem: the authors of pages that were selected by this method often complained that their mirror URLs were entered in the index at the expense of the correct URL. For instance, because of the vagaries of the crawl, WebCrawler might choose the URL *http://host.lcs.mit.edu/mirrors/company/a.html* over the more desirable *http://host.company.com/a.html*. To correct this problem, WebCrawler employs a simple, yet remarkably effective, heuristic. Whenever a page is found to have a duplicate CRC in the database, WebCrawler computes a score for the URLs of each duplicate page, and chooses the page with the highest score. If the page with the highest score is already in the

index, no action is taken. However, if the new page has a better score, the old page is deleted from the database and the new one is queued for indexing. The details of the scoring algorithm are found in Figure 4.3.

```
Each document is awarded an unbounded score that is the sum of:

15 if the document has ever been manually submitted for indexing
 5 for each time the document has been submitted

 7 if any other document in the database links to the document
 3 for each such inbound link

 7 if the URL ends in /, or
 5 if it ends in .html
 1 for each byte by which the path is shorter than the maximum (255)

20 if the hostname is a hostname and not an IP address
 5 if the hostname starts with www.
 5 if the URL's scheme (protocol) is HTTP, or
 5 if the URL's scheme is HTTPS
 5 if the URL is on the default port for HTTP (80)
 1 for each byte by which the name is shorter than the maximum (255)
```

Figure 4.3: Details of the URL scoring heuristic. This heuristic did not change during two years of use.

Another problem occurred because of the increasing importance of search engines in driving traffic to Web sites. Because many users rely on search engines to direct their Web experience, unscrupulous authors on the Web began to try to increase their ranking in search engines' results, at whatever the price. Often, this meant simply trying to improve their ranking. Over time, it also meant having one's page appear highly in the search results for any topic, no matter how unrelated that topic was to the page. This tactic quickly earned the name *spam*, after the electronic mail technique of sending junk mail to many recipients.

WebCrawler employed one overriding principle in choosing techniques to combat spam: index only what the end-user of the Web page could see in their browser. This led to the elimination of hidden text in meta tags, for instance, and of text in fonts that were either too small to see or in the same color as the background color.

One variant of spam is very difficult to detect: a spammer will serve an altogether different page to WebCrawler than they would serve to a user. The spammer can do this by altering their server to notice

requests from the WebCrawler crawler, identified by a specific HTTP User-Agent header or a known IP address range, and serve different content. Because WebCrawler crawls from a specific set of IP addresses, and accurately sends the HTTP User-Agent header, this kind of spoofing will always be successful. One way to combat it is to crawl from widely different IP addresses and mimic a browser by sending a User-Agent header identifying the crawler as, for instance, the Netscape browser. Another way is to mask the crawler's traffic in a stream of otherwise legitimate requests by crawling from behind a large cache, such as those found at many large ISPs.

4.3 Summary

In this chapter, I have described the problem of crawling and the evolution of WebCrawler's two crawlers. The ultimate size of the Web, and the impossibility of getting a perfect snapshot, led to the development of WebCrawler's ability to choose a useful subset of the Web to index. This choice is influenced by several factors, including metadata from the Web and input from administrators. Ultimately, the task is made possible by the most recent database-centered architecture.

Many of the problems described in the previous section are avoidable with the right design and right components. Indeed, several of them were overcome by falling back on more simple technology. Several improvements, however, could still be made.

While the architecture of the second crawler is sound and relatively scalable, it did not take into account the unreliability and poor performance of its constituent parts. Designing for distribution and failure from the start would help: each component would not need to perform at peak capacity all the time, and could occasionally fail without bringing the entire system with it. The main difficulty in realizing such a distributed crawler is making sure that URLs are not crawled more than once and that multiple URLs identifying the same document are recognized as such. Such distribution also makes metadata queries more expensive.

Another improvement would be to maintain the link table in a data structure optimized for its unique structure. Using a general purpose database for this task is fine during the crawling stage, but

becomes a bottleneck when more complex indexing and quality assurance algorithms need to be run against the data. A good example of such an optimized structure is the Connectivity Server [Bharat 1998].

Chapter 5

The Information Retrieval System

In this Chapter, I shall describe WebCrawler's information retrieval system, the philosophy behind its features, its evolution from a simple system to a more complex, scalable one, and its contributions.

An *information retrieval (IR) system* is a software system that helps its users find information. In the context of WebCrawler, the IR system contains information about a set of Web pages, called the *collection*. WebCrawler takes text queries expressed by searchers and retrieves lists of pages matching the search criteria. Like most information retrieval systems, WebCrawler's IR system employs several databases and some processing logic to answer queries. The *query model* of an information retrieval system is the model underlying the searcher's interaction with the system. It specifies the content of the query and the way that query is related to the collection to produce results. The concept of *relevance* is at the heart of most text-based information retrieval systems (see Chapter 2). Relevance is a measure of the similarity of a document and a query and is used to order WebCrawler's search results. At the core of the WebCrawler IR system is a *full-text index*. A full-text index is an inverted structure that maps words to lists of documents containing those words, and is used to resolve queries against large document collections [Salton 1989]. In a traditional database, the index acts merely as an accelerator with the true results coming from tables of actual data. In a full-text index, however, the index itself contains the information necessary to determine and produce the resulting set of documents.

This chapter describes WebCrawler's two successive information retrieval systems, their query models, their implementations, and their shortcomings. For each, I shall describe how the changing context of the Web and WebCrawler's users contributed to the evolution of WebCrawler's design. WebCrawler's chief contributions were two new approaches — external weighting and shortcuts — for enhancing the traditional information retrieval system in the specific context of a widely used search engine.

5.1 The First Information Retrieval System

WebCrawler's first information retrieval system was based on Salton's vector-space retrieval model [Salton 1975]. I shall first describe the query model and semantics of the system, and then describe the details of its implementation and some of the problems with the query model.

This first system supported collection sizes from 40,000 documents up to nearly 300,000 documents. It was implemented using the B-Tree support in NEXTSTEP's IndexingKit. The software ran comfortably on a PC: first, a 486 with 64MB of RAM; and later, a series of Pentium PCs each with 128MB of RAM.

5.1.1 Query Model

The first system used a simple vector-space retrieval model, as described in Chapter 2. In the vector-space model, the queries and documents represent vectors in a highly dimensional word space (one dimension for each word in the dictionary.) The component of the vector in a particular dimension is the significance of the word to the document. For example, if a particular word is very significant to a document, the component of the vector along that word's axis would be strong. In this vector space, then, the task of querying becomes that of determining what document vectors are most similar to the query vector. Practically speaking, this task amounts to comparing the query vector, component by component, to all the document vectors that have a word in common with the query vector. WebCrawler determined a similarity number for each of these comparisons that formed the basis of the relevance score returned to the user.

The math behind this computation was fairly simple: the score was simply the dot product of the query vector with any document vector that had a term in common with the query vector. Because the query vector contained many zero components (for all the words not in the query), this computation was fast: I needed only look at the set of components in each document's vector that were in common with the query vector. If there were *w* words in the query, then the score for each document was

$$relevance(doc, query) = MIN\left(\sum_{i=1}^{w} tf(i),\ maxweight\right)$$

where *tf(i)* was the frequency of the word in the particular document, computed as the number of occurrences of a word divided by the total number of words in the document. The rationale for using this weighting was that the frequency of a term in a document was a good measure of the importance of that term to the document.

While the computation of relevance determined the relative importance of each document to the query, a domain weighting calculation determined the relative importance of each word in the query. The rationale for this calculation was that some words in the query were more likely to identify the right documents than other words. For instance, in the query *X-ray crystallography on the World-Wide Web* the words most likely to lead to relevant documents are *X-ray crystallography.* This is true because the words *World-Wide Web* occur frequently in all documents, and would not help in narrowing the search to a precise set. Thus, WebCrawler used a domain weighting calculation to more heavily weight those words that were most likely to narrow the search to a particular set of documents. An important feature of the domain weighting calculation in the vector-space model is that it does not exclude documents from the result list, but instead changes the ranking so that the more relevant documents will be more likely to occur first.

The key to making this calculation is determining the contribution of an individual word. To do that, WebCrawler simply relied on its own full-text index: each word was associated with a list of documents containing that word, and the number of documents relative to the size of the collection was an indicator of

how likely that word is to discriminate among documents. Specifically, WebCrawler took the log of the number of documents that contain each query word and multiplied a document's score by the results.

$$relevance(doc, query) = MIN\left(\sum_{i=1}^{w} tf(i) \cdot \left(1 - \frac{\log(\text{docs containing i})}{\log(\text{total \# of docs})}\right),\ maxweight \right)$$

WebCrawler's first query model was a simple vector-space model. It did not support boolean combinations of words, nor did it support phrase searching or other measures of adjacency. However, with the small size of the early WebCrawler indexes, these problems did not introduce a lack of precision.

WebCrawler's query model also included several minor features that made a difference for searchers. First, WebCrawler included a stop list to filter common words from the queries and index. The stop list is used principally to reduce the size of the index by removing cases where nearly every document in the collection is listed for a particular word. Because the words in the stop list are so common, removing them from the queries has a negligible impact on recall in the vector-space model. WebCrawler performs limited stemming on words, reducing only plurals to their root form using a restricted Porter stemmer [Porter 1980]. Using the full breadth of the Porter stemming algorithm would have led to too many false positives given the breadth of languages and strange words on the Web.

5.1.2 User Interface

WebCrawler reported the result of a query by replacing the search page with a results page, shown in Figure 5.1, containing links to the relevant Web pages. The results were ranked according to their relevance to the query. Each result was identified and displayed by title. The underlying relevance numbers (described in the previous section) were displayed in front of the document titles, and were normalized so that the highest relevance was 100%.

My key priority in the design was to make it easy for the searcher to see the breadth of his results. With that in mind, WebCrawler presented 25 results per page, a number that enabled most of the search results to fit in the visible portion of the searcher's browser. The page containing the next 25 results was accessible by a link at the bottom of the current 25. Because of the stateless nature of HTTP requests and the

Figure 5.1: WebCrawler search results circa 1995.

difficulty of caching the results for many queries on the server, requests for subsequent pages of a query's result were processed as new search requests.

5.1.3 Implementation Details

WebCrawler's first IR system had three pieces: a query processing module, an inverted full-text index, and a metadata store. As illustrated in Figure 5.2, the query processing module parses the searcher's

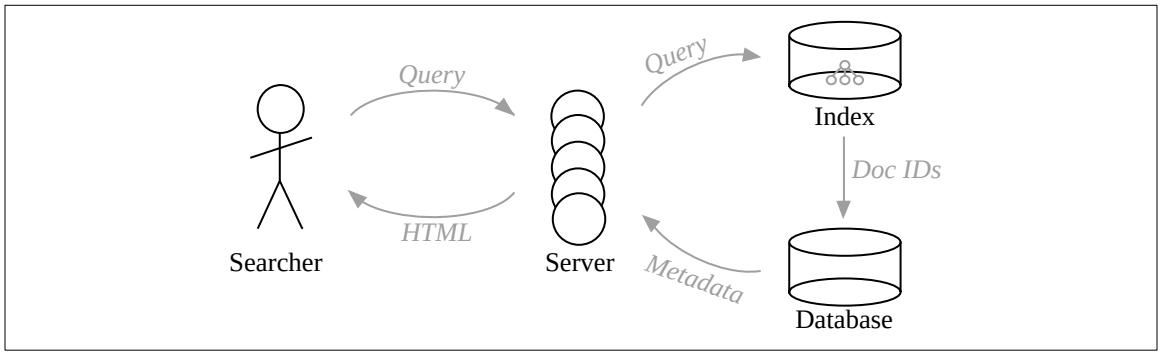

Figure 5.2: Query processing in the first IR system.

query, looks up the words in the inverted index, forms the result list, looks up the metadata for each result, and builds the HTML for the result page.

The query processing module used a series of data structures and algorithms to generate results for a given query. First, this module put the query in a canonical form (all lower case, no punctuation), and parsed each space-separated word in the query. If necessary, each word was converted to its singular form using a modified Porter stemming algorithm [Porter 1980], and all words were filtered through the stop list to obtain the final list of words. Finally, the query processor looked up each word in the dictionary, and ordered the list of words for optimal query execution. This ordering was applied only in conjunctive queries (that is, *dolphins AND whales AND porpoises*) by processing the least common words first.

Once the list of words was determined, the query processor retrieved the posting lists for each word and used them to compute the final results list. Finally, the metadata for each result was retrieved and used to generate the final HTML for the searcher.

The inverted index was the underlying data structure that the query processor used to compute the relevance for each document. Conceptually, the index mapped a particular word to a list of documents that contained that word and to a list of weights that indicated the relative importance of the word to each document. The documents were identified by a numeric identifier (ID) to save space in the index. Practically, this mapping was nothing more than a B-Tree, keyed on word, that contained a packed list of document-weight pairs for each of the words. Each document-weight pair is called a *posting*, and the list of postings is called a *posting list.* WebCrawler's postings were 32 bits each, with 20 bits devoted to the document ID, and 12 bits to the weight.

Once the list of result documents was computed, the query processor had to map the document IDs to information more meaningful to the user. This mapping was accomplished by using the same document table that the Crawler used to maintain state about each document (see "Details of the First Crawler" on page 31). WebCrawler simply took the ID, looked it up in the document table, retrieved the URL and title for that document, and used those to generate the results page.

When the first Crawler changed, splitting up the crawling and indexing steps, I also changed the metadata lookup process to use a dedicated B-Tree. With this change, the Crawler produced a special B-Tree with the inverted index whose sole purpose was to map from a document ID to a URL and title (i.e. only the information that needed to be presented to the user). This step simplified the lookup, making query processing faster.

5.1.4 Problems and Lessons

Early experience with the first model suggested that searchers could easily find documents in the collection and could navigate the user interface without confusion. However, as the size of the index and the search volume grew, several problems became evident.

The first problem was introduced by a combination of the increase in the size of the index and in the number of searchers using WebCrawler. As both the size of the user base increased and as a larger number of naive users came to use WebCrawler, the queries became less sophisticated. This trend, coupled with a larger number of documents in the index — and an increase in the breadth of the subjects covered by those documents — meant that WebCrawler needed a more sophisticated query model. The single most common request from searchers was for a way to restrict documents from the results that did not contain all the query terms. Another frequent request was to support phrase searching, so that only documents containing a specific phrase could be found.

The second problem with the query model became evident as WebCrawler indexed documents with an increasing diversity of terms. This change was due partly to the increasing number of documents, but primarily to the larger number of topics that those documents covered. For instance, WebCrawler increasingly found documents not just on academic topics, but also on many commercial topics and in many languages. Each new area introduced new terms and acronyms that swelled WebCrawler's dictionary. To reduce the conflicts among these terms, one of the early modifications to the query model was to eliminate plural stemming. More aggressive stemming was already turned off because of earlier problems, and eliminating plural stemming increased precision for some queries. This move did hurt recall to some degree. However,

because the fuzzy nature of the vector model made recall so good to start with, the decrease was not a significant problem for searchers.

The third problem was in the user interface of the results page. Though many searchers preferred WebCrawler's title-only results format, many of them requested that WebCrawler show a summary of each document along with the title. Such presentation allows the searcher to quickly evaluate a document's usefulness without having to follow the link to the document. However, it also increases the amount of screen space devoted to each result and, therefore, makes scanning a long list of results more difficult.

Fourth, as the index grew, I found that I frequently had to change the number of bits allocated to the document ID in the postings. Although these changes were not a big problem, they did make the indexes bigger. A more efficient posting format would have been a big improvement, though it never became a severe enough problem to warrant change.

Fifth, as the volume of search requests to WebCrawler grew, making efficient use of WebCrawler's computing hardware became an important issue. Indeed, simply making the code more efficient could stave off the purchase of new server hardware, thus saving money. Sorting was the principal bottleneck in computing the results list. Early on, I replaced the call to the generic Unix quicksort routine, qsort, with a call to a more highly optimized version. While this optimization helped, its savings did not last long: a bigger index and higher search volume quickly erased any benefit.

To further optimize the results processing, I replaced the sort with a simple find-the-top-N algorithm. Running such an algorithm is fundamentally faster than sorting as not all results are ordered: only those results that will be presented to the user need to be determined. To generate the first results page, then, WebCrawler need only scan down the list of results postings, checking each result against the list of the 25 most relevant found so far. In practice, the running time of this sort is linear in the number of postings with a constant substantially under 2. In the worst case, the constant is the maximum number of comparisons necessary to find and insert each value into a list of 25 sorted results, or $\log_2(25)+1 + 25$. Furthermore, the memory requirements for processing the query are reduced: one only needs to keep the top N results, and

need not store the entire result list for sorting. For a large index with many queries being executed in parallel, this memory savings can be significant.

However, as the user presses further and further into the results list, the benefit of running this algorithm instead of a conventional sort decreases. As a practical matter, WebCrawler always used this algorithm because the number of requests for results pages falls off very quickly after the first page. Figure 5.3 shows the breakdown of requests for each results page.

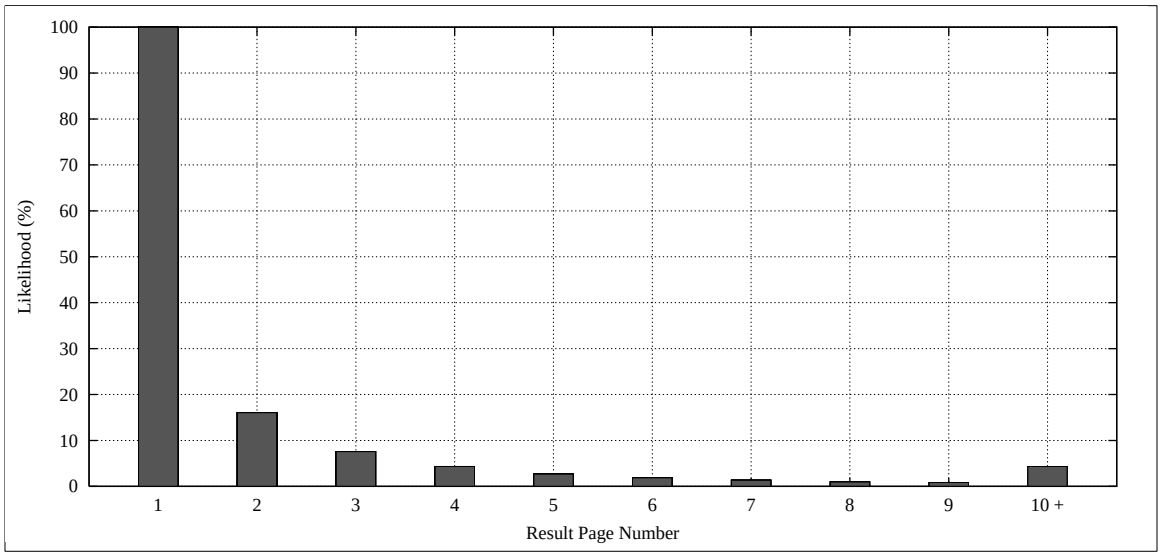

Figure 5.3: The likelihood that a searcher will view a particular results page. Every searcher sees the first page of results, and only 17% go on to view the next page. The tenth and higher pages are collectively likely to be viewed in only 5% of searches. This chart is based on data collected in October, 1995.

5.2 The Second Information Retrieval System

Because of the many query-model and scale problems with the first index, I moved WebCrawler to an entirely new IR system after two years. I chose to use a third-party package, Personal Library Software's Callable Personal Librarian (CPL) [PLS 1996], rather than building a new full-text system by hand. The need for an improved query model was driven primarily by my desire to increase the number of documents in the WebCrawler index, and the corresponding necessity of improving the query model. This coupling occurs because as the size of the collection increases, the precision of results naturally decreases for a given

set of queries. Thus, a query model that allows the searcher to specify his query more precisely is essential to maintaining precision.

I chose a third-party software package primarily because it offered most of the features WebCrawler needed, and because those advanced features were tedious to implement at scale. Coupled with this choice, however, came the necessity of moving off the inexpensive PC hardware platform and on to a more expensive minicomputer platform, the Silicon Graphics (SGI) Challenge XL.

5.2.1 Query Model

Like the first index, the core of CPL is ultimately based on Salton's vector model [Salton 1975]. However, CPL supported a large number of query operators on top of that basic model.

CPL supported basic boolean operators (AND, OR, NOT), as well as the critical phrase operator (ADJ, for adjacent) that was missing in the first index. As in later versions of the first index, WebCrawler employed no stemming at all. I increased the size of the stop list to accommodate the larger number of documents and to eliminate some of the very common words that were not helpful in making queries more precise. The term *www*, for instance, appeared in 80% of Web pages, making it essentially useless as a query term, and a useful addition to the stop list. The entries in this list were determined based both on their frequency in the collection and on their presence as discriminatory terms in queries. Some common terms, such as *server,* were present as discriminators in queries like *HTTP server,* and *window server,* and so they were kept from the stop list.

Perhaps the most important feature of CPL, however, was the way it handled queries with no advanced query operators. Around the time I introduced CPL, WebCrawler's user base was increasingly dominated by novice searchers who could not be relied on to issue well-formed boolean queries to increase the precision of their searches. At most, WebCrawler could depend on the searcher to enter a multi-word query that contained enough words to get reasonable precision. CPL helped with this problem by employing a clever ranking algorithm for multi-word queries with no operators.

The rationale for this algorithm is straightforward: for the query *Restaurants in San Francisco,* the searcher is clearly looking for restaurants in the City of San Francisco. In the first index, each of the words in the query would be treated independently, with no regard to the implicit phrase *(San Francisco)* or the implied desire for documents containing all the query words. CPL employed an algorithm that would first weight documents with all the terms more highly than documents with only some of the terms, and would then weight documents in which the query terms occurred as phrases more highly than documents where they did not. This weighting turned out to be an essential feature for WebCrawler, improving the precision of its top results on queries that would have been much less precise in the first index. The key lesson from this feature is simply that the search interface should adapt to the searchers and to their non-precise queries, and should not require the searchers to learn an arcane (to them) syntax for specifying searches.

5.2.2 User Interface

WebCrawler's user interface remained largely unchanged through the transition to the new index. CPL came with a built-in summarizer, so accommodating searcher requests for summaries on the results page was easy. Despite these requests, however, most users had become attached to the concise titles-only results format, so summaries became an option, selectable at the top of the results page. Searchers could also set a preference that would enable summaries to be returned by default.

Another addition to the user interface was a form of relevance feedback that allowed the searcher to find documents similar to one already present in the search results. The rationale for this feature was that users would often find a document that was close to, but not quite, what they were looking for. Using the feature, they could request more documents like the document of interest, without having to enter any query terms. Figure 5.4 shows a results page, with summaries and the relevance feedback links.

Figure 5.4: A modified results page with summaries and relevance feedback links. The second result is present because it uses gardening as an example.

5.2.3 Details of Second Index

The structure of the second index was very similar to that of the first, with CPL being used for the detailed query-processing and inverted-index components. Overall, query processing and metadata lookup remained outside of CPL.

Query processing in the second index consisted of performing initial cleanup of the query, and then passing the query off to CPL for its processing. CPL would then return results for the query, with each result expressed as a relevance-document ID pair. As in the first index, the query processing module mapped each result to its metadata: its title, URL, and, if necessary, its summary.

Although CPL had the ability to maintain all the metadata inside its own data structures, the overhead of that piece was significant enough that I elected to keep the metadata outside CPL. Where the first

index used B-Trees to store the metadata, the second index employed Gnu DBM (GDBM) hash tables outside the CPL database. CPL was responsible for mapping queries to document IDs, while the hash tables mapped document IDs to metadata. I used two separate metadata tables: one for titles and URLs, and one for summaries. The reason for this split was primarily to avoid bringing the large summary data into memory unless necessary, since it was not used by default.

For efficiency of lookup, I rewrote the GDBM read interface to memory map the GDBM files and walk the data structures in memory, rather than issue a read system call for each page needed from the file. This optimization was ideally suited to Apache's multi-process model and drastically reduced the overhead of looking up metadata.

5.2.4 Implementation

With the use of CPL came less control over the platform on which the query processing ran, as implementations of CPL were available only for a few select platforms. We chose three Silicon Graphics Challenge XL systems for the query processing. Two of the systems had sixteen processors each, and one had eight. The rationale for choosing this architecture was simple: we would bring as much of the index into memory as possible, and amortize the cost of a large bank of memory across as many processors as possible. The Challenge XL allowed us to put at least 32 processors in a single machine. This system had enough memory bandwidth to keep the processors busy, even in a memory intensive application like full-text search.

5.2.5 Problems and Lessons

WebCrawler's second index was a dramatic improvement over the first. The query model worked very well, and the size of the index allowed searchers to either find their page directly or get close to their destination. Nevertheless, several problems appeared during the development and deployment of the index.

One of the first problems I encountered was the problem of assigning document IDs. WebCrawler's crawler maintained a unique document identifier for each document in its database. However, for efficiency reasons, CPL was structured such that it had to assign its own document IDs for each document in

the index. The reason CPL could be more efficient with its own IDs is simple: WebCrawler document IDs identified documents in the Crawler database, many of which were not to be included in the full-text index. With the holes in the document ID space introduced by this strategy, the CPL index compaction techniques were much less efficient, making the index larger and less efficient at query time. Thus, I opted to allow CPL to create its own internal document IDs and to provide the query processing engine with a mapping to these database IDs used in the metadata tables. This strategy was reasonably efficient, because I needed to use the relatively slow mapping function only for each result returned to the searcher, typically only 25 per query.

With the phrase-searching support in CPL, searchers could issue queries for any phrase. However, occasionally a searcher would want to issue a query in which crucial words in the phrase were on the stop list, (for example: *'music by The The'* or *'Documentation on the X Window System'*), and the query would be transformed into a much less useful query when those words were eliminated. Running with no stop words was not an option, because the lack of stop words would slow down ordinary queries too much. Although techniques exist in the full-text literature for handling this problem, CPL did not implement them. Having support to handle these queries would have been useful.

Perhaps the most fundamental problem we encountered with CPL was performance. Because the SGI Challenge hardware supporting the query processing was expensive, using it wisely was important. The first optimization we made was to use external storage for the metadata, as mentioned in Section 5.2.3. Subsequently, however, it was evident that we would need to work on CPL directly. Since it was a third-party product, making these optimizations was up to the manufacturer.

My analysis of the performance of CPL identified four important areas for improvement. First, an instruction-level breakdown of the posting list copying code revealed many unnecessary calls to the memory copy routine *bcopy*. These could be eliminated to save CPU overhead in a critical inner loop. Second, the mere fact that CPL was copying postings internally was a serious problem itself. Typically, high-performance query processing involves walking the internal data structures and accumulating the results, without making intermediate copies of the postings. CPL's copying was causing it to use more CPU than necessary.

Third, CPL was using a read/write interface to its underlying data store, exacerbating the data-copying problem at the system level. A more efficient memory-mapped model would have helped dramatically. Not only would CPL have avoided copying the data from the buffer cache to CPL's own buffer, but the operating system could do a much more intelligent job of managing cached pages. Fourth, CPL was written with a process-based model in mind. The library was not multi-threaded, and could not be used — even in read-only mode — from multiple threads. The use of a single address space would have helped, both from a caching and context-switching perspective.

5.3 Beyond Full-Text

Because the CPL query model was providing such good results, albeit slowly, I began to look for other ways in which the search results could be improved. Two ideas emerged as significantly improving results: influencing ranking by external weighting, and shortcuts. At the time, these ideas were new to search engines and, indeed, to the field of information retrieval. Today, they are present in several different search engines in many different ways.

5.3.1 External Weighting and Link Counting

Simple queries, generally one or two words, were most prominent in WebCrawler's query stream; so solving the problem of how to return meaningful results for these queries was important. The full-text ranking of results from a one-word query depended largely on the presence of the word towards the beginning of the document (a CPL feature), and the length of the document. In a large collection like WebCrawler's, the number of otherwise undifferentiated results for such a query was large, forcing the searcher to look through many results.

With WebCrawler, I experimented with — but did not release commercially — an algorithm that uses both the full-text relevance score and an external score to compute a new ranking for a set of documents. The external score was a numeric property of each document and could be based on any attribute of the document: timeliness, popularity, quality, size, or even the number of images it contains. The theory

behind using such an external score is that there will often be a different way to rank documents — other than relevance — that depend on the query or the type of documents being returned. For instance, a search that returns primarily news articles may be appropriately ranked on recency instead of, or in addition to, the normal full-text ranking. This kind of influence becomes particularly important when searchers' queries are imprecise. With so many pages on the Web, another measure is possible: a measure of a document's quality. The rationale for this approach is simple: for common queries, with all other things being equal, one would rather have documents of higher quality ranked more highly. However, what constitutes a document of high quality is in the eye of the beholder. For example, a teenager may consider popular documents as quality documents, while a researcher may be interested only in documents that have been heavily reviewed. I will describe WebCrawler's quality rating in section 5.3.1.2.

5.3.1.1 Details of External Weighting

To perform external weighting, I used a table, organized by document ID, of external weights. The naive approach would have been to simply take these weights, in a fixed ratio with relevance, to form a final score for each document.

$$score(doc) = \frac{1}{2}relevance(doc, query) + \frac{1}{2}external(doc)$$

This approach works well, but does not normalize either the relevance or the external score to a scale, nor does it take into account the fact that the relevance score can be either better or worse, depending on the query. For example, a query like *travel* against a large collection will return a meaningful fraction of the documents in the collection, with relevance scores that don't differentiate them much. On the other hand, a more specific query such as *'surf conditions' and 'Santa Cruz'* is likely to produce fewer documents, with relevance scores that are much more meaningful. In the former case, one would rather use less of the relevance score, and more of the external weight; whereas in the latter case, the opposite is true: the relevance score tells a much better story.

WebCrawler's approach was to do exactly that: approximate the quality of the relevance score, and use either more or less of it in the final document score. I used two metrics to approximate the quality of the

relevance ranking: the total number of results, and the spread of relevance scores between the first and Nth result. When results are few, an external weight is less likely to be needed, as the searcher will not be combing through thousands of results. When results are many, however, an external weight could be more useful. However, just because a search returns a lot of results does not mean that it was a poor search: using CPL's relevance ranking is good enough to give the same results near the top for the queries *'surf conditions' and 'Santa Cruz'* and *surf conditions Santa Cruz,* even though the latter query will return many more results. So I also take into account the spread in relevance between the first and Nth term. In practice, I used a value of about 25% of the size of the result list for N. If the spread is small, then the top results are largely undifferentiated and an external weight will be more useful. If the spread is great, then relevance ranking is doing a good job of discriminating among results, and less external weighting is needed.

$$score(doc) = a \cdot relevance(doc, query) + (1 - a) \cdot external(doc)$$

$$a = MAX\left(\frac{2}{3}, 1 - \frac{1}{20}\log(\textit{all docs matching query})\right)$$

This algorithm, although expensive to compute at run time, gave wonderful results. Its expense came from the way CPL computed the results list incrementally: a traversal of even 25% of a large results list was expensive, and resulted in this operation being about twice as slow as computing the results using the normal method.

5.3.1.2 Link Counting

The external weight algorithm works with any kind of external weight. For WebCrawler, however, I chose to use weights that approximated the quality of a document. I took this approach because, as the size of the collection grew, it was clear that some documents were much better than others. The metric for quality I chose was the number of distinct sites that linked to a particular document. This metric uses the structure of the Web to capture what publishers around the Web view as the pages worth linking to. Presumably, these pages are those that are most informative. I specifically chose to use the number of sites rather than the number of pages because the number of pages could be heavily influenced by a lot of links emanating from, say, a friend's site.

One problem with this linking strategy is that new pages of high quality will not be linked to for some time after they appear on the Web. So the page may appear in the WebCrawler collection, yet have a low external weight because of its relative age. To solve this problem, I designed, but did not implement, a strategy to enhance a page's value based on its age in the WebCrawler database. Thus, pages that appeared more recently would have their link scores boosted to allow them to compete more favorably with pages that had been around for a long time.

Performing the link counting itself was easy: WebCrawler's Crawler maintained a table of link data. I simply ran a query on this table to collect the number of sites linking to each document, and then summarized that data in a GDBM hash table keyed on document ID. Even though the query was expensive, it only needed to be run once a week because the link count information was slow to change.

5.3.2 Shortcuts

Another new idea for improving results became known as *shortcuts*. The idea behind shortcuts was simple: in many cases, the searcher's query identifies a particular topic directly, and one can offer information with high relevance on that particular topic. For instance, if the query is *San Francisco*, one could offer the searcher geographical information and weather in San Francisco, as well as a list of Web pages containing that information. If a searcher enters an address into the search box, WebCrawler could offer a map of that address.

To be effective, the shortcuts content had to stand out on the results page. Rather than place the content at the top of the search results, where it would interfere with a searcher who preferred only Web results, we chose to place it on the right-hand side of the results. This presentation has the advantage of providing a consistent presentation for queries in which shortcuts are not present: the module simply disappears and does not disrupt the searchers attempt to find the beginning of the Web search results. Figure 5.5 shows the presentation of these shortcuts modules.

Shortcuts are content specific. That is, one has to know something about a particular kind of query to build a shortcut for that query. We chose to build shortcuts in a framework that allowed new modules to

Figure 5.5: A search results page with a shortcuts module presented on the right, circa 1999.

be added in a simple way: each shortcut was represented by a processing module that took the query as input and returned a score as output. The score was then used to determine if that shortuct module was displayed for a particular query. This framework allowed each module to perform its own query recognition in whatever way it wanted. The address module, for instance, looked for a query that matched the format of an address, while the musician module would compare the query to an internal list of known musicians. Shortcuts were implemented as a server module on the front-end that processed the results in parallel with the request to the back-end for the Web results.

The shortcuts strategy worked well, though it did not gain a large amount of user acceptance for two reasons: it was not widely implemented (we did only a small number of modules) and, as such, its placement on the results page did not command much attention from users. Furthermore, the shortcuts model was not especially scalable, since each module had to perform processing on every query. Even with strict per-

formance requirements for the modules, one could only involve so many modules on each query. A more general strategy is necessary. Recently, this fundamental idea has been successfully used to augment Excite search results [Excite 2000].

5.4 Summary

The expanding Web imposed constraints that caused us to constantly improve WebCrawler's information retrieval system. In the first system with its small collection and low traffic, a standard vector query model was sufficient. As traffic increased, so did pressure to improve the user interface and the query model, necessitating a simplification of user interface, a change to commercial IR software, and a larger index. These changes were largely successful at making WebCrawler one of the Web's top search engines.

Further growth in the size of the index, however, came with reductions in the precision of searcher's queries, particularly for naive users with naturally imprecise queries. To alleviate these problems, we introduced two important new features: external ranking with link counting, and shortcuts. External ranking was designed to improve relevance ranking on short queries without sacrificing relevance quality for more complex queries. Shortcuts were designed to surface material directly relevant to the query alongside the ordinary search results. Both methods were improvements in the user experience, though external ranking was not widely deployed.

Chapter 6

WebCrawler as a Service

So far in this dissertation, I have described WebCrawler's functional side, but have said little about its actual deployment as a service. Examining WebCrawler as a service means looking not just at the task of determining answers to queries, but also at the task of receiving the queries, transmitting the results, and serving the other content that supports WebCrawler's search functionality. These activities largely entail solving the problem of high-volume HTTP serving, where clients request both static content (HTML pages and GIF images that don't vary among requests), and dynamic content computed on the fly (search results). Doing an effective job of serving this content means looking not just at efficient query processing, but also at the Web servers and even the connections to clients.

At first, deploying WebCrawler as a service was relatively easy because of the small index size and low traffic volume. However, as WebCrawler's index has grown and as WebCrawler has acquired a larger base of users, the task of operating the service has become more difficult, leading to several interesting systems problems. In addition to simply keeping up with growth, two other important goals drove the process to improve WebCrawler's service: better service for searchers, and a decreased cost of running the service.

While decreased cost and keeping up with scale are self-explanatory goals, improving service to searchers needs further explanation. To improve service to searchers, I focused on improving response time and availability: in short, the service should always be fast, and it should always be up. As with the crawler, a common theme in this chapter is that the aggregation of unreliable parts creates a system that is difficult to

operate reliably. In the course of improving the service, I employed several different techniques: namely, distribution, load balancing, and fault tolerance.

In Chapter 4, I explained how the increasing size of the Web caused problems for the Crawler, and in Chapter 5 I explained how the combination of the Web size and the larger number of users made query processing a challenge. The following sections describe WebCrawler's three successive services, from the simple initial architecture to a more complex, scalable, and distributed service that has survived a two-order-of-magnitude increase in traffic. Indeed, this architecture remains the reference architecture for many of today's Web services. For each service, I will describe its architecture, its important load-balancing features, its handling of fault tolerance, and some of the problems with the service.

6.1 Introduction

Before describing WebCrawler's three successive services, I shall introduce four topics from distributed systems that are particularly important to understanding WebCrawler.

6.1.1 Architecture of the Service

Like that of the crawling and information retrieval system components, the architecture of the WebCrawler service has evolved over time, with two major revisions following the initial deployment. Making progress on each of the goals mentioned in this chapter's introduction motivated these revisions, though increasing scale proved to be essential.

Essential to understanding the WebCrawler service is a knowledge of how HTTP serving works. The environment of the Web is a simple request–response model where a user equipped with a browser issues HTTP requests to a server that returns content ultimately to be displayed in the user's browser.

In most cases, a Web page that is returned to the user is *static*. That is, the page's contents reside in a file in a server, which is read off disk and transmitted over the network. Many of WebCrawler's pages such as the home page, the graphics, and the help pages were static. However, as described in Chapter 5, all of WebCrawler's search results are *dynamic*: they are created by the query processor in response to a

searcher's query. This distinction is important, because dynamic pages typically take more resources to serve than do static ones.

6.1.2 Load Balancing

Load balancing is a common problem in distributed systems, particularly those that are distributed to handle scale. In WebCrawler's case, load balancing appears as a problem because the latest architectures employ a distributed model with more than one computer handling requests.

Three attributes of WebCrawler's traffic make it easy to load balance. First, HTTP requests, the units of distribution, are stateless: that is, WebCrawler does not maintain any client state on the server side that needs to be present the next time a particular client makes a request. Second, the requests to WebCrawler each put a similar load on the system, meaning that they can be considered identical for load-balancing purposes. Third, the volume of requests is high enough so that abrupt changes in the arrival rate are infrequent.

6.1.3 Fault Tolerance

An important issue that parallels load balancing in distributed systems is *fault tolerance*. Fault tolerance refers to the ability of a distributed system to function in the presence of the failure of one or more of its components. All of the load balancing architectures have implications for fault tolerance. In many mission-critical systems, fault tolerance is a serious concern: transactions simply must always work. On the Web, it would be difficult to complete a client's transaction if the server processing it crashed mid-transaction, so we are content with a definition of fault tolerance that allows for the mid-transaction failure of a server. In most cases, this problem is manifested to the searcher as a server failure, and he will simply reload the page. In WebCrawler, fault tolerance came to mean that the failure of one (or several) servers should not cause the service as a whole to fail. In particular, searchers should still be able to get answers to their queries, despite a failure of several components.

6.1.4 Workload and Capacity Planning

One of the operational aspects of WebCrawler was capacity planning. Because traffic was rising at such a fantastic rate, adding hardware to the service was almost a monthly occurrence. I used the degradation in response time, along with throughput estimates of the capability of each server, to determine when to add new hardware and how much extra capacity was required. Here, the mission of availability helped greatly: some hardware was needed in reserve so the service could continue to operate at peak load even with some failed components. This buffer zone was useful for absorbing unanticipated peaks in load, though it meant risking the fault tolerance of the system.

6.2 The First Service

The first service was simple: everything ran on one computer, no distribution was required. All HTTP requests to the service were handled by the Apache Web server [Apache 2000]. The server handled requests for static content by accessing the file system for the content. Search requests were initially handled using Apache's Common Gateway Interface (CGI), a method whereby the HTTP request is transferred from the Apache process to a newly created process, as shown in Figure 6.1. The CGI hand-off process

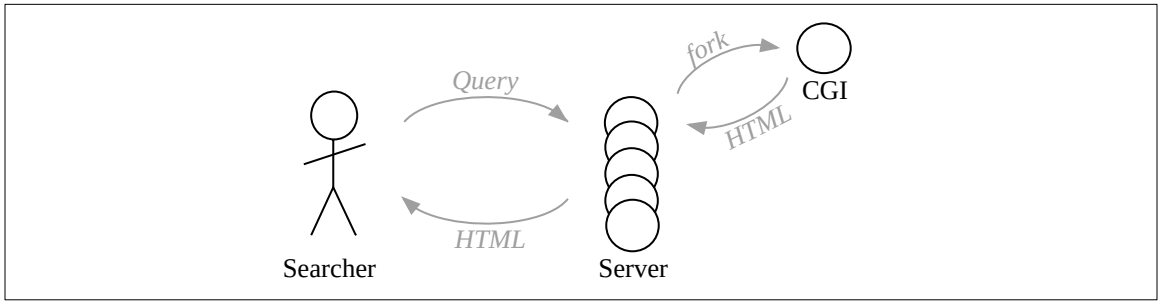

Figure 6.1: Architecture of the first service. A CGI program was forked from one process of the Apache HTTP server to handle each incoming query. Requests for static HTML were served directly by Apache.

works well, but because it involves creating a process, it is relatively slow and expensive.

Because of this CGI overhead, the first modification I made was to remove the CGI and compile the query functionality directly into Apache. This approach saved the overhead in creating the new process, and also in opening the database each time in that process. However, from an operational perspective, prob-

lems (bugs) in Apache could affect the query processing, and vice versa, making the server less reliable. The quick solution to this reliability problem was to have the Apache processes restart themselves every 300 requests or so.

6.2.1 Fault Tolerance

With only one machine for the entire service, availability was a key concern. When that one machine was down, the service was unavailable. At 300,000 queries per day, 10 minutes of downtime could mean disappointing over 5000 people during the peak time of day. Worse, the machine would generally fail in such a way that searcher requests would take a minute to time out, causing an unfriendly pause in the searcher's use of WebCrawler. The problem was exacerbated because the machine would typically fail in a way that required an operator to physically push the reset button (PCs cannot generally be reset from a remote console). To work around this problem, I built a simple circuit, described in Figure 6.2, that would

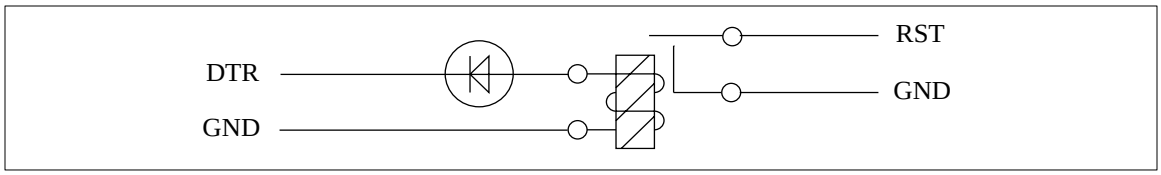

Figure 6.2: Rebooter circuit. The controlling computer's serial port is on the left, the server is on the right.

allow the reset button to be activated from another computer, facilitating remote resets of the server.

This hardware solution for availability was only so good, however. Even with a program automatically monitoring the server and resetting it, the delay introduced by the hung server would put the service out of commission for at least 10 minutes, making many searchers unhappy. Clearly, something more was needed to improve availability.

6.2.2 Problems with the First Service

In addition to the availability problem mentioned above, the first service suffered from three other problems. First was that the increasing scale of the Web caused the size of the full-text index to grow dramatically in size. There was a clear limit to the number of queries per second that one computer could han-

dle; moreover, that number was getting lower as the size of the collection increased. One obvious solution to the problem was simply to buy a faster computer. However, the rate of traffic growth was such that buying faster processors would not alone solve the problem. Figure 6.3 shows the growth of WebCrawler traffic from 1995 to 1997, growth that clearly outpaces the advances in processors predicted by Moore's Law.

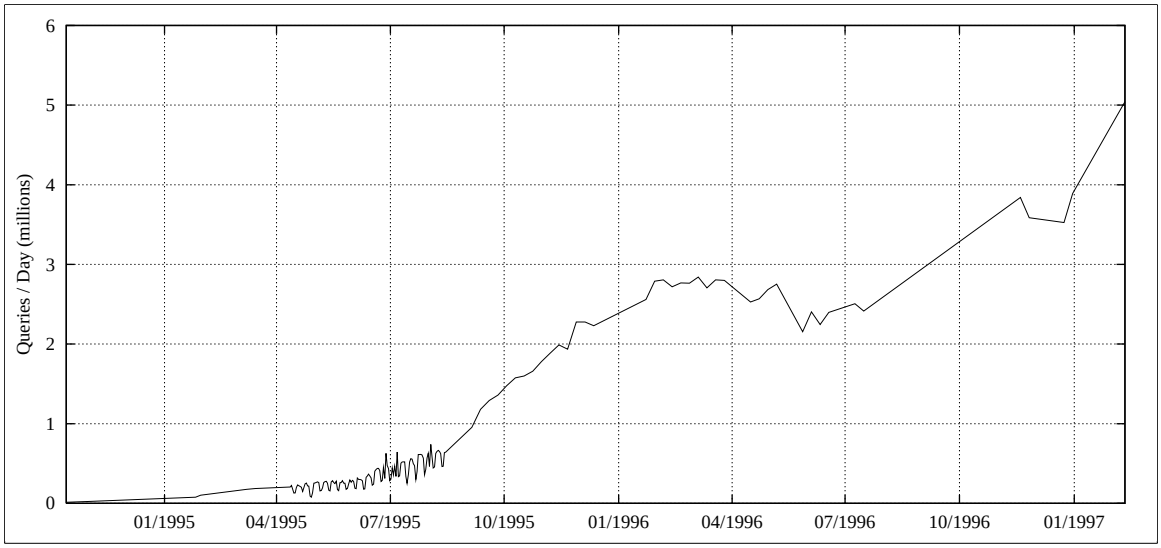

Figure 6.3: Growth of WebCrawler traffic, as queries per day, from 1994 to 1997. Where daily data was available (in 1995), daily variations appear because traffic patterns are different during each day of the week.

Second, and perhaps most surprising, was that early operating systems and drivers were not prepared for the network load introduced by running a Web service. Although the problem was most noticeable when the machine was heavily loaded, problems could occur any time. In WebCrawler's case, the NEXTSTEP Mach kernel needed several changes to withstand the repeated setup, use, and destruction of TCP connections. The most obvious problem was that the maximum number of TCP connections that could be pending on a socket at any one time was limited in the kernel to five. Indeed, this problem was not unique to NEXTSTEP. Nearly every Unix implementation had this problem in 1994. In all of those, this constant was a compile-time constant whose purpose was to limit the maximum resource usage by incoming connections. Unfortunately, this number did not scale with the capacity of the machine. I also encountered several driver and network buffering issues that the vendor helpfully fixed.

Third, the early strategy to couple crawling and indexing with serving made expanding the service difficult. This problem stemmed from two aspects of the architecture: the tight coupling of components meant that failures in one component caused a disruption for the others, and the load imposed by one would affect the response time of the others. Separating the components, as described in Chapter 5, helped solve these problems by creating an environment dedicated to serving queries. This action also set the stage for replicating the serving environment.

6.3 The Second Service

To solve the problems with the first service, I upgraded the server to a larger and faster machine, replicated the entire service on several equivalent machines, and moved the crawling and indexing to a separate set of servers. The new Web server was a considerably more powerful 133Mhz Pentium PC, with 128MB of RAM. Initially, I had three such servers. Over time, I increased that number to 10. Otherwise, the service remained the same as before: each machine contained a complete copy of the service and could function in isolation, depending only on updates from the crawler and indexer. Figure 6.4 shows the replicated service.

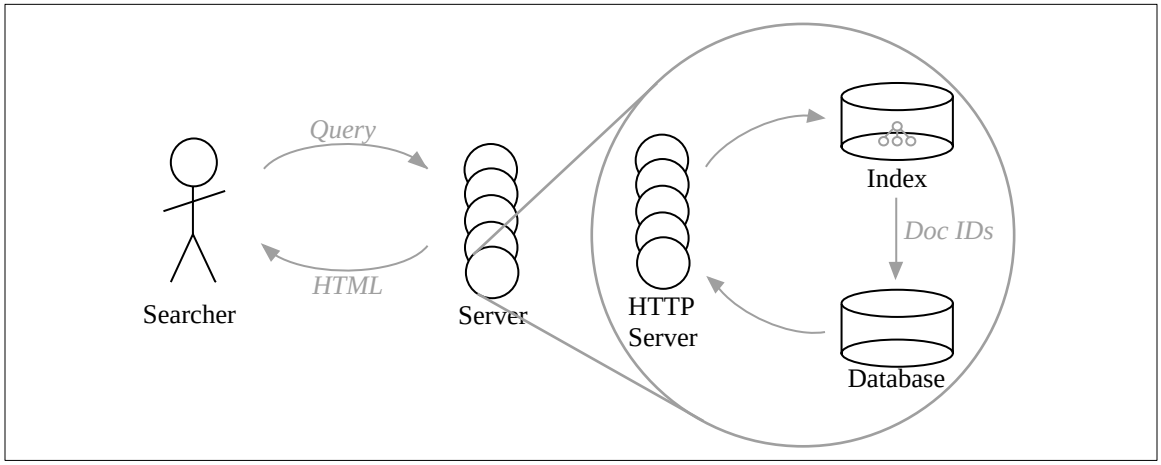

Figure 6.4: The replicated service. Each server had a copy of the index and could respond to any query.

6.3.1 Load Balancing in the Second Service

Replicating the service on several machines was fine, but one key question remained: how would the requests from users be distributed across the machines?

One option for distributing the requests was to create well-known *mirrors* of the site, and let users choose the least loaded site. This technique was widely used in the early days of the Web, particularly for software download sites. However, the problem with this approach is that it is visible to the users of the site: they must select the closest, least loaded server by typing in the URL of a specific mirror. How are they to know what mirrors are available? Considering this option led me to the important realization that WebCrawler should continue to be available on a single URL, and that single URL should be used for every request. Fundamentally, the searchers should not bear the burden of making a load balancing solution work.

DNS load balancing was one solution to this problem — and the first one I tried. DNS load balancing works by infiltrating the hostname to IP address mapping and returning different IP addresses for one name to different clients. Thus, one client will get one answer, and a second client will get a different answer. The client, or the client's name server, will cache that response for a time period called the time-to-live (TTL), specific to that name. In WebCrawler's case, the hostname *www.webcrawler.com* (and *webcrawler.com* itself) mapped to a list of 10 IP addresses, each corresponding to one of the 10 servers. Each client would get the same list of 10 IP addresses, ordered differently. Since the clients generally used the first IP address in the list, and since many clients were making requests, load was effectively balanced among the servers.

This DNS load balancing technique is commonly called DNS Round Robin (DNS RR). Most modern nameservers include the ability to do DNS RR; however, this technique was not widely available in WebCrawler's early days. WebCrawler's implementation of DNS balancing used a policy of random ordering: on each request, it would select a random IP address in the list, and exchange it with the first in the list. A later implementation used straight DNS RR, but we reverted to a variant of the first algorithm because a failing server would double the load on the server immediately behind it in the list.

DNS load balancing works well in practice for WebCrawler because it has so many clients, and each client makes only a few requests before moving on. The DNS technique also works well on servers that are geographically dispersed. We also used the DNS technique to take advantage of server location and route requests to the closer servers: within the America Online (AOL) network, clients were sent to the set of servers that were closest to them on the network.

6.3.2 Fault Tolerance in the Second Service

Fault tolerance in the second service was better than the first service, but still problematic. The machines would still fail occasionally; the rebooter circuits, coupled with automatic monitors, would restart the failed machines without human intervention. DNS provided the next step toward fault tolerance: inoperative servers could simply be removed from the list of IP addresses returned by the DNS system, and new clients would not see the failed server. Furthermore, because the DNS system handed out several addresses for WebCrawler, a browser that had cached the IP address of a failed server could automatically retry with the next address in the list.

6.3.3 Problems with the Second Service

The second service shared many problems with the first, because the underlying implementation of the service on each node was the same. The new problems introduced in the second service were a direct result of the DNS load balancing.

First, because the DNS system allows clients (and intermediate servers) to cache the results of a lookup, clients may not discover that a server has been removed from the IP address list for a period of time equal to the timeout associated with that particular name to IP address mapping. In fact, this problem can be worse when the client is behind an intermediate name server that has also cached the mapping: if the client queries the intermediate server just before its mapping is set to time out, the client may not check for updates for a period equal to twice the timeout. Thus, if the timeout is 10 minutes, a searcher could be without access for as long as 20 minutes.

Second, load-balancing by DNS made balancing the traffic among the servers a sometimes difficult task. When the servers are identical and capable of handling similar loads, then the problem is relatively straightforward and complicated only by the timeout problem. However, if one server is capable of handling more load than another, then the DNS system must give out the IP addresses in the right proportion. To solve this problem, we used two techniques. In the first, we simply used a single address multiple times in the list to increase the load sent to that server. For instance, if we wanted to balance load between two servers, with one-third of the load going to the first, and two-thirds going to the second, we would put one entry in the list for the first server, and two for the second. The second strategy involved manipulating the timeout values for each of the IP addresses in the list. The rationale for this approach is that if the entries are uniformly returned in lookups, then those entries with the longer timeouts will tend to get used more than those with the smaller timeouts. In practice, this approach was not as accurate as the duplication method because of an aging issue and variations in caching and client implementations. The aging issue is simple: a searcher is likely to use WebCrawler for a limited time, so an address with a long timeout will not be used in direct proportion to an address with a shorter one.

A third difficulty in balancing load is that some intermediate name servers cache data for many clients, while some only retrieve data for one client. This imbalance can be quite serious: America Online, for instance, operates several nameservers on behalf of its millions of users. To work around this problem with AOL in particular, we installed the round-robin DNS software on the AOL nameservers. However, this option is not available to most Web services.

Fourth, other problems appeared because of incorrect DNS implementations in both clients and servers. One annoying problem was that several client implementations of DNS (resolvers) had a built-in limit on the number of IP addresses that could be returned in response to a name lookup. When WebCrawler responded with 10 (or more) addresses, these clients would crash resulting in an entirely unsatisfactory user experience. To work around this issue, I modified the name server to return a maximum of six addresses per request. Another problem was that some client implementations would ignore the timeout values, and use a name to IP address mapping for as long as the client was running. This led to some clients having persistent

problems when servers were removed from the list, or worse, when all the server IP addresses were renumbered.

6.4 The Third Service

The third iteration of the service was a more radical change from the first two. This generation coincided with the deployment of the CPL-based retrieval system on large multiprocessors. As such, it was WebCrawler's first departure from running on inexpensive commodity hardware. The new systems were Silicon Graphics (SGI) Challenge multiprocessors. The basic architecture of the third service was simple: unvarying static content would be handled by one set of servers, and the dynamic search results would be served from the large multiprocessors.

This split was ideal: the large, expensive machines would be spared the simple work; they would answer queries exclusively. Meanwhile, the inexpensive commodity hardware would serve the easily replicable static content. The Apache server architecture remained the same: the information retrieval system was compiled into the Apache server, and ran continuously on the query servers. This architecture is shown in Figure 6.5.

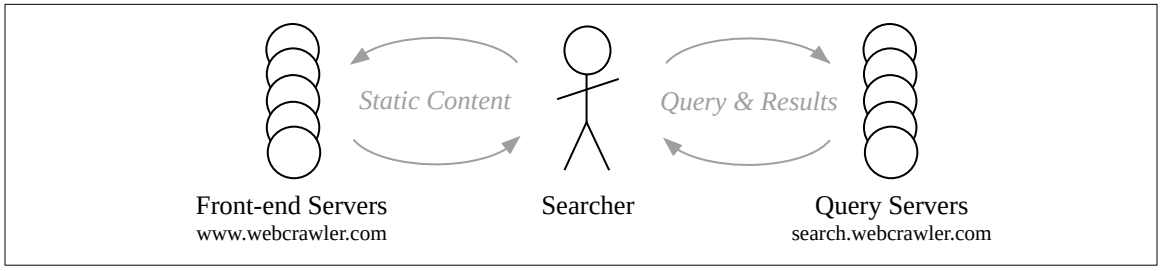

Figure 6.5: An architecture in which search results go directly to query servers.

Unfortunately, this split architecture did not last long. Within hours of its replacing the second service, it was overwhelmed: it could not handle the load. Such behavior was mystifying, because we had tested the service under a load greater than the actual real-world one. After debugging, we found that the problem originated from many slow clients connecting to and tying up state on the query servers. A client that was slow in disconnecting from the server would tie up not only an Apache process, but also the entire

memory footprint of the information retrieval library. Since this was a significant size (tens of megabytes), it did not take many slow clients to overwhelm the usable memory of a server. These slow clients were simply searchers that had connected with modems. With a workload containing millions of such connections a day, many of these were destined to cause memory problems on the server.

Fundamentally, we had failed to realize that connection management was itself an important management function, and deserved consideration in the architecture. To solve the problem, we quickly revised the live services' architecture to the one illustrated in Figure 6.6. This architecture introduced the distinction

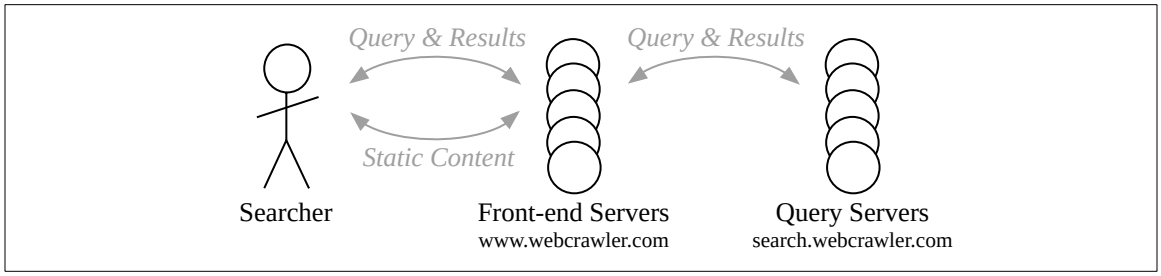

Figure 6.6: A modified architecture with query servers proxied by front-end servers.

between *front-end* (or user-facing) servers and *back-end* query servers.

In the new architecture, maintaining the HTTP connections with searchers' browsers was considered a first-class problem deserving of its own set of machines. We changed the function of the static content machines to be front ends that managed all connections with the users, including query requests. Requests for static content were handled directly by the front ends, while query requests were handed off to the query servers, or back-end servers, for further processing. When the results were available, the query servers passed them to the front ends who, in turn, passed them to the searcher. In this case, the front ends were acting as a reverse proxy for the query servers.

This change in architecture allowed the query servers to function at full speed. No longer would a query server process wait for a slow connection to time out: all the connections to the back end were well-behaved local network connections from the front ends. This architecture did, however, introduce some overhead into the query process: a searcher's request now had to be handled first by the front end, then

passed off to the back end. For simplicity, we employed HTTP on top of TCP as the transport between the front and back end. The back ends generated all the HTML that was passed back to the user.

Although the front-end load balancing issues in this architecture remained the same as earlier ones, this change did introduce a new problem: the front ends, in addition to being servers, were now clients of a distributed set of query servers and were subject to some of the same problems that WebCrawler's users were. I shall elaborate on these issues in this Chapter's load balancing and fault tolerance sections.

Another important feature of the third service, to be discussed further in Section 6.4.2, is that it was the first to be distributed among several locations. One third of the service's capacity was housed in a data center in San Francisco, California, while two thirds were hosted in America Online's data center in Reston, Virginia. This physical distribution, although problematic from a development and maintenance point of view, was useful for availability: the failure of a single data center, though it would reduce the capacity of the service, would not cause it to fail altogether.

6.4.1 Load Balancing in the Third Service

Because of some of the problems with DNS load balancing in the second service, we sought a new implementation for the third service, the LocalDirector. We also introduced a new problem: balancing the query load originating with the front ends among the query servers.

6.4.1.1 Client Load Balancing

Because of the problems inherent in the DNS load balancing solution, we went looking for a new solution. We chose the Cisco LocalDirector [Cisco 2000a] as a way to get more precise control over the distribution of load among the front-end servers. The LocalDirector is a hardware solution that acts as a single point of contact for a service. All TCP connections initially go to the LocalDirector, and the device subsequently fans them out to the front-end servers. Because of its placement between the client and server, the LocalDirector has direct control over the balancing of load among front-end servers. Figure 6.7 shows a description of the service with a LocalDirector.

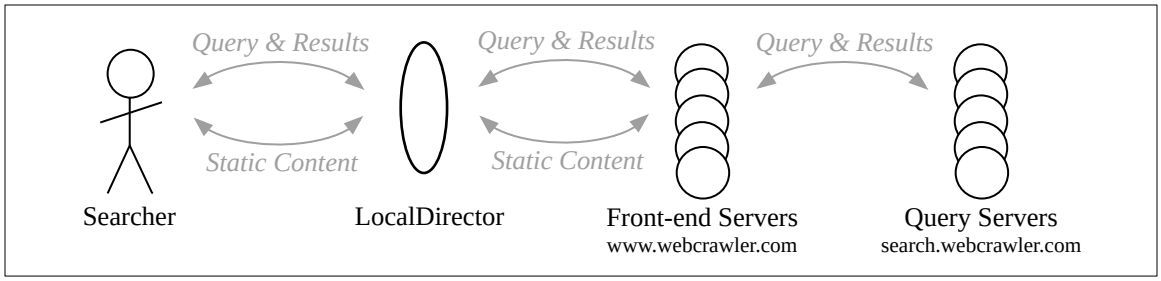

Figure 6.7: The LocalDirector directs a user's connection to the appropriate front end.

The LocalDirector has several algorithms available for distributing this load. One algorithm will route requests to the server that is responding most quickly, while another uses simple round robin. We experimented with many of them, and chose the round-robin scheme in the end because it was the most stable.

In terms of its load-balancing potential, the LocalDirector's per-connection balancing was no more effective than the per-lookup balancing of DNS round-robin. However, a combination of both is necessary for good fault tolerance, as I shall describe in Section 6.4.2.

6.4.1.2 Back-end Communication

With the new architecture, WebCrawler also needed to balance load between the front ends and the query servers. In principle, this load-balancing problem was the same as that for the front ends: some number of clients were trying to balance their load across a smaller set of servers. However, a number of factors made this problem different. First, the front and back ends shared a dedicated high-speed network (FDDI and Fast Ethernet). Second, both sets of machines were under the same administrative control. Finally, the pool of front ends was very small (approximately 20 servers), making DNS RR unlikely to work well in practice.

Because of these differences, we used a custom load balancing solution. The front ends were configured with a (changeable) list of back-end servers. At run time, they randomly chose a server from that list and sent the request to it. Since there were hundreds of client processes per front end, and they were all performing the same random calculation, the load was spread evenly across the back ends.

Another option would have been to use another LocalDirector. However, using hardware where a relatively trivial software solution would suffice seemed excessive. Also, by implementing a custom software solution, we could do a better job of achieving fault tolerance, as described in Section 6.4.2.

6.4.2 Fault Tolerance in the Third Service

The third service was the most advanced service from a fault-tolerant point of view. Not only did it protect against failures of individual nodes, but it did so in a way that resulted in a near immediate response to problems. Furthermore, the third service could tolerate the loss of a data center by redirecting load to the other center.

For local fault tolerance, the LocalDirector was a big improvement over the DNS solution used in the second service: it could drop a server immediately on detecting a problem. Thus, the problem of searchers' browsers caching stale address data is eliminated by the LocalDirector. However, because the LocalDirector is a single component, its failure can bring down the entire service. Fortunately, Cisco provides a solution whereby a pair of LocalDirectors can be used in a hot standby situation, providing an immediate solution in the case of failure.

A new element in the third service — geographic distribution among data centers — brought yet another level of fault tolerance to the service. Housing the service at a single geographic location means that an event that affects that area, an earthquake for instance, has the potential to render the service running in the data center inoperable. Even if the data center itself remains intact, damage to the network, power, and other infrastructure around the data center may make it useless. Therefore, having a duplicate service at a second location is prudent.

WebCrawler's third service had two locations: San Francisco and Reston. LocalDirectors were used to provide load balancing and fault tolerance for the servers at each location, and DNS was used to provide those features across the two locations. If one of the locations were to fail catastrophically, the DNS would be modified to remove that location.

In the case of the back-end communication, the third service's fault tolerance was quite robust. If a front end noticed a communication failure to a back end, it simply marked that server down for some period of time that depended on the exact nature of the failure. This information was shared with other processes on the same machine so it was more immediately available. After noticing the failure, the front end would simply try a different back end. This way, if a back end was available, the front ends would eventually find it and use it. In practice, this mechanism worked so well that we did not need to perform any administration to remove a back end from service: we simply turned off its server or rebooted it and let the front ends adjust automatically. Finally, an important feature of this back-end fault tolerance was that clients did not see failures in the back end, as long as one back end was available. Thus, the service seemed more robust to the searchers.

Such flexibility even extended to having a front end in one data center call a back end in a different data center. Although not desirable during times of heavy load, this capability proved to be an effective way of handling the temporary failure of a back-end at one location. In the event of a catastrophic failure of the back ends at one location, this technique could also be used to handle the residual query traffic at that location until the DNS timeouts had taken effect.

6.4.3 Problems with the Third Service

Despite its many improvements, the third service still had a few problems.

First, updating two different components of the service in synchrony to make a simple user-interface change was difficult. Because both the front ends and back ends controlled elements of the user interface, they both needed to change to make a global interface change. To address this problem, we moved query-time HTML generation to the front ends, and altered the protocol between front and back ends to transmit the query results in a compact binary form. Thus, another important design distinction was born: for consistency and ease of administration, the look-and-feel of the site was generated in one place — the front end.

Second, the convenience of using HTTP as a transport between front and back ends was useful, but not efficient. HTTP's overhead was small, but the overhead of setting up and tearing down a TCP connection for each query was simply wasteful. Worse, the operating system of our front ends, SGI's IRIX, had been optimized as a Web server to handle many incoming connections, but had not yet been optimized for the corresponding multitude of *outgoing* TCP connections required to get query results. In the end, we chose to keep the HTTP approach for simplicity's sake and also because the query results were generally too large to fit in a single packet. However, we did get an updated operating system that could efficiently handle outgoing connections as well as incoming ones.

Third, the LocalDirectors were not providing total fault tolerance: servers could fail at the application level, fooling the LocalDirectors into thinking they were still providing useful data. Early on, this happened if the HTTP server was not running on the front end: the LocalDirector would interpret a refused connection as a valid server response. Second, if a server was misconfigured and responded to every connection with an HTTP 404 (object not found), the LocalDirector would continue to think that the server was fine. Cisco subsequently upgraded the LocalDirector to account for the first problem, but the second problem had to be handled at a higher level.

Finally, the cost of hardware in the third service was dramatically higher than that of the hardware in the second service. The costliest components, indeed the only ones that mattered from a cost perspective, were the back-end query servers. Not only were they more expensive than those of the previous service, but they were more expensive on a cost per query basis as well. Although we expected this because of the larger index, we were not prepared for the cost of purchasing, and then maintaining, large mainframe computer systems. The maintenance costs showed up as higher component failure rates, higher component costs, higher software costs, and more frequent attention from administrators.

6.5 Summary

In this chapter, I described WebCrawler's service: that part of WebCrawler concerned with handling requests from users. The critical metrics in providing the service are response time and availability:

degradations in each negatively affect the searcher's experience. WebCrawler employed a distributed architecture to meet the goals of scale and fault tolerance. This architecture saw a growth of two orders of magnitude in the number of requests without significant changes.

WebCrawler's key contribution to distributed systems is to show that a reliable, scalable, and responsive system can be built using simple techniques for handling distribution, load balancing, and fault tolerance.

Chapter 7

Contributions and Future Directions

With the relentless exponential growth of the World-Wide Web, tools for locating information have become essential resources for nearly every Web user. Moreover, the increasing growth and use of the Web have created interesting challenges for the implementors of these tools and other high-volume Web sites. WebCrawler was the first comprehensive full-text search engine deployed on the Web, and has continued several years of explosive growth. In this dissertation, I have described WebCrawler's architecture, its evolution from a simple system to a complex distributed system, and many of the problems encountered along the way.

7.1 Contributions

In this dissertation, I have focused on five specific contributions to Computer Science. In this section, I shall summarize each contribution.

1. WebCrawler broke ground by being the first full-text search engine to broadly catalog the Web. Other engines that came before WebCrawler were useful, but were either not as comprehensive or more difficult to use.

   The combination of full-text search and a broad, careful selection of documents from the Web made WebCrawler a useful addition to the suite of Web tools. Using a full-text approach, where both the title

and the contents of a Web page are indexed together, proved to be an effective way of providing access to documents.

2. WebCrawler introduced the concept of carefully choosing a subset of the Web to index, rather than approaching that choice on a hit-or-miss basis. Indexing the entire Web is difficult. All search engines, including WebCrawler, catalog a subset of the Web's documents. Although this approach makes scaling the system much easier, it does create a new problem: how to select the subset.

   WebCrawler accomplishes the task of selecting the subset by crawling a larger subset and using the metadata from that crawl, along with several heuristics, to select a final subset. This approach ensures that the collection will contain a broad set of documents from many different sources. It also provides important metadata, such as link data, that can be used to improve the quality of search results. Finally, the fact that the crawler knows about many more documents than exist in the collection allows the administrator to easily choose a new policy and build a new collection.

   In contrast to other crawlers, WebCrawler uses a relational database to store metadata associated with each page on the Web. This approach allows the crawler to easily select documents that match the criteria needed for certain operations, such as finding those whose contents need to be updated, selecting those that should be included in the index, or generating a link database to enable more effective ranking in the server.

3. WebCrawler raised IR to the level that naive users were easily able to find documents on the Web. The Web's users are primarily naive and unsophisticated users. Designing an information gathering experience that works for them was a new problem and had not been addressed by past information retrieval research. User interface, query format, and query specificity — all make a difference to the searcher's experience, and WebCrawler optimizes those for the naive user.

   At first, WebCrawler employed a simple vector-space query model that required little of searchers other than that they type as many words as possible that were relevant to their query. The lack of a complex query syntax made this an appropriate choice for users that had never been exposed to an information retrieval system. As the Web grew, the pure vector model became inadequate: query recall remained

high, but precision fell as queries targeted unrelated documents. Changing the system to allow the searcher to issue boolean queries that included phrases made a significant impact on precision, but at the cost of creating a more complex user experience: to get great results, searchers now had to master a more complex query language. For those searchers that did not learn the new syntax (or chose not to use it), we developed a new ranking scheme that used link information to move more relevant documents closer to the top of the results.

Presentation is as important or, perhaps, more important to the searcher's experience than is the underlying query model. Providing the searcher with a simple, unassuming query page made a big difference. WebCrawler's query page was a simple query box with, at most, one pop-up menu and a search button. On the results page, we strove to present results in a simple format that could be easily understood by the user. Early on, we found that some searchers liked to see more information about the contents of documents, while some liked the titles-only presentation. More recently, we implemented *shortcuts*, a method for surfacing targeted content about the query in cases where such content was available.

4. WebCrawler creates a more useful relevance ranking by combining traditional full-text methods with metadata found on the Web. The results produced by information retrieval are generally ordered on the basis of the relevance of those results to the searcher's query. WebCrawler takes advantage of additional information, including the structure of the Web, to improve the relevance ranking.

   The Web is more than just a set of documents. It also contains a wealth of information about the documents themselves. WebCrawler used this information to augment the information traditionally used in a full-text setting. In particular, WebCrawler used inbound link information to refine result ranking. The rationale for this approach is simple: many queries, especially one-word queries, result in a set of results whose ranking, at least on a small-scale, is meaningless. To the extent that query results exhibit this quality, applying an external ranking based on other aspects of document relevance can make a big difference in the perceived quality of the results. In WebCrawler's case, I knew little about the relevance of a document except what was expressed in the query; so I fell back on an estimate of the document's

overall quality: the number of inbound links to a document. Because the Web contains many documents whose quality is suspect, surfacing those that are known to be good is one way of improving search results. Of course, this approach is dangerous, because the quality measure is only one interpretation and may not have any bearing on what the searcher wanted.

The external ranking approach can also be used in the future when more is known about the searcher and his search context. For instance, if the searcher was known to be a resident of San Francisco, a query such as *take-out pizza* might be augmented by an external ranking that contained *San Francisco, CA*.

5. WebCrawler takes a simple approach to achieving scale, load-balancing, and fault tolerance that is less complex than the approach that is predicted by the literature. Ironically, WebCrawler's biggest contribution to the area of distributed systems was to determine that much of the previous work did not help to build a scalable, fault-tolerant service. WebCrawler's constraints as a large system running on commercial hardware in a 24x7 environment made using these ideas difficult. WebCrawler is implemented using a simple fault-tolerant method that also simplifies growth.

   Multiple servers are used to handle the aggregate load and improve fault tolerance. These servers are dedicated to specific tasks, each with specific hardware and software requirements: front-end servers handle connections and serve static content; back-end servers handle proxied queries from the front-end servers; crawling and indexing servers are dedicated to a separate set of hardware. Effective low-overhead load balancing is achieved by using a combination of connection multiplexing (LocalDirector) and service location (DNS round-robin). Where servers interact to handle a request, the interfaces among them are kept simple and stateless. This approach allows the system as a whole to easily survive failures of some of its constituent parts: as long as enough computing hardware remains operational to handle the workload, the system will continue to operate perfectly.

7.2 Future Directions

The future of search engines is bright. As the Web continues to expand and increasing numbers of users begin to use it, the role of search tools will become even more important. At the same time, the search engine's job will become more difficult, resulting in many opportunities for research and development. These challenges can be broken up into four main areas: user experience, algorithms for high-volume information retrieval, searching more than the Web, and high-volume service architecture.

7.2.1 User Experience

The user model that WebCrawler adopted — the single search box and simplified results — will be the foundation of searching for some time to come. However, different user models have already appeared, and these will continue to evolve to meet the ever-changing demands of the Web and its searchers. One new model of search, embodied in Alexa Internet's "What's Related" feature, allows the searcher to go to a Web page and invoke a menu in the browser to find related Web sites [Alexa 2000]. Another model uses conventional search input interface, but allows the navigation of results in a three-dimensional graphic depiction of the results [Extreme 2000]. A third service, GroupFire, uses layered searches of community resources and the entire Web to obtain more focused and potentially relevant results. Putting the user in the context of a self-chosen community is a good way to implicitly obtain more information about a particular search and deliver better results [GroupFire 2000]. These solutions are emblematic of the variation that is beginning to occur in search engines. The future will need to bring a much richer variety of user interfaces to capture the way different people search and interpret results.

In addition to the different styles of searching, user models of the future should make better use of the interactivity possible with high-bandwidth connections to the Web. As these kinds of connections proliferate in the home, low-latency access to the information is possible. Real-time navigation of information spaces, with immediate feedback on user actions, should be possible soon.

7.2.2 Algorithms for High-Volume Information Retrieval

Given the ever-increasing size of search engine databases, serving answers to queries using conventional IR models is, in some sense, the greatest challenge search engines face. Maintaining the collection and indexing the large volume of information is difficult, but providing timely answers from such a collection is even more difficult. Developing algorithms for quickly answering queries from large collections will become crucial, particularly when those algorithms preserve, or even improve on, the precision and recall of current systems.

Working outside the conventional IR models is also increasingly important. As I mentioned in Chapter 2, several systems are already using completely different relevance ranking models than do conventional IR systems. The use of these models will only accelerate as traffic and collection sizes increase

7.2.3 Searching More than the Web

As the Web evolves, one of the most significant changes has been the expansion of content available from previously off-Web collections. An increasing number of sites offer access to such information not by traditional embedded links, but through a query or other dynamic user interface element. Electronic commerce sites are a good example: a Web site may detail tens of thousands of products whose descriptions are only available through a search interface. Content available behind such interfaces is unavailable to traditional search engines unless links are found to the content somewhere on the Web. Some search engines are starting to index this content by obtaining feeds of data directly from the sites that have it. The Extensible Markup Language (XML), promises to make such exchanges easier, but manual intervention will still be required in most cases because the semantics of the data varies across applications. Furthermore, because these efforts are usually dependent on business relationships, they do not assure any kind of uniform coverage. WebCrawler's shortcuts feature is one step in this direction but more work is needed to ensure a comprehensive user experience.

The proliferation of different media types further complicates this issue. No longer is meaningful Web content found solely in text or HTML format; audio, video, and animated formats are not only becom-

ing more popular, but are often legitimate results to a particular search. Finding such content with the search methods described in this dissertation is difficult, because it cannot be digested in text-based form. Several search engines have multimedia searches, but these do not provide a very rich query model and are often limited strictly to file names [Excite 2000].

7.2.4 High-Volume Service Administration

Finally, the task of high-volume Web serving deserves much attention. Currently, most high-volume operations devote significant computational and human resources to managing a complex Web service. Distributed systems that can support a variety of applications in a fault-tolerant manner are crucial. Though these systems exist today, they are neither inexpensive nor easy to administer. Focusing research efforts on software systems that can combine inexpensive hardware in a distributed network, particularly with low administration overhead, will be important. For instance, nodes should be self-configuring when they join the system, and the system should adapt to their failure. In this way, the administration overhead of the system is reduced and is based more on the initial engineering of the system.

Of course, the rapidly evolving usage and business models of Web services provide a difficult target on which to settle a particular service architecture. So whatever solutions arise, they must be flexible and take into account that the applications may change frequently.

7.3 Summary

WebCrawler's development marked the beginning of a race to provide useful navigation services to the Web's users. The profound growth of the service, and the Web as a whole, has provided an environment in which we have been able to learn a lot about developing great Web services. However, the lessons described here only provide the foundation for new ideas as we move forward. Seven years into the development of the Web, the pace of change continues to accelerate.

Bibliography

[Akamai 2000] Akamai, Inc. *Akamai: Delivering A Better Internet* http://www.akamai.com/

[Alexa 2000] Alexa, Inc. *Alexa: The Web Information Company* http://www.alexa.com/

[AltaVista 2000] AltaVista, Inc. *AltaVista - Welcome* http://www.altavista.com/

[Alteon 2000] Alteon Websystems, Inc. *Alteon Websystems Home Page* http://www.alteonwebsystems.com/

[Anderson 1995] Anderson, T.E., Culler, D.E. and Patterson, D. A case for NOW (Networks of Workstations). *IEEE Micro*, *15* (1). 54-64.

[Apache 2000] Apache Software Foundation *Apache Project* http://www.apache.org/httpd.html

[Berners-Lee 1992] Berners-Lee, T., Cailliau, R., Groff, J.F. and Pollermann, B. World-Wide Web: the information universe. *Electronic Networking: Research, Applications and Policy*, *2* (1). 52-58.

[Bharat 1998] Bharat, K., Broder, A., Henzinger, M., Kumar, P. and Venkatasubramanian, S. The Connectivity Server: fast access to linkage information on the Web. *Computer Networks and ISDN Systems*, *30* (1-7). 469-477.

[Birrell 1982] Birrell, A.D., Levin, R., Needham, R.M. and Schroeder, M.D. Grapevine: an exercise in distributed computing. *Communications of the ACM*, *25* (4). 260-274.

[Bowman 1995] Bowman, C.M., Danzig, P.B., Hardy, D.R., Manber, U. and Schwartz, M.F. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, *28* (1-2). 119-125.

[Brewer 1998] Brewer, E.A. Delivering high availability for Inktomi search engines. *SIGMOD Record*, *27* (2). 538.

[Brin 1998] Brin, S. and Page, L. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, *30* (1-7). 107-117.

[Broder 1997] Broder, A.Z., Glassman, S.C., Manasse, M.S. and Zweig, G. Syntactic clustering of the Web. *Computer Networks and ISDN Systems*, *29* (8-13). 1157-1166.

[Bush 1945] Bush, V. As We May Think. *Atlantic Monthly,* 1945, 101-108.

[Cheriton 1983] Cheriton, D.R. Local networking and internetworking in the V-system. in *Proceedings. Eighth Data Communications Symposium*, IEEE Computer Soc. Press, Silver Spring, MD, USA, 1983, 9-16.

[Cisco 2000a] Cisco, Inc. *Cisco LocalDirector 400 Series* http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml

[Cisco 2000b] Cisco, Inc. *Cisco Distributed Director* http://www.cisco.com/warp/public/cc/pd/cxsr/dd/index.shtml

[Colajanni 1997] Colajanni, M., Yu, P.S. and Dias, D.M. Scheduling algorithms for distributed Web servers. in *Proceedings of the 17th International Conference on Distributed Computing Systems (Cat. No.97CB36053)*, IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1997, 169-176.

[Colajanni 1998] Colajanni, M., Yu, P.S. and Cardellini, V. Dynamic load balancing in geographically distributed heterogeneous Web servers. in Papazoglou, M.P., Takizawa, M., Kramer, B. and Chanson, S. eds. *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, 295-302.

[Coombs 1990] Coombs, J.H., Hypertext, Full Text, and Automatic Linking. in *SIGIR*, (Brussels, 1990).

[Crestani 1998] Crestani, F., Sanderson, M., Theophylactou, M. and Lalmas, M. Short queries, natural language and spoken document retrieval: experiments at Glasgow University. in Voorhees, E.M. and Harman, D.K. eds. *Sixth Text Retrieval Conference (TREC-6) (NIST SP 500-240)*, NIST, Gaithersburg, MD, USA, 1998, 667-686.

[Croft 1989] Croft, W.B. and Turtle, H. A retrieval model for incorporating hypertext links. *SIGCHI Bulletin*. 213-224.

[Crouch 1989] Crouch, D.B., Crouch, C.J. and Andreas, G. The use of cluster hierarchies in hypertext information retrieval. *SIGCHI Bulletin*. 225-237.

[Dasgupta 1990] Dasgupta, P., Chen, R.C., Menon, S., Pearson, M.P., Ananthanarayanan, R., Ramachandran, U., Ahamad, M., LeBlanc, R.J., Appelbe, W.F., Bernabeu-Auban, J.M., Hutto, P.W., Khalidi, M.Y.A. and Wilkenloh, C.J. The design and implementation of the Clouds distributed operating system. *Computing Systems*, *3* (1). 11-46.

[Dean 1999] Dean, J. and Henziger, M.R. Finding related pages in the World-Wide Web. *Computer Networks*, *31* (11-16). 1467-1479.

[DeRose 1999] DeRose, S.J. and van Dam, A. Document structure and markup in the FRESS hypertext system. *Markup Languages: Theory & Practice*, *1* (1). 7-32.

[Dreilinger 1997] Dreilinger, D. and Howe, A.E. Experiences with selecting search engines using metasearch. *ACM Transactions on Information Systems*, *15* (3). 195-222.

[Eager 1986] Eager, D.L., Lazowska, E.D. and Zahorjan, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, *SE-12* (5). 662-675.

[Eichmann 1994] Eichmann, D., McGregor, T., Dudley, D., The RBSE Spider - Balancing Effective Search Against Web Load. in *First International World-Wide Web Conference*, (Geneva, 1994).

[Emtage 1991] Emtage, A. and Deutsch, P. archie-an electronic directory service for the Internet. in *Proceedings of the Winter 1992 USENIX Conference*, USENIX, Berkeley, CA, USA, 1991, 93-110.

[Excite 2000] Excite@Home, Inc. *Excite Home Page* http://www.excite.com/

[Extreme 2000] Excite@Home, Inc. *Excite Extreme - Home Space* http://www.exciteextreme.com/

[Fast 2000] Fast Search and Transfer ASA *All the Web, All the Time* http://www.ussc.alltheweb.com/

[Feeley 1995] Feeley, M.J., Morgan, W.E., Pighin, F.H., Karlin, A.R., Levy, H.M. and Thekkath, C.A. Implementing global memory management in a workstation cluster. *Operating Systems Review*, *29* (5). 201-212.

[Fletcher 1993] *The Jumpstation* http://www.stir.ac.uk/jsbin/js/

[Frisse 1988] Frisse, M.E. Searching for information in a hypertext medical handbook. *Communications of the ACM*, *31* (7). 880-886.

[Frisse 1989] Frisse, M.E. and Cousins, S.B. Information retrieval from hypertext: update on the dynamic medical handbook project. *SIGCHI Bulletin*. 199-212.

[Google 2000] Google, Inc. *Google* http://www.google.com/

[Gray 1996] *Web Growth Summary* http://www.mit.edu/people/mkgray/net/web-growth-summary.html

[Gribble 1999] Gribble, S.D., Welsh, M., Brewer, E.A., and Culler, D., The MultiSpace: an Evolutionary Platform for Infrastructural Services. in *Usenix Annual Technical Conference*, (Monterey, CA, 1999).

[GroupFire 2000] GroupFire, Inc. *About GroupFire* http://www.groupfire.com/

[Halasz 1988] Halasz, F.G. Reflections on NoteCards: seven issues for the next generation of hypermedia systems. *Communications of the ACM*, *31* (7). 836-852.

[Hearst 1996] Hearst, M.A. Improving full-text precision on short queries using simple constraints. in *Proceedings. Fifth Annual Symposium on Document Analysis and Information Retrieval*, Univ. Nevada, Las Vegas, NV, USA, 1996, 217-228.

[Hunt 1999] Hunt, G.C. and Scott, M.L., The Coign Automatic Distributed Partitioning System. in *3rd Symposium on Operating Systems Design and Implementation*, (New Orleans, LA, 1999).

[Infoseek 2000] Infoseek, Inc. *Infoseek Home Page* http://www.infoseek.com/

[ISC 2000] Internet Software Consortium *Internet Domain Survey* http://www.isc.org/

[Jul 1988] Jul, E., Levy, H., Hutchinson, N. and Black, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, *6* (1). 109-133.

[Kahle 1991] Kahle, B. and Medlar, A. An information system for corporate users: wide area information servers. *Online*, *15* (5). 56-60.

[Kleinberg 1998] Kleinberg, J., Authoritative sources in a hyperlinked environment. in *9th ACM-SIAM Symposium on Discrete Algorithms*, (1998).

[Koster 1994] Koster, M. ALIWEB-Archie-Like Indexing in the WEB. *Computer Networks and ISDN Systems*, *27* (2). 175-182.

[Koster 2000] *Robots Exclusion* http://info.webcrawler.com/mak/projects/robots/exclusion.html

[Lancaster 1979] Lancaster, F.W. *Information Retrieval Systems: Characteristics, Testing and Evaluation*. John Wiley and Sons, New York, 1979.

[Lazowska 1981] Lazowska, E.D., Levy, H.M., Almes, G.T., Fischer, M.J., Fowler, R.J. and Vestal, S.C. The architecture of the Eden system. *Operating Systems Review*, *15* (5). 148-159.

[Machovec 1993] Machovec, G.S. Veronica: a gopher navigational tool on the Internet. *Online Libraries and Microcomputers*, *11* (10). 1-4.

[Mauldin 1997] Mauldin, M.I. Lycos: design choices in an Internet search service. *IEEE Expert*, *12* (1). 8-11.

[McBryan 1994] McBryan, O., GENWL and WWWW: Tools for Taming the Web. in *First International World-Wide Web Conference*, (Geneva, 1994).

[Meyrowitz 1986] Meyrowitz, N. Intermedia: the architecture and construction of an object-oriented hypermedia system and applications framework. *SIGPLAN Notices*, *21* (11). 186-201.

[MIDS 2000] Matrix Information and Directory Services, Inc. *Matrix.Net Home* http://www.mids.org/

[Mockapetris 1987] *Domain Names - Implementation and Specification* http://www.ietf.org/rfc/rfc1035.txt

[Nelson 1972] Nelson, T.H. As we will think. in *Proceedings of Online 72, International Conference on Online Interactive Computing*, Online Computer Systems Ltd, Uxbridge, Middx., UK, 1972, 439-454.

[Nelson 1981] Nelson, T. *Literary Machines*. Mindful Press, Sausalito, 1981.

[Nelson 1988] Nelson, T.H. Unifying tomorrow's hypermedia. in *Online Information 88. 12th International Online Information Meeting*, Learned Inf, Oxford, UK, 1988, 1-7.

[Netcraft 2000] Netcraft *Netcraft Web Server Survey* http://www.netcraft.com/survey/

[NEXT 1992] NEXT Computer, I. *Nextstep Object-Oriented Software Programming Interface Summary : Release 3*. Addison-Wesley, 1992.

[NEXT 1994] NEXT Computer, I. *NEXTSTEP Database Kit Concepts : NeXTSTEP Developer's Library.* Addison Wesley, 1994.

[Pinkerton 1994] Pinkerton, B., Finding What People Want: Experiences with the WebCrawler. in *Second International Conference on the World Wide Web*, (Chicago, IL, 1994).

[PLS 1996] PLS, Inc. *Callable Personal Librarian* http://www.pls.com/cpl.htm

[Popek 1981] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G. LOCUS: a network transparent, high reliability distributed system. *Operating Systems Review*, *15* (5). 169-177.

[Porter 1980] Porter, M.F. An algorithm for suffix stripping. *Program*, *14* (3). 130-137.

[Salton 1975] Salton, G., Wong, A. and Yang, C.S. A vector space model for automatic indexing. *Communications of the ACM*, *18* (11). 613-620.

[Salton 1983] Salton, G., Fox, E.A. and Wu, H. Extended Boolean information retrieval. *Communications of the ACM*, *26* (11). 1022-1036.

[Salton 1989] Salton *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, 1989.

[Sampath 1985] Sampath, G. *An Introduction to Text Processing*. River Valley Publishing, Jeffersontown, KY, 1985.

[Schemers 1995] Schemers, R.J., III. lbnamed: a load balancing name server in Perl. in *Proceedings of the Ninth Systems Adminstration Conference (LISA IX)*, USENIX Assoc, Berkeley, CA, USA, 1995, 1-11.

[Schroeder 1984] Schroeder, M.D., Birrell, A.D. and Needham, R.M. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, *2* (1). 3-23.

[Schwartz 1994] Schwartz, M.F. and Pu, C. Applying an information gathering architecture to Netfind: a white pages tool for a changing and growing Internet. *IEEE/ACM Transactions on Networking*, *2* (5). 426-439.

[Selberg 1997] Selberg, E. and Etzioni, O. The MetaCrawler architecture for resource aggregation on the Web. *IEEE Expert*, *12* (1). 11-14.

[Small 1973] Small, H. Co-citation in the scientific literature: a new measure of the relationship between two documents. *Journal of the American Society for Information Sciences*, *24* (4). 265-269.

[Tanenbaum 1991] Tanenbaum, A.S., Kaashoek, M.F., van Renesse, R. and Bal, H.E. The Amoeba distributed operating system-a status report. *Computer Communications*, *14* (6). 324-335.

[Tombropoulos 1997] Tombropoulos, R.Z. Treatment Planning for Image-Guided Robotic Radiosurgery *Section on Medical Informatics*, Stanford University, Stanford, 1997, 217.

[TREC 2000] National Institute of Standards *TREC: Text Retrieval Conference* http://trec.nist.gov/

[van Dam 1969] van Dam, A., Carmody, S., Gross, T., Nelson, T., and Rice, D., A Hypertext Editing System for the /360. in *Conference in Computer Graphics*, (1969), University of Illinois.

[van Dam 1988] van Dam, A. Hypertext '87 Keynote Address. *Communications of the ACM*, *31* (7). 887-895.

[Virtual 2000] Virtual Library Group *The WWW Virtual Library* http://vlib.org/

[W3C 2000] The World-Wide Web Consortium *Libwww - the W3C Sample Code Library* http://www.w3.org/Library/

[WebCrawler 2000] Excite@Home, Inc. *WebCrawler* http://www.webcrawler.com/

[Weiping 1995] Weiping, Z. Dynamic load balancing on Amoeba. in Narashimhan, V.L. ed. *ICAPP 95. IEEE First ICA/sub 3/PP. IEEE First International Conference on Algorithms and Architectures for Parallel Processing (95TH0682-5)*, IEEE, New York, NY, USA, 1995, 355-364.

[Wilkinson 1996] Wilkinson, R., Zobel, J. and Sacks-Davis, R. Similarity measures for short queries. in Harman, D.K. ed. *Fourth Text REtrieval Conference (TREC-4) (NIST SP 500-236)*, NIST, Gaithersburg, MD, USA, 1996, 277-285.

[Yahoo 2000] Yahoo, Inc. *Yahoo! Home Page* http://www.yahoo.com/